The Structure of Authority: Why Security Is not a Separable Concern

Mark S. Miller[1,2], Bill Tulloh[3,**], and Jonathan S. Shapiro[2]

[1] Hewlett Packard Labs
[2] Johns Hopkins University
[3] George Mason University

Abstract. Common programming practice grants excess authority for the sake of functionality; programming principles require least authority for the sake of security. If we practice our principles, we could have both security and functionality. Treating security as a separate concern has not succeeded in bridging the gap between principle and practice, because it operates without knowledge of what constitutes least authority. Only when requests are made – whether by humans acting through a user interface, or by one object invoking another – can we determine how much authority is adequate. Without this knowledge, we must provide programs with enough authority to do anything they *might* be requested to do.

We examine the practice of least authority at four major layers of abstraction – from humans in an organization down to individual objects within a programming language. We explain the special role of object-capability languages – such as *E* or the proposed Oz-E – in supporting practical least authority.

1 Excess Authority: The Gateway to Abuse

Software systems today are highly vulnerable to attack. This widespread vulnerability can be traced in large part to the excess authority we routinely grant programs. Virtually every program a user launches is granted the user's full authority, even a simple game program like Solitaire. All widely-deployed operating systems today – including Windows, UNIX variants, Macintosh, and PalmOS – work on this principle. While users need broad authority to accomplish their various goals, this authority greatly exceeds what any particular program needs to accomplish its task.

When you run Solitaire, it only needs the authority to draw in its window, to receive the UI events you direct at it, and to write into a file you specify in order to save your score. If you had granted it only this limited authority, a corrupted Solitaire might be annoying, but not a threat. It may prevent you from

[**] Bill Tulloh would like to thank the Critical Infrastructure Protection Project at George Mason University for its financial support of this research.

P. Van Roy (Ed.): MOZ 2004, LNAI 3389, pp. 2–20, 2005.
© Springer-Verlag Berlin Heidelberg 2005

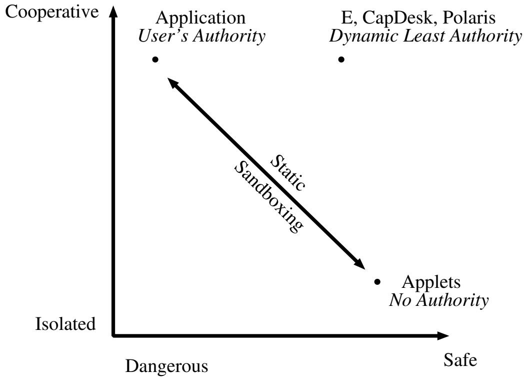

Fig. 1. Functionality vs. Security?

playing the game or lie about your score. Instead, under conventional systems, it runs with all of your authority. It can delete any file you can. It can scan your email for interesting tidbits and sell them on eBay to the highest bidder. It can install a back door and use your computer to forward spam. While Solitaire itself probably doesn't abuse its excess authority, it could. If an exploitable bug in Solitaire enables an attacker to gain control of it, the attacker can do anything Solitaire is authorized to do.

If Solitaire only needs such limited authority, why do you give it all of your authority? Well, what other choice do you have? Figure 1 shows your choices. On the one hand, you can run Solitaire as an application. Running it as an application allows you to use all the rich functionality and integration that current application frameworks have been built to support; but at the price of trusting it with all your authority. On the other hand, you could run it as an applet, granting it virtually no authority, but then it becomes isolated and mostly useless. A Solitaire applet could not even offer to save its score into a file you specify.

Sandboxing provides a middle ground between granting a program the user's full authority, and granting it no authority. Most approaches to sandboxing enable you to configure a static set of authorities (as might be represented in a policy file) to be granted to the program when it is launched. The problem is that you do not know in advance what authorities the program actually needs; the least authority needed by the program changes as execution progresses [Schneider03]. Or it might allow you to add authority incrementally, so you can trade away your safety piecemeal for functionality, but only by suffering a torrent of annoying security dialog boxes that destroy usability.

In order to successfully apply the *principle of least authority* (POLA), we need to take a different approach. Rather than trading security for functionality, we need to limit potential abuse without interfering with potential use. How far out might we move on the horizontal axis without loss of functionality or usability? Least authority, by definition, includes adequate authority to get the job done. Providing authority that is adequate means providing it in the right amount and at the right time. The key to putting POLA into practice lies in the dynamic allocation of authority; we must provide the right amount of authority just-in-time, not excess authority just-in-case.

In this paper we explain how narrow least authority can be practically achieved. We report on recent experience in building two working systems that put POLA into practice, and that demonstrate the potential for building secure systems that are both useful and usable. CapDesk is an open source proof-of-concept secure desktop and browser built from the ground up using the *E* language. Polaris is an experimental prototype from HP Labs that shows how the benefits of POLA can be applied to legacy applications.

1.1 How Much Authority Is Adequate?

How do we know how much authority a program actually needs? Surprisingly, the answer depends on architectural choices not normally thought to be related to security – the logic of designation. Consider two Unix shell commands for copying a file. In the following example, they both perform the same task, copying the file foo.txt into bar.txt, yet they follow very different logics of designation in order to do so. The result is that the least authority each needs to perform this task differs significantly.

Consider how cp performs its task:

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

Your shell passes to the cp program the two strings "foo.txt" and "bar.txt". The cp program uses these strings to determine which files it should copy.

By contrast consider how cat performs its task:

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

Your shell uses these strings to determine which files you mean to designate. Once this is resolved, your shell passes direct access to the files to cat, as open file descriptors. The cat program uses these descriptors to perform the copy.

Now consider the least authority that each one needs to perform its task.

With cp, you tell it which files to copy by passing it strings. By these strings, you mean particular files in your file system – your namespace of files. In order for cp to open the files you name, it must already have the authority to use your namespace, and it must already have the authority to read and write any file you might name. Given this way of using names, cp's *least authority* still includes all of your authority to the file system. The least authority it needs is so broad as to make achieving security hopeless.

With cat, you tell it which files to copy by passing it direct access to those two specific files. Like the cp example, you still use names in your namespace to say which files you wish to have cat copy, but these names get evaluated in your namespace prior to being passed to cat. By passing cat direct access to each file rather than giving it the file name, it does not need broad authority to do its job. Its least authority is what you'd expect – the right to read your foo.txt and the right to write your bar.txt. It needs no further access to your file system.

Currently under Unix, both cp and cat, like Solitaire, run with all your authority. But the least authority they require to copy a file differs substantially. Today's widely deployed systems use both styles of access control. They grant authority to open a file on a per-user basis, creating dangerous pools of excess authority. These same systems dynamically grant access to a file descriptor on a per-process basis. Ironically, only their support for the first style is explained as providing a form of access control.

2 Composing Complex Systems

In order to build systems that are both functional and secure, we need to provide programmers with the tools, practices, and design patterns that enable them to combine designation with authority. We can identify two main places where acts of designation occur: users designate actions through the user interface, and objects designate actions by sending requests to other objects. In both places, developers already have extensive experience with supporting such acts of designation. User-interface designers have developed a rich set of user-interface widgets and practices to support user designation [Yee04]. Likewise, programmers have developed a rich tool set of languages, patterns, and practices to support designation between objects.

That the tools for separating and integrating actions (and potentially authority) already exist should not be too surprising. Programmers use modularity and abstraction to first decompose and then compose systems in order to meet the goals of providing usability and functionality. By combining designation and authority, the same tools can be applied to meeting the goals of providing security.

2.1 The Object-Capability Model: Aligning Rights with Responsibilities

Object-oriented programming already embodies most of what is needed to provide secure programming. We introduce the object-capability model as a straightforward extension of the object model. Computer scientists, usually without any consideration for security, seeking only support for the division and composition of knowledge by abstraction and modularity, have recapitulated the logic of the object-capability model of secure computation.

In the object model, programmers decompose a system into objects and then compose those objects to get complex functionality. Designers use abstraction to carve a system into separate objects that embody those abstractions. Objects package abstractions as services that other objects can request. Each object is responsible for performing a specialized job; the knowledge required to perform the job is encapsulated within the object [Wirfs-Brock02].

Objects are composed dynamically at run-time through objects acquiring references to other objects. In order for an object to collaborate with another object, it must first come to know about the other object; the object must come to hold a reference that identifies a particular object that is available for collaboration. Objects must also have a means of communicating with these potential collaborators. References create such paths of communication. Objects can send messages along these references to request other objects to perform services on their behalf.

In the object-capability model references indivisibly combine the designation of a particular object, the means to access the object, and the right to access the object. By requiring that objects interact *only* by sending messages on references, the reference graph becomes the access graph. The object-capability model does not treat access control as a separate concern; rather it is a model of modular computation with no separate access control mechanisms.

By claiming that security is not a separable concern, we do not mean to suggest that no degree of separation is possible. Dijkstra's original modest suggestion – that we temporarily separate concerns as a conceptual aid for reasoning about complex systems [Dijkstra74] – is applicable to security as it is to correctness and modularity. What we wish to call into question, however, is the conventional practice of treating access control concerns – the allocation of access rights within a system – separately from the practice of designing and building systems. One cannot make a system more modular by adding a modularity module. Security, again like correctness and modularity, must first and foremost be treated as part of the process of de-composing and composing software systems. Access control in the object-capability model derives from the pursuit of abstraction and modularity. Parnas' principle of *information hiding* [Parnas72] in effect says our abstractions should hand out information only on a *need to know* basis. POLA simply adds that authority should be handed out only on a *need to do* basis. Modularity and security each require both.

2.2 The Fractal Locality of Knowledge: Let "Knows-About" Shape "Access-to"

What the object model and object-capability model have in common is a logic that explains how computational decisions dynamically determine the structure of knowledge in our systems – the topology of the "knows-about" relationship. The division of knowledge into separate objects that cooperate through sending requests creates a natural sparseness of knowledge within a system. The object-capability model recognizes that this same sparseness of knowledge, created in pursuit of good modular design, can be harnessed to protect objects from one another. Objects that do not know about one another, and consequently have no way to interact with each other, cannot cause each other harm. By combining designation with authority, the logic of the object-capability model explains how computational decisions dynamically determine the structure of authority in our systems – the topology of the "access-to" relationship.

What we typically find in computational systems is a hierarchical, recursive division of responsibility and knowledge. Computation, like many complex systems, is organized into a hierarchic structure of nested levels of subsystems. We can identify four majors layers of abstraction: at the organizational level systems are composed of users; at the user level, systems are composed of applications; at the application level, systems are composed of modules; at the module level, systems are composed of objects. Each layer of abstraction provides a space where the subsystems at that level can interact, while at the same time significantly limiting the intensity of interaction that needs to occur across these layers.

Computer scientist and Nobel Laureate in Economics, Herbert Simon, argues that this hierarchic nesting of subsystems is common across many types of complex systems [Simon62]. Complex systems frequently take the form of a hierarchy, which can be decomposed into subsystems, and so on; "Hierarchy," he argues, "is one of the central structural schemes that the architecture of complexity uses." Simon shows how this nesting of subsystems occurs across many different types of systems. For example, in the body, we have cells that make up tissues, that make up organs, that make up organisms. As Simon notes, the nesting of subsystems helps bring about a sparseness of knowledge between subsystems. Each subsystem operates (nearly) independently of the detailed processes going on within other subsystems; components within each level communicate much more frequently than they do across levels. For example, my liver and my kidney in some sense know about each other; they use chemical signals to communicate with one another. Similarly, you and I may know about each other, using verbal signals to communicate and collaborate with one another. On the other hand we would be quite surprised to see my liver talk to your kidneys.

While the nesting of subsystems into layers is quite common in complex systems, it provides a rather static view of the knowledge relationship between layers. In contrast, within layers we see a much more dynamic process. Within layers of abstraction, computation is largely organized as a dynamic subcontracting network. Subcontracting organizes requests for services among clients and providers. Abstraction boundaries between clients and providers enable separation of concerns at the local level. They help to further reduce knows-about relationships, not just by thinning the topology of who knows about whom, but also by reducing how much they know about each other [Tulloh02]. Abstraction boundaries allow the concerns of the client (the reasons why it requests a particular service) to be separated from the concerns of the provider (how it implements a particular service). Abstraction boundaries, by hiding implementation details, allow clients to ignore distractions and focus on their remaining concern. Applied to authority, abstraction boundaries protect clients from further unwanted details; by denying the provider authority that is not needed to do its job, the client does not need to worry as much about the provider's intent. Even if the intent is to cause harm, the scope of harm is limited.

Simon's fellow Nobel Laureate in Economics, Friedrich Hayek, has argued that the division of knowledge and authority through dynamic subcontracting relationships is common across many types of complex systems [Hayek45, Hayek64]. In particular, Hayek has argued that the system of specialization and exchange that generates the division of labor in the economy is best understood as creating a division of knowledge where clients and providers coordinate their plans based on local knowledge. Diverse plans, Hayek argues, can be coordinated only based on local knowledge; no one entity possesses the knowledge needed to coordinate agents' plans. Similarly no one entity has the knowledge required to allocate authority within computer systems according to the principle of least authority. To do this effectively, the entity would need to understand the duties of every single abstraction in the system, at every level of composition. Without understanding the duties of each component, it's impossible to understand what would be the least authority needed for it to carry out these duties. "Least" and "duties" can only be understood locally.

3 The Fractal Nature of Authority

The access matrix model [Lampson74, Graham72] has proven to be one of the most durable abstractions for reasoning about access control in computational systems. The access matrix provides a snapshot of the protection state of a particular system, showing the rights (the filled-in cells) that active entities (the rows) have with respect to protected resources (the columns). While not specifically designed for reasoning about least authority, we adapt the access matrix model to show how the consistent application of POLA across levels can significantly reduce the ability of attackers to exploit vulnerabilities. We show how POLA applied at the four major layers of abstraction – from humans in an organization down to individual objects within a programming language – can achieve a multiplicative reduction in a system's attack surface.

The access matrix is normally used to depict only permissions – the direct access rights an active entity has to a resource, as represented by the system's protection state. Since we wish to reason about our overall exposure to attack, in this paper access matrices will instead depict authority. Authority includes both direct permissions and indirect causal access via the permitted actions of intermediary objects [Miller03]. It is unclear whether Saltzer and Schroeder's famous "principle of least privilege" [Saltzer75] should be understood as "least permission" or "least authority". But it is clear that, to minimize our exposure, we must examine authority. To avoid confusion, when we wish to speak specifically about the structure of permissions, we will instead refer to the "access graph" – an alternate visualization in which permissions are shown as arcs of the graph [Bishop79].

Howard, Pincus and Wing [Howard03] have introduced the notion of an attack surface as a way to measure, in a qualitative manner, the relative security of various computer systems. This multi-dimensional metric attempts to capture the notion that system security depends not only on the number of specific bugs found, but also on a system's "process and data resources" and the actions that can be executed on these resources. These resources can serve as either targets or enablers depending on the nature of the attack. Attackers gain control over the resources through communication channels and protocols; access rights place constraints on which resources can be accessed over these channels.

They define the attack surface of a system to be the sum of the system's attack opportunities. An attack is a means of exploiting a vulnerability. Attack opportunities are exploitable vulnerabilities in the system weighted by some notion of how exploitable the vulnerability is. By treating exploitability not just as a measure of how likely a particular exploit will occur, but as a measure of the extent of damage that can occur from a successful attack, we can gain insight into the role least authority can play in reducing a system's attack surface.

We can use the area of the cells within the access matrix to visualize, in an abstract way, the attack surface of a system. Imagine that the heights of the rows were resized to be proportional to the likelihood that each active entity could be corrupted or confused into enabling an attack. Imagine that the width of the columns were resized to be proportional to the damage an attacker with authority to that asset could cause. Our overall attack surface may, therefore, be approximated as the overall filled-in area of the access matrix. (In this paper, we do not show such resizing, as the knowledge needed to quantify these issues is largely inaccessible).

By taking this perspective and combining it with Simon's insight that complex systems are typically organized into nested layers of abstractions, we can now show how applying POLA to each level can recursively reduce the attack surface of a system. While it is well-recognized that the active entities in an access matrix can be either people or processes, the precise relationship between them is rarely recognized in any systematic way. We show how the same nesting of levels of abstraction, used to organize system functionality, can be used to organize the authority needed to provide that functionality.

We now take a tour through four major levels of composition of an example system:

1. among the people within an organization
2. among the applications launched by a person from their desktop
3. among the modules within an application
4. among individual language-level "objects"

Within this structure, we show how to practice POLA painlessly at each level, and how these separate practices compose to reduce the overall attack surface multiplicatively.

Some common themes will emerge in different guises at each level:

- the relatively static nesting of subsystems
- the dynamic subcontracting networks within each subsystem
- the co-existence of legacy and non-legacy components
- the limits placed on POLA by the "TCB" issue, explained below, and by legacy code.

3.1 Human-Granularity POLA in an Organization

When an organization is small, when there's little at stake, or when all of an organization's employees are perfectly non-corruptible and non-confusable, the internal distribution of excess authority creates few vulnerabilities. Otherwise, organizations practice separation of responsibilities, need to know, and POLA to limit their exposure.

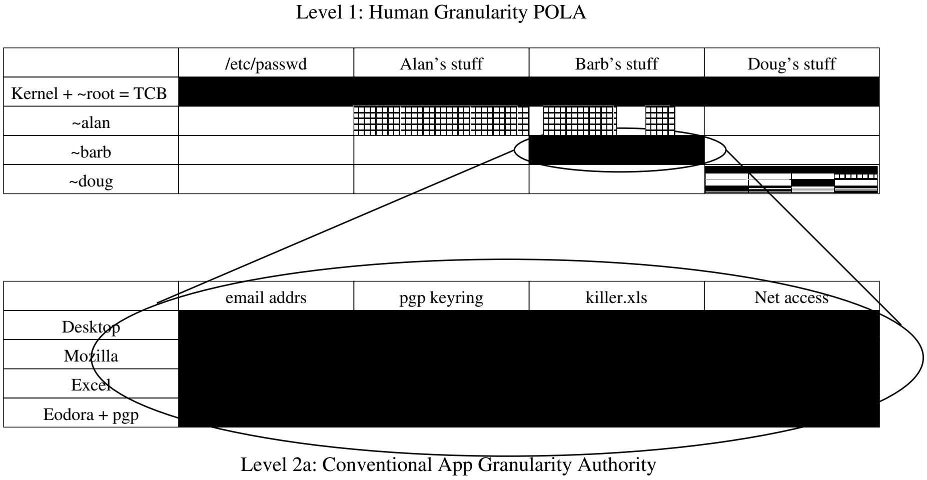

Fig. 2. Barb’s situation

The figure labeled “Level 1” (in Figure 2) uses the access matrix to visualize how conventional operating systems support POLA within a human organization. Alan (the “˜alan” account) is given authority to access all of Alan’s stuff, and likewise with Barb and Doug. In addition, because Barb and Alan are collaborating, Barb gives Alan authority to access some of her stuff. The organization should give Alan those authorities needed for him to carry out his responsibilities. This can happen in both a hierarchical manner (an administrator determining which of the organization’s assets are included in “Alan’s stuff”) and a decentralized manner (by Barb, when she needs to collaborate with Alan on something) [Abrams95]. If an attacker confuses Alan into revealing his password, the assets the attacker can then abuse are limited to those entrusted to Alan. While better training or screening may reduce the likelihood of an attack succeeding, limits on available authority reduce the damage a successful attack can cause.

To the traditional access matrix visualization, we have added a row representing the TCB, and a column, labeled /etc/passwd, which stands for resources which are effectively part of the TCB. Historically, “TCB” stands for “Trusted Computing Base”, but is actually about vulnerability rather than trust. To avoid the confusion caused by the traditional terminology, we here define TCB as that part of a system that everything in *that* system is necessarily vulnerable to. In a traditional timesharing context, or in a conventional centrally-administered system of accounts within a company, the TCB includes the operating system kernel, the administrator accounts, and the administrators. The TCB provides the mechanisms used to limit the authority of the other players, so all the authority it manages is vulnerable to the corruption or confusion of the TCB itself. While much can be done to reduce the likelihood of an exploitable flaw in the TCB – primarily by making it smaller and cleaner – ultimately any centralized system will continue to have this Achilles heel of potential full vulnerability. (Decentralized systems can escape this centralized vulnerability, and distributed languages like *E* and Oz should support the patterns needed to do so. But this issue is beyond the scope of this paper.)

3.2 Application-Granularity POLA on the Desktop

With the exception of the TCB problem, organizations have wrestled with these issues since long before computers. Operating System support for access control evolved largely in order to provide support for the resulting organizational practices [Moffett88]. Unfortunately, conventional support for these practices was based on a simplifying assumption that left us exposed to viruses, worms, Trojan horses, and the litany of problems that, now, regularly infest our networks. The simplifying assumption? When Barb runs a program to accomplish some goal, such as killer.xls, an Excel spreadsheet, conventional systems assume the program is a perfectly faithful extension of Barb's intent. But Barb didn't write Excel or killer.xls.

Zooming in on Level 1 brings us to Level 2a (Figure 2), showing the conventional distribution of authority among the programs Barb runs; they are all given all of Barb's authority. If Excel is corruptible or confusable – if it contains a bug allowing an attacker to subvert its logic for the attacker's purposes – then anything Excel may do, the attacker can do. The attacker can abuse all of Barb's authority – sending itself to her friends and deleting her files – even if her operating system, her administrator, and Barb herself are all operating flawlessly. Since all the assets entrusted to Barb are exposed to exploitable flaws in any program she runs, all her programs are in her TCB. If Barb enables macros, even her documents, like killer.xls, would be in her TCB as well. How can Barb reduce her exposure to the programs she runs?

Good organizational principles apply at many scales of organization. If the limited distribution of authority we saw in Level 1 is a good idea, can we adopt it at this level as well?

Level 2b (Figure 3) is at the same "scale" as Level 2a, but depicts Doug's situation rather than Barb's. Like Barb, Doug launches various applications interactively from his desktop. Unlike Barb, let's say Doug runs his desktop and these apps in such a way as to reduce his exposure to their misbehavior. One possibility would be that Doug runs a non-conventional OS that supports finer-grained POLA [Dennis66, Hardy85, Shapiro99]. In this paper, we explore a surprising alternative – the use of language-based security mechanisms, like those provided by *E* [Miller03] and proposed for Oz by the paper on Oz-E in this volume [Spiessens-VanRoy05]. We will explain how Doug uses CapDesk and Polaris to reduce his exposure while still running on a conventional OS. But first, it behooves us to be clear about the limits of this approach. (In our story, we combine the functionality of CapDesk and Polaris, though they are not yet actually integrated. Integrating CapDesk's protection with that provided by an appropriate secure OS would yield yet further reductions in exposure, but these are beyond the scope of this paper.)

CapDesk [Stiegler02] is a capability-secure distributed desktop written in *E*, for running *caplets* – applications written in *E* to be run under CapDesk.

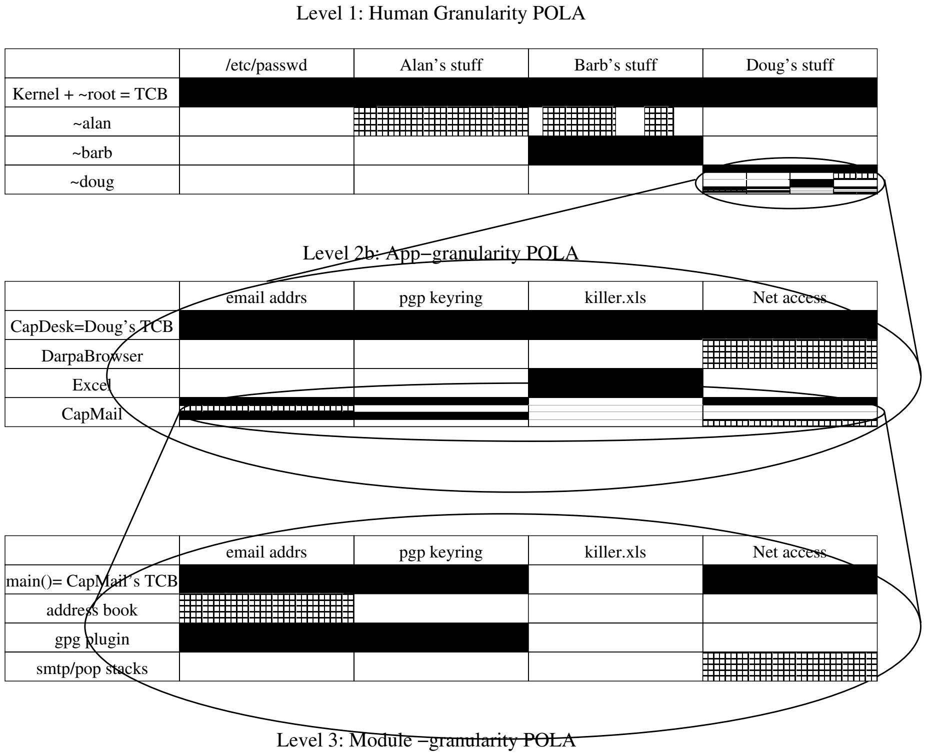

Fig. 3. Doug's situation

CapDesk is the program Doug uses to subdivide his authority among these apps. To do this job, CapDesk's least authority is all of Doug's authority. Doug launches CapDesk as a conventional application in his account, thereby granting it all of his authority.

Doug is no less exposed to a flaw in CapDesk than Barb is to a flaw in each app she runs. CapDesk is part of Doug's TCB; but the programs launched by CapDesk are not. Doug is also no less exposed to an action taken by Barb, or one of her apps, than he was before. If the base OS does not protect his interests from actions taken in other accounts, then the whole system is in his TCB. Without a base OS that provides foundational protection, no significant reduction of exposure by other means is possible. So let us assume that the base OS *does* at least provide effective per-account protection. For any legacy programs that Doug installs or runs in the conventional manner – outside the CapDesk framework – Doug is no less exposed than he was before. All such programs remain in his TCB. If "˜doug" is corrupted by this route, again, CapDesk's protections are for naught.

However, if the integrity of "˜doug" survives these threats, Doug can protect the assets entrusted to him from the programs he runs by using CapDesk + Polaris to grant them least authority. This granting must be done in a usable fashion – unusable security won't be used, and security which isn't used doesn't protect anyone. As with cat, the key to usable POLA is to bundle authority with designation [Yee02, Yee04]. To use Excel to edit killer.xls, Doug must somehow designate this file as the one he wishes to edit. This may happen by double clicking on the file, by selecting it in an open file dialog box, or by drag-and-drop. (Drag-and-drop is supported by CapDesk, but not yet by Polaris.) The least authority Excel needs includes the authority to edit this one file, but typically not any other interesting authorities. Polaris [Stiegler04] runs each legacy app in a separate account, created by Polaris for this purpose, which initially has almost no authority. Under Polaris, Doug's act of designation dynamically grants Excel the authority to edit this one file. Polaris users regularly run with macros enabled, since they no longer live in fear of their documents.

3.3 Module-Granularity POLA Within a Caplet

Were we to zoom into Doug's legacy Excel box, we'd find that there is no further reduction of authority within Excel. All the authority granted to Excel as a whole is accessible to all the modules of which Excel is built, and to the macros in the spreadsheets it runs. Should the math library's sqrt function wish to overwrite killer.xls, nothing will prevent it. At this next smaller scale (the third level) we'd find the same full-authority picture previously depicted as Level 2a.

Caplets running under CapDesk do better. The DarpaBrowser is a web browser caplet, able to use a potentially malicious plug-in as a renderer. Although this is an actual example, the DarpaBrowser is "actual" only as a proof of concept whose security properties have been reviewed [Wagner02] – not yet as a practical browser. We will instead zoom in to the hypothetical email client caplet, CapMail. All the points we make about CapMail are also true for the DarpaBrowser, but the email client makes a better expository example. Of the programs regularly run by normal users – as opposed to system administrators or programmers – the email client is the worst case we've identified. Its least authority includes a dangerous combination of authorities. Doug would grant some of these authorities – like access to an smtp server – by static configuration, rather than dynamically during each use.

When Doug decides to grant CapMail these authorities, he's deciding to rely on the authors of CapMail not to abuse them. However, the authors of CapMail didn't write every line of code in CapMail – they reused various reusable libraries written by others. CapMail should not grant its crypto library the authority needed to read your address book and send itself to your friends.

Zooming in on the bottom row of Level 2b brings us to Level 3. A caplet has a startup module that's the moral equivalent of the C or Java programmer's "main()" function. CapDesk grants to this startup module all the authority it grants to CapMail as a whole. If CapMail is written well, this startup module should do essentially nothing but import the top level modules constituting the bulk of CapMail's logic, and grant each that portion of CapMail's authority that it needs during initialization. This startup module is CapMail's TCB – its

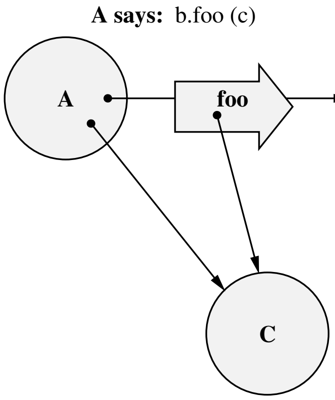

Fig. 4. Level 4: Object Granularity POLA

logic brings about this further subdivision of initial authority, so all the assets entrusted to CapMail as a whole are vulnerable to this one module.

When a CapMail user launches an executable caplet attachment, CapMail should ask CapDesk to launch it, in which case it would only be given the authority the user grants by explicit actions. CapMail users would no longer need to fear executable attachments. (The DarpaBrowser already demonstrates equivalent functionality for downloaded caplets.)

3.4 Object-Granularity POLA

At Level 3, we again see the co-existence of boxes representing legacy and non-legacy. For legacy modules, we've been using a methodology we call "taming" to give us some confidence, under some circumstances, that a module doesn't exceed its proper authority [Miller02]. Again, for these legacy boxes, we can achieve no further reduction of exposure within the box. Zooming in on a legacy box would again give a full authority picture like that previously depicted as Level 2a, but at the fourth level. Zooming in on a non-legacy box takes us instead to a picture of POLA at Level 4 (Figure 4). This is our finest scale application of these principles – at the granularity of individual programming language objects. These are the indivisible particles, if you will, from whose logic our levels 2 and 3 were built.

By "object", we do not wish to imply a class-based system, or built-in support for inheritance. We are most comfortable with the terms and concepts of object-oriented programming, but the logic explained below applies equally well to lambda calculus with local side effects [Morris73, Rees96], Actors [Hewitt77], concurrent logic/constraint programming [Miller87], and the Pi calculus. Oz's semantics already embodies this logic. (The following explanation skips some details; see [Miller03] for a precise statement of the object-capability model.)

Let's examine all the ways in which object B can come to know about, i.e., hold a reference to, object C.

1. *By Introduction.* If B and C already exist, and B does not already know about C, then the only way B can come to know about C is if there exists an object A that
   - already knows about C
   - already knows about B
   - decides to share with B her knowledge of C.

   In object terms, if A has variables in her scope, b and c, that hold references to B and C, then A may send a message to B containing a copy of her reference to C as an argument: "b.foo(c)". Unlike the cp example, and like the cat example, A does not communicate the string "c" to B. B does not know or care what name A's code uses to refer to C.
2. *By Parenthood.* If B already exists and C does not, then, if B creates C, at that moment B is the only object that knows about C (has a reference to C). From there, other objects can come to know about C only by inductive application of these steps. Parenthood may occur by normal object instantiation, such as calling a constructor or evaluating a lambda expression, or by import, which we return to below.
3. *By Endowment.* If C already exists and B does not, then, if there exists an object A that already knows about C, A can create B such that B is born already endowed with knowledge of C. B might be instantiated by lambda evaluation, in which case a variable "c" which is free within B might be bound to C within B's creation context, as supplied by A. Or A might instantiate B by calling a constructor, passing C as an argument. If A creates module B by importing data describing B's behavior (in Oz, a functor file), then A's importing context must explicitly provide bindings for all the free variables in this functor file, where these values must already be accessible to A. The imported B module must not be able to magically come into existence with authorities not granted by its importer. (The underlying logic of the Oz module manager seems ideally designed to support this, though various details need to be fixed.)
4. *By Initial Conditions.* For purposes of analysis, there's always a first instant of time. B might already know about C when our universe of discourse came into existence.

By these rules, only connectivity begets connectivity – new knows-about relationships can only be brought about from existing knows-about relationships. Two disjoint subgraphs can never become connected, which is why garbage collection can be transparent. More interestingly, if two subgraphs are almost disjoint, they can only interact or become further connected according to the decisions of those objects that bridge these two subgraphs.

An object can affect the world outside itself by sending messages on references it holds. An object can be affected by the world outside itself by receiving messages from objects that hold a reference to it. If objects have no possibility of causal access by other means, such as global variables, then an object's permissions are the references it holds. The object reference graph becomes the access graph. Together with designational integrity (also known as the y-property [Close03]), and support for defensive correctness, explained in the paper on Oz-E in this volume, these *are* the rules of object-capability security [Spiessens-VanRoy05].

But knowing the rules of chess is distinct from knowing how to play chess. The practice of using these rules well to write secure code is known as capability discipline. As we should expect, capability discipline is mostly just an extreme form of good modular software engineering practice. Of the people who have learned capability discipline, several have independently noticed that they find themselves following capability discipline even when writing programs for which security is of no concern. We find that it consistently leads to more modular, more maintainable code.

Table 1. Security as extreme modularity

Good software engineering	**Capability discipline**
Responsibility driven design	Authority driven design
Omit needless coupling	Omit needless vulnerability
assert(..) preconditions	Validate inputs
Information hiding	Principle of Least Authority
Designation, need to know	Permission, need to do
Lexical naming	No global name spaces
Avoid global variables	Forbid mutable static state
Procedural, data, control, ···	··· and access abstractions
Patterns and frameworks	Patterns of safe cooperation
Say what you mean	Mean only what you say

This completes the reductionist portion of our tour. We have seen many issues reappear at each level of composition. Let's zoom back out and see what picture emerges.

3.5 Nested TCBs Follow the Spawning Tree

The nesting of subsystems within each other corresponds to a spawning tree. The TCB of each system creates the initial population of subsystems within it, and endows each with their initial portion of the authority granted to this system as a whole. The organization decides what Alan's responsibilities are, and its administrators configure Alan's initial authorities accordingly. Doug uses CapDesk to endow CapMail with access to his smtp server by static configuration. CapMail's main() grants this access to its imported smtp module. A lambda expression with a free variable "c" evaluates to a closure whose binding for "c" is provided by its creation context. The spawning tree has the hierarchic structure that Herbert Simon explains as common to many kinds of complex systems [Simon62]. Mostly static approaches to POLA, such as policy files, may succeed at mirroring this structure.

3.6 Subcontracting Forms Dynamic Networks of Authority

Among already instantiated components, we see a network of subcontracting relationships whose topology dynamically changes as components make requests of each other. Barb finds she needs to collaborate with Alan; or Doug selects killer.xls in an open file dialog box; or object A passes a reference to object C as an argument in a message to object B. In all these cases, by following capability discipline, the least authority the subcontractor needs to perform a request can often be painlessly conveyed along with the designations such requests must already carry. The least adjustments needed to the topology of the access graph are often identical to the adjustments made anyway to the reference graph.

3.7 Legacy Limits POLA, but Can Be Managed Incrementally

Among the subsystems within each system, we must engineer for a peaceful co-existence of legacy and non-legacy components. Only such co-existence enables non-legacy systems to be adopted incrementally. For legacy components, POLA can and indeed must be practiced separately. For example, Polaris restricts the authority available to killer.xls without modifying the spreadsheet, Excel, or WindowsXP. However, we can only impose POLA on the legacy component – we cannot enable the component to further practice POLA with the portion of its authority it grants to others, or to sub-components of itself. Following initial adoption, as we replace individual legacy components, we incrementally increase our safety.

3.8 Nested POLA Multiplicatively Reduces Attack Surface

The cross-hatching within the non-legacy boxes we did not zoom into – such as the "~alan" row – represents our abstract claim that exposure was further reduced by practicing POLA within these boxes. The claim can now be explained by the fine structure shown in the non-legacy boxes we did zoom into – such as the "~doug" box. Whatever fraction of the attack surface we removed at each level by practicing POLA; these effects compose to create a multiplicative reduction in our overall exposure. Secure languages used according to capability discipline can extend POLA to a much finer grain than is normally sought. By spanning a large enough range of scales, the remaining attack surface resembles the area of a fractal shape which has been recursively hollowed out. Although we do not yet know how to quantify these issues, we hope any future quantitative analysis of what is practically achievable will take this structure into account.

4 Conclusions

To build useful and usable systems, software engineers build sparse-but-capable dynamic structures of knowledge. The systems most successful at supporting these structures – such as object, lambda, and concurrent logic languages – exhibit a curious similarity in their logic of designation. Patterns of abstraction and modularity divide knowledge, and then use these designators to compose divided knowledge to useful effect. Software engineering discipline judges these design patterns partially by their support for the principle of information hiding – by the sparseness of the knowledge structures they build from these designators.

To build useful, usable, and safe general purpose systems, we must leverage these impressive successes to provide correspondingly sparse-but-capable dynamic structures of authority. Only authority structures aligned with these knowledge structures can both provide the authority needed for use while narrowly limiting the excess of authority available for abuse. To structure authority in this way, we need "merely" make a natural change to our foundations, and a corresponding natural change to our software engineering discipline.

Capability discipline judges design patterns as well by their support for the principle of least authority – by the sparseness of the authority structures they build from these permissions. Not only is this change needed for safety, it also increases the modularity needed to provide ever greater functionality.

An object-capability language can extend this structuring of authority down to finer granularities, and therefore across more scales, than seem practical by other means. The paper on Oz-E in this volume explores how Oz can become such a language [Spiessens-VanRoy05]. In this paper we have presented a proof-of-concept system – consisting of *E*, CapDesk, and Polaris – that explains an integrated approach for using such foundations to build general purpose systems that are simultaneously safer, more functional, more modular, and more usable than is normally thought possible.

Acknowledgements

For various comments and suggestions, we thank Per Brand, Scott Doerrie, Jack High, Alan Karp, Christian Scheideler, Swaroop Sridhar, Fred Spiessens, Terry Stanley, and Marc Stiegler. We thank Norm Hardy for first bringing to our attention the intimate relationship between knowledge and authority in computation.

References

[Abrams95] Marshall Abrams and David Bailey. "Abstraction and Refinement of Layered Security Policy." In Marshall D. Abrams, Sushil Jajodia, and Harold J. Podell, eds. Information Security: An Integrated Collection of Essays. IEEE Computer Society Press. Los Alamitos, CA 1995: 126-136.

[Bishop79] Matt Bishop and Lawrence Snyder. "The Transfer of Information and Authority in a Protection System." Proc. 7th ACM Symposium on Operating Systems Principles (Operating Systems Review **13**(4)), 1979, pp. 45–54.

[Close03] Tyler Close "What Does the 'y' Refer to", 2003.
http://www.waterken.com/dev/YURL/Definition/

[Dennis66] J.B. Dennis, E.C. Van Horn. "Programming Semantics for Multiprogrammed Computations", Communications of the ACM, 9(3):143-155, March 1966.

[Dijkstra74] Edsger W. Dijkstra, “On the role of scientific thought”, EWD 447, 1974, appearing in E.W.Dijkstra, Selected Writings on Computing: A Personal Perspective, Springer Verlag, 1982.

[Graham72] Graham, G.S., and Denning, P.J. Protection-principles and practice. Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., pp. 417-429.

[Hardy85] N. Hardy. “The KeyKOS Architecture” ACM Operating Systems Review, September 1985, p. 8–25.
http://www.agorics.com/Library/KeyKos/architecture.html

[Hayek45] Friedrich A. Hayek “Use of Knowledge in Society” American Economic Review, XXXV, No. 4; September, 1945, 519-30.
http://www.virtualschool.edu/mon/Economics/HayekUseOfKnowledge.html

[Hayek64] Friedrich A. Hayek “The Theory of Complex Phenomena”, 1964, in Bunge, editor, The Critical Approach to Science and Philosophy.

[Hewitt77] Carl Hewitt, Henry Baker, “Actors and Continuous Functionals” , MIT-LCS-TR-194, 1977. Locality Laws online at
http://www.erights.org/history/actors.html

[Howard03] Michael Howard, Jon Pincus, Jeannette M. Wing. “Measuring Relative Attack Surfaces” Proceedings of the Workshop on Advanced Developments in Software and Systems Security, 2003.

[Lampson74] Butler W. Lampson. “Protection” ACM Operating Systems Review. 8:1, Jan. 1974.

[Miller87] M. S. Miller, D. G. Bobrow, E. D. Tribble, J. Levy, “Logical Secrets” Concurrent Prolog: Collected Papers, E. Shapiro (ed.), MIT Press, Cambridge, MA, 1987.

[Miller02] Mark S. Miller, “A Theory of Taming”, 2002.
http://www.erights.org/elib/legacy/taming.html

[Miller03] Mark S. Miller, Jonathan S. Shapiro, “Paradigm Regained: Abstraction mechanisms for access control” , Proceedings of ASIAN’03, Springer Verlag, 2003. Complete version online at
http://www.erights.org/talks/asian03/index.html

[Moffett88] Jonathan D. Moffett and Morris S. Sloman, “The Source of Authority for Commercial Access Control” IEEE Computer, February 1988.

[Morris73] J. H. Morris. “Protection in Programming Languages” CACM 16(1) p. 15–21, 1973.
http://www.erights.org/history/morris73.pdf

[Parnas72] David L. Parnas. “On the Criteria To Be Used in Decomposing a System into Modules.” Communications of the ACM, Vol. 15, No. 12, December 1972: pp. 1053–1058.

[Rees96] J. Rees, A Security Kernel Based on the Lambda-Calculus. MIT AI Memo No. 1564. MIT, Cambridge, MA, 1996.
http://mumble.net/jar/pubs/secureos/

[Saltzer75] J. H. Saltzer, M. D. Schroeder, “The Protection of Information in Computer Systems” Proceedings of the IEEE 63(9), September 1975, p. 1278–1308.

[Schneider03] Fred B. Schneider. “Least Privilege and More.” IEEE Security & Privacy, September/October, 2003: 55-59.

[Simon62] Herbert S. Simon, “The Architecture of Complexity: Hierarchic Systems” Proceedings of the American Philosophical Society, 106:467-482, 1962

[Shapiro99] J. S. Shapiro, J. M. Smith, D. J. Farber. “EROS: A Fast Capability System” Proceedings of the 17th ACM Symposium on Operating Systems Principles, December 1999, p. 170–185.

[Spiessens-VanRoy05] Fred Spiessens and Peter Van Roy, “The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language”, Lecture Notes in Artificial Intelligence, Vol. 3389, Springer Verlag, 2005.

[Stiegler02] M. Stiegler, M. Miller. “A Capability Based Client: The DarpaBrowser”, 2002.
http://www.combex.com/papers/darpa-report/index.html

[Stiegler04] Marc Stiegler, Alan H. Karp, Ka-Ping Yee , Mark Miller, “Polaris: Virus Safe Computing for Windows XP”, HP Tech Report, in preparation.

[Tulloh02] Bill Tulloh, Mark S. Miller. “Institutions as Abstraction Boundaries”, To appear in Economics, Philosophy, & Information Technology: The Intellectual Contributions of Don Lavoie, George Mason University, Fairfax, VA. 2002.
http://www.erights.org/talks/categories/

[Wagner02] David Wagner, Dean Tribble, “A Security Analysis of the Combex DarpaBrowser Architecture”, 2002.
http://www.combex.com/papers/darpa-review/index.html

[Wirfs-Brock02] Rebecca Wirfs-Brock and Alan McKean. Object Design: Roles, Responsibilities, and Collaborations. Addison-Wesley, 2002.

[Yee02] Ka-Ping Yee, “User Interaction Design for Secure Systems”, In Proceedings of the International Conference on Information and Communications Security, 2002. Complete version online at http://zesty.ca/pubs/csd-02-1184.ps

[Yee04] Ka-Ping Yee, “Aligning Usability and Security”, In IEEE Security & Privacy Magazine, Sep 2004.