A better security paradigm: authority-oriented design

Julien Couvreur

December, 2004

Abstract

Security today focuses mostly on preventing malevolent code from getting run on the system. But thinking we can keep the malware out is a fallacy.
The prevailing security paradigm, based on principals[1] and access control, is the reason our systems are so difficult to secure. The paradigm itself is fundamentally flawed, leading to easily corruptible systems. As a result, using a firewall, patching code, and running anti-virus software offer little defense against the old security truism - "once you run code, it's game over".

This paper introduces a new model that could provide significantly better protection against malevolent code: authority-oriented design. It elegantly merges the security principle of least authority with the object-oriented design philosophy. It can apply throughout the system, from the user's desktop level down to the object level, and extends to distributed systems.
By providing each application and component only the authority it needs, the attack surface is reduced. Individual components may still be vulnerable to attacks, but not much damage can be achieved anymore. Finally, the new model allows engineers to focus security improvement efforts on components with the most privileges instead of dealing with every piece of the code.

Authority-oriented design could give us the strategic advantage needed to win the malware arms race, with the potential to offer a future of virus safe computing.

1 Motivation and disclaimer

This paper reports on existing research. The information was gathered through research papers, presentations, talks and mailing lists. The main contributors

[1] principal: the entity in a computer system to wich authorizations are granted; thus the unit of accountability in a computer system.

are the Virus-Safe Computing Initiative at Hewlett Packard Laboratories and the open source secure distributed language E (available at `http://erights.org`). Besides the E language implementation, other working systems are available: EROS, the Extremely Reliable Operating System, and CapDesk, the capability secure desktop that includes a secure browser (DarpaBrowser) and supports instant messaging and distributed file management. Norman Hardy, Marc Stiegler, Mark Miller, Alan H. Karp and Ka-Ping Yee are some of the many contributors, whose ideas I'm trying to summarize here.

I've been reading about this approach for over a year and became convinced that it is a very promising way that can make our computer future significantly more secure. This concept not only would have helped render the worst viruses to date almost ineffective, it also makes writing new viruses much harder, by laying a solid security foundation. Unfortunately these ideas are still spreading rather slowly, hence my attempt here to advocate them.

Authority-oriented design is not the proper name for this security model (instead called the "capabability security model"), but I think it provides a better image for the concept, that seems a bit obscure at first.

Thanks to Lina (linat) and Erren (errenl) for the numerous discussions, helpful feedback and overall patience.

2 The security model is the problem

Malicious software, or "malware", is any software developed for the purpose of doing harm to a computer system: viruses, worms, trojan and spyware. There has been a dramatic rise of these programs in 2004 (over 50% growth in the number of known viruses[2]) and they are increasingly used by financially driven criminal activities[3].

Many blame bad administration (configuration and patching) and software engineering (common security bugs, lack of security training or code reviews) for the damage that we let malware do. But in fact we may have brought this upon ourselves, as we're relying on a security model that constantly gives out excessive authority and that ubiquitous excessive authority is a guarantee that it will be abused.

2.1 Principals and ACLs

> Microsoft's 1st law of security: "If a bad guy can persuade you to run his program on your computer, it's not your computer anymore."

[2] `http://news.bbc.co.uk/1/hi/technology/4105007.stm`
[3] `http://news.zdnet.co.uk/internet/security/0,39020375,39180203,00.htm`

The security model used in Windows, Unix and variants like the NSA's Security-Enhanced Linux relies on the concept of principals. A principal is the computer representation of a user, real or virtual, for example redmond\billg, machine\administrator or machine\system.
All programs are run in the security context of a principal, or possibly more than one via programmatic impersonation. When a program requests access to a resource, such as the file "filename", the system looks up the resource for that name and checks its access control lists (ACLs) to determine if the principal (and thus the program) should be granted access.

The first problem in this scheme is that the large number of system objects and possible permissions (read, write, modify, create, ...) makes managing the ACLs a painful and error-prone process.
This was partially mitigated by adding new concepts and complexity in the system to make it easier to manage larger scale systems: groups, roles or policies.

The second problem is that any program the user runs implicitly gets all of the user's authority, instead of just the authority needed to do the job.

To illustrate this with the following scenario: a user using notepad. The ACLs on the system are set so that the user can do what he needs to do: edit his documents and send mails and access the web.
When notepad is started, it is implicitly getting all of the users authority, which means it now has access to the all user's documents, it's mails and the network. Notepad is not programmed to abuse any of these resources. But if anything was to go wrong, like opening a file exploiting a notepad vulnerability, notepad could do a lot of damage, for example sending the user's documents over the internet and erasing them from the hard drive.

The more power the user needs, the more dangerous notepad becomes, not to mention the program that you downloaded from the internet. Because even the simplest program is such a security risk, it also becomes a more interesting target. Therefore, security efforts are spread across a very large attack surface, trying to secure each line of code in each program.

Many mitigations attempt to define what is safe to run and prevent the "untrustworthy" programs to reach the system or get run by the user. Also, it is possible for security-conscious developpers that need some "system" privileges for their application (ie. IIS) to adapt their design for more security. But it usually comes back to building complex work-arounds, involving multiple principals and switching between them at various times in the process.

2.2 Good vs. bad security measures

2.2.1 Negative examples

Security mitigations are located at the edge are more brittle in nature: once a barrier is breached, it is as good as gone, instead of degrading gracefully.

Holes are found before fixes are written, patches deployed and virus-scanners updated. Firewalls don't protect from social engineering attacks or other means of reaching inside the protection boundary. Users make mistakes and download trojan-carrying programs, opening attachements and connecting to websites that have some exploit payloads.

All of these methods require knowledge about what is malware (in the form of network traffic, file signatures or user education/intuition) which takes time and effort to keep up-to-date. The delay to effectiveness, the complexity, the performance and the maintenance costs are clear limitations in those systems.

Authenticating code isn't much of a solution either. Gator and C-dilla sign their spywares and crackers sign some ActiveX exploit payloads with legitimate Verisign certificates. Some unauthentic "Microsoft Corporation" certificates have been issued on occasion. And although NGSCB/Palladium does offer some protection, by storing the keys more safely, it is still vulnerable to exploits and the remaining data can still be attacked by conventional methods.

Code Access Security (in .Net) is a variation on the ACL system and code signing. It is a more elaborate way to set permission configuration and policies. In essence, permission is granted based on the source of the code (internet zone), any signature that it may carry and the policy settings for the user. This model has poor usability, as defining these policies is as difficult as fine tuning ACLs (if not harder) and users get presented with incomprehensible error prompts when the program is denied a permission that it needs.

2.2.2 Positive examples

Many technological changes that had positive security effects involved changing the inner design of the system. Process isolation protects applications from interfering by running them in different virtual memory spaces. Managed code relies on automatic memory management to prevent a number of classes of bugs, including the infamous buffer overflow. Other similar improvements include the NX (No eXecute) memory flags, stack overflow detection using canaries (StackGuard) or having a separate stack for return addresses. These architecture investments have provided more scalable advantages against malware.

But are still insufficient. Even though managed code is more secure against buffer overflows, some security bugs can be found in the runtime and the applications themselves. Not to mention running "untrusted" code getting run. When any of these incidents occurs, all the user's data is potentially compromised and his account and machine may be exploited as a zombie for other mischieves.

Filtering technologies such as firewalls or virus scanners are still useful. But they won't always prevent the malware from running on the system, one way or another. To achieve better "defense in depth", systems should be designed to remain mostly secure even when that occurs.

3 Principle of least authority

The Principle Of Least Authority (POLA) can be stated as "the authority for a component should be reduced to the minimum possible".

This applies at many different granularity levels: the user, the application, the object within an application or the computer in a distributed system. Authority is the sum of direct permission (what a component is permitted to do) and indirect access (getting another component to do it). A component should only get the authority it needs to perform its tasks. For example, notepad should not be given general access to the network or to most of your files. Notepad should not be able to get access to these indirectly (by means of another program) either.

Most applications don't need that much authority. If they only had the minimum required authority when they got compromised, the damage would be very limited.
The applications that do need a wider array of authorities should be handled by applying POLA at a deeper level. In the case of an email client, that would mean compartmentalizing the authority by separating the inbox module and the email rendering module from the network module, so the damage is isolated to the authority of the compromised component. If a malware needs more authority, it needs to compromise more modules independently. An analogy is the comparion of a tyrant to committee: it is much easier to corrupt the tyrant, which holds all of the authority, than multiple members of the committee, which each hold partial authority.

Confining authority within component boundaries breaks down the unit of compromise in smaller chunks than simply the "user's account". And reducing the scope of authority of each component also shrinks the attack surface. Security reviews and testing can now concentrate on the parts that hold more authority. The same way code reviews consider the responsibility of each component and their interactions, the scope of security reviews is now reduced in both breadth and depth, to focus on the component's authority and how it shares it:

- the risk of notepad copying your files or replicating a worm over the network can be quickly accounted for as a low risk, since notepad doesn't need any network capabilities,
- the network component in the email reader should only expose the received emails resulting from the network operations, but never give out the network capabilities it holds.

The goal of applying POLA to the system is not to make it impossible to write viruses, but rather to limit the amount of damage that can be done when the exploit does occur and to help identify and minimize the area of vulnerability so the security efforts can focus on the powerful/risky actors.

4 Object-capability model

Applying POLA means getting ride of the ambient and excessive authority. The solution is to have no default or implicit authority and to bundle authority with designation (this combination is called a "capability").

Object oriented programming divides responsibilities amongst objects, reducing the scope of each function and providing a framework for the functions to interact. The object-capability model is an extension of the object model that implements POLA by dividing authority amongst the objects and providing rules that constrains how that authority is accessed and delegated.

An object-capability is an object reference, that confers the authority to use the object, within a system that follows the capability discipline. The capability discipline states that:

- **Authority can only be accessed by capabilities**:
  - an object Alice can only access object Bob (in the form of a method call on Bob or message passing to Bob) if Alice holds a capability to Bob.
  - authority can be derived from the capability graph.
- **Only connectivity begets connectivity**:
  - capabilities can only be acquired through:
    - introduction (Alice decides to share a capability to Carol with Bob, and sends a message to Bob containing a Carol capability),
    - parenthood (Alice creates a new object Carol and thus holds the Carol capability),
    - endowment (Alice holds a Carol capability and creates object Bob including the Carol capability) or
    - initial conditions.
  - the graph defines what new connections can be created (disjoint graphs can never become connected).
- **Absolute encapsulation:**
  - if an object holds a capability, but decide to keep it private, then it should be impossible to steal this capability from that object. But the object can share the capability or expose some interfaces that indirectly give some access to that capabilities' functionality.

As Tyler Closer (Waterken Inc.) puts it elegantly, "Access control is achieved by constructing a graph that simultaneously defines and enforces the access policy"[4].

[4] `http://www.waterken.com/dev/Web/Calculus/`

5 Applying capabilities

5.1 No ambient authority

Authority is not implicitly derived from the principal that the application is running as. By default, an application or component has no authority, unless it is provided some. Delegation of authority is made explicit: authority to an object is passed along with the designation of that object, instead of having the permissions be attached to the object itself (ACLs). In short: "if you shouldn't use it, you just can't see it".

For example, "solitaire" would need to be started only with the authority to open a window. "notepad" would also need either a direct capability to a file or a capability to the file chooser dialog of your desktop (that would return a capability on the selected file).

This solves the "confused deputy problem"[5]: when a program is tricked into using some authority for a wrong purpose. For example, when a program has some authority to write to some audit files, for self-tracing purpose, and is tricked into outputting its result file over the audit files, thus erasing the audit trail. Instead, with the capability security model, because the user doesn't have authority to the audit files, he cannot specify these audit files as the output parameter and have the program overwrite them.
Another example would be cross-site scripting attacks[6]: when the user opens a malevolent email, the browser is tricked to load some email or website administration page within the security context of the user (authentication cookies) that will erase the user's website or emails.

5.2 No global name spaces

Object oriented programming already discourages the use of global variables and favors breaking down a program's scope into object. Similarly, capability discipline requires that any global mutable variables be forbidden and that scope of authority be reduced.

The file system can also be viewed as a global name space. In principal-based systems, the naming and the permission are separate. But in a capability system, files can no longer be accessed by their name from the code, but only through another file or directory capability.
This means that the static File.Open() method in the .Net framework would need to be replaced by an instance method aFile.Open() or aDirectory.OpenFile(). Since "solitaire" doesn't have any file or directory capability, there is no way that it could access the filesystem.

[5] `http://c2.com/cgi/wiki?ConfusedDeputyProblem`
[6] `http://lists.canonical.org/pipermail/kragen-tol/2000-August/000619.html`

The shell and the command line need to be adapted to handle object-capabilities natively (the same way the Monad shell can handle objects).

5.3 Capabilities cannot be forged

Capabilities cannot be forged, they can only be aquired as described in the capability discipline. For example, capabilities are not pointers, so no arithmetic is allowed (by the language) or possible (the objects are allocated at random locations in the memory, or memory is divided into protected segments). Also, capabilities is a special type, that is handled specially by the system, so they cannot be leaked as bits over a communication channel.

Encapsulation is enforced so that private members really are private. This isolation is important to control the scope of authority that components can get, by isolating them from each other. Absolute encapsulation let's you write wrapper objects that securely limit authority, which is a very important pattern of safe cooperation between untrusted components.

5.4 Capabilities and engineering

Numerous parallels can be drawn between the principles of object orientation and the capability discipline.

Object-oriented design breaks down the scope of responsibility and makes the graph of interaction the essence of the design. Similarly, capability-based design breaks down the scope of authority and by bundling authority with designation, it makes security an integral part of the design.

Sharing responsibility between objects can be guided by patterns of interaction. Similarly sharing of authority between objects and designing the capability graph can be guided by common patterns of safe cooperation (restriction, revocation, audit, ...).

Conventional design and security practices advocate defense-in-depth. Applying POLA to the system at various granularity levels creates massive defense-in-depth. That said, capabilities don't dispense us from the other traditional good engineering practices (validating inputs, handling errors,...).

5.5 Mutual trust under suspicion

The capability security model allows for fractal definition of authority: the design can be broken down into more details, with more different security compartiments, if needed. It allows for trust, but with moderation: a user doesn't give all of his authority when running code, by default. This allows for truly useful and safe mobile code.

This security model doesn't need to try and define what is trustworthy. Instead, design and patterns for safe cooperation help programmers build secure software with components that safely co-operate even if they don't fully trust each other. "Satan comes to Dinner in E"[7] is an example of such a design. It models the dining philosophers problem in a way that Satan hidding amongst the philosophers couldn't cause a deadlock (preventing other participants from eating).

5.6 Distributed systems

In the E language, the concept of capability is extended transparently beyond the boundary of a single machine. Distributed capabilities follow the same discipline as local ones: you should not be able to guess a capability. Cryptography is used to achieve this (remote capabilities seem similar to certificates, that give you access to a certain service), as well as encrypting all communications.

E also introduces the concept of promise, related to the concept of capability. It is a reference to a yet uncomputed return value from a capability call (or message passing). Promises are used to optimize distributed computing against network latency, to avoid roundtrips, blocking calls and deadlocks.

6 Usability

The traditional approach to security has been a trade-off between power, security and usability. But existing systems mostly only achieve two of these three goals. For example, Java applets are pretty secure and usable, but powerless (they can't save any files). Sandboxes have a different trade-off, where power and security are achieved, but usability lacks (the user is interrupted by security popups).

In capability-based systems, the bundling of designation and authority may help get all three aspects right at the same time. Instead of choosing a file and then getting a security popup (as in the case of sandboxes), the act of picking a file also explicitly gives authority to access it. The need for separate security popups is overcome by integrating the security aspects into the common system dialogs. Some new usability solutions will have to be explored. Instead of simply choosing a file, the file chooser now requests for read access or write access to a file.

One of the difficulties will be to find the right level of granularity for the division of authority. At the extreme, it is possible to give a program very fine-grained authority such as the ability to open a GUI window or access a certain slice of the CPU time, but so much detail would be detrimental to the user's ability to control the system.

[7] `http://www.erights.org/e/satan/index.html`

Ka-Ping Yee has documented guidelines for secure UI design[8]. Here are his ten suggested principles for secure interaction design include:

- Path of Least Resistance. The most natural way to do any task should also be the most secure way.
- Appropriate Boundaries. The interface should expose, and the system should enforce, distinctions between objects and between actions along boundaries that matter to the user.
- Explicit Authorization. A user's authorities must only be provided to other actors as a result of an explicit user action that is understood to imply granting.
- Visibility. The interface should allow the user to easily review any active actors and authority relationships that would affect security-relevant decisions.
- Revocability. The interface should allow the user to easily revoke authorities that the user has granted, wherever revocation is possible.
- Expected Ability. The interface must not give the user the impression that it is possible to do something that cannot actually be done.
- Trusted Path. The interface must provide an unspoofable and faithful communication channel between the user and any entity trusted to manipulate authorities on the user's behalf.
- Identifiability. The interface should enforce that distinct objects and distinct actions have unspoofably identifiable and distinguishable representations.
- Expressiveness. The interface should provide enough expressive power (a) to describe a safe security policy without undue difficulty; and (b) to allow users to express security policies in terms that fit their goals.
- Clarity. The effect of any security-relevant action must be clearly apparent to the user before the action is taken.

His research paper illustrates how these principles are violated in todays user interfaces, as well as positive design suggestions for fixing these issues.

Much of the capability graph needs to be laid during setup, with the user relying on the default configuration for the most part and not having to review all the authority that each process gets. But there is still some usability challenges for allowing users to safely install programs.

[8] `http://www.sims.berkeley.edu/~ping/sid/`

Marc Stiegler has come up with 6 simple rules that can guide users through a secure desktop experience (using CapDesk, the capability-secure desktop), Grandma's rules of POLA[9]:

1. If an application, during installation, proposes for itself a name or an icon that looks a lot like the name or the icon of something else, give it a new name and new icon.
2. If an application asks for a bunch of different authorities, just say no.
3. If an application asks for read or edit authority on anything outside the Desktop folder, just say no.
4. If an applicatin asks for edit authority on a bunch of stuff, just say no.
5. If an application asks for wide-ranging access to the Web, rather than access to one or two specific sites,, only say yes if you plan to use the application as a Web browser.
6. If an application asks for read authority on a bunch of stuff, and also asks for a connection to the Web, just say no.

7 Various thoughts on capability-based security

- This model doesn't solve all problems. It doesn't get ride of the authentication problem, it doesn't stop spam (although making the computing platform safer does help machines becoming zombies as part of bot nets) and it doesn't implement DRM.
- It doesn't need need new hardware, although it is possible that it could benefit from some. For example, some encapsulation (keeping private capabilities private) can be enforced thru software, but some hardware isolation (similar to virtual memory mapping) might make it more robust or efficient.
- The main advantages of the capability model compared to the ACL model, is that it doesn't rely on ambient authority and allows for true least-privilege (instead of giving out excessive authority). A deeper analysis of the abilities of this model compared to ACLs is available in "Capability Myths Demolished"[10].
- Going beyond identity? By removing the notion of principal, the notion of user is also blurred. A user is just an entry point to a network of capabilities. This model allows for a very natural integration of sharing activities,

[9] `http://www.skyhunter.com/marcs/granmaRulesPola.html`
[10] `http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf`

like collaborative file ownership. For example, owners and contributors of a file are concept that no longer need to be hard-coded into the system, instead they can be emulated with capabilities (a more powerful capability is . Capabilities are more powerful and flexible than ACLs. Applying concept to the web and distributed environments.

7.1 Various benefits

Fractal security approach: system level (users and various system authorities like files, network,...), single-user/desktop level (applications and various user authorities like files, network...), single-application level (modules, components or objects and the various module authorities).

Quick general security principles: isolation, simplicity, default to secure.

Confinement

8 References

- E programming language (wikipedia) `http://en.wikipedia.org/wiki/E_programming_language`
- E language homepage `http://erights.org`
- Features of CapDesk and demo links: http://www.skyhunter.com/marcs/CapDeskSpec.html
- Erights.org: Home of the E programming language
- Combex: Capability secure solutions based on E.
- EROS-OS: The open source capability secure operating system
- Waterken: Capability based security for the WWW
- Caplore: Home of numerous items of wisdom from the fountainhead of modern capabilities, Norm Hardy

9 Buffer

Polaris: Virus Safe Computing for Windows XP -> not capabilities, but uses the principal paradigm to apply POLA: each application runs under a different user. http://www.hpl.hp.com/techreports/2004/HPL-2004-221.html

http://ricardo.ecn.wfu.edu/~cottrell/wp.html

perimeter-based defense is porous

Recognizing the porous nature of perimeter defense does not mean that border security should not be improved or that additional steps to prevent illegal immigration should not be taken, however, not recognizing its porous nature is unrealistic, counter to current trends in security practice,

Giving too much authority to even a "trustworthy" program is foolish.

Migration strategy:

*Sandboxes/runtimes/virtual machines

* E is built on top of Java

http://www.erights.org/talks/skynet/

The SkyNet Virus (Why it is Unstoppable; How to Stop it)