

# On the Management of Object Interrelationships

Martin Uhl, Werner Held  
*Boltzmannstraße 3, Institut für Informatik  
Technische Universität München  
85748 Garching b. München, Germany*

## Abstract

*Still one of the main problems in computing security is the scope malicious intruders can gain by introducing their own thread of control. To make this worse, coarse grained structures of current operating systems are not designed to contain such breaches. So essentially the task of maintaining security lies not only with the operating system, but also within the attacked process.*

*Obviously however, a conventional process is not at all equipped properly to defend itself nor is this task within its scope of responsibility.*

*In introducing a new concept called nano protection domains backed by the technique of so called one-step capabilities we strive to give the operating system a much finer grained structure which will make it much harder for a security breach to actually do harm.*

## Keywords

operating system structures, objects, capabilities, security, intrusion containment

## 1. Introduction

Within the last ten years, the computing landscape has changed drastically. While in former times most computers were mainly used as standalone workstations, internet-worked (if at all) only inhouse.

Nowadays lots of computers are connected to each other via the worldwide internet. Today, millions of nodes are connected, be it the network of a multinational corporation, or the PC of your mother.

When connected to the Internet, a computer becomes an equal and integrated participant of it and may therefore accept connections from and send messages to the myriad of the rest of the connected computers. This makes any node a potential target for manual and automated attacks coming from nearly everywhere.

The increasing number of nodes has made it more and more “rewarding” for malicious subjects to break into systems. The reasons for this are manifold - may it be personal pleasure, to let out steam or even monetary gain. Especially rededicating cracked computers and then selling access to them as so called “Botnets” is a new and most insidious practice. It has already found its application in the ever growing nuisance of e-Mail spam, which use these rededicated computers as anonymous spam relays. We are now at a state, where breaking into computers has thus even become commercially interesting. Even this is just the tip of the iceberg.

At the core of most intrusions into today’s computer systems is a technique called “buffer overflow” which exploits the lack of security checks inside an operating system process, especially the check for array boundaries. Purpose of such a “buffer overflow” attack is to introduce own code into a process and to get it subsequently executed. Once achieved, the attacker takes custody of the thread of control. Effectively the original program has now been replaced. By now exploiting this entry point into the system, an attacker can now take any appropriate measures which he sees fit.

Unfortunately no operating system employed today is prepared for this kind of attack - not even being aware of it - and will run the now malicious process happily alongside its other ones. Unfortunately UNIX-structured systems and programming paradigms inherently lack the security features necessary. Note that nearly all operating systems fall under this kind of category, even those not directly associated with UNIX, e.g. Windows.

What makes matters worse, is that the scope an intruder can rule over - after he has successfully introduced his code into the process - is the *whole process* and *anything the process is able to access*.

The high growth and adoption rate of the Internet encourages attackers and leads to a growing number of attacks, amplified by commercial interest (see above), in fact boosting the problem further more.

## 2. Operating System Design

Alas, something which has not changed these past years is the composition of the employed operating systems. At this time, most operating systems follow the UNIX Principle [4].

### 2.1. Protection

Which security measures do operating systems offer today? The current approach is to separate kernel functionality from user functionality and furthermore separate user functionality into different processes. To attain that purpose, operating systems are partitioned horizontally and vertically.

The horizontal partition holds the so called user space - kernel space boundary apart which separates operating system management tasks from conventional user calculations.

This boundary can only be crossed by issuing special instructions, so called *system calls* which are implemented using a special CPU instruction called `trap` and a redirector table. This partition is backed by hardware protection which only allows the set of kernel mode related instructions to be issued while actually in kernel mode. Also it is assured that a switch into kernel mode can only happen through the `trap` instruction.

This hardware protection is quite solid, since the flow of control is stopped at the `trap` instruction and may only continue at predefined locations within the kernel, specified by the system call table. Of course this table is protected by hardware and not accessible from user space.

The vertical partitioning of the operating system holds the process - process boundary apart. It is only applied to the user space, which means that there is only one kernel space but multiple user spaces. This boundary is realized through the application of different personal virtual address spaces which are assigned to the processes in a one to one relationship.

Through the usage of different and usually disjoint address spaces, no process can reference memory of another one since the employed address space is part of the process' context and may only be switched by the kernel. The memory translation mechanism is protected by a hardware memory management unit. More so, all the relevant operating system memory management functions reside in the kernel space and thus profit from the kernel mode protection.

This process to process protection is quite effective, even so effective, that it makes it hard for processes to communicate among themselves. It has served well over the past decades and is therefore the dominant method of protection inside today's operating systems.

### 2.2. Evaluation

Since operating systems are partitioned in such coarse grains, a potential intruder is hindered only by a few security barriers, namely the horizontal and vertical partitioning. However, in most of the cases he might actually be content by what he got in breaking just one process.

A process puts his entire protection domain at risk by connecting to untrusted communication partners. So the dominant method of attaining security is to check and possibly forbid these connections. To achieve this, computers and networks can be protected by firewalls. In fact nearly all current operating systems come bundled with a firewall today.<sup>1</sup> But even firewalls are too coarsely grained, since most of them only operate on the connection level and therefore cannot separate between an attack or a harmless connection, they only block on rather simple rules. Although there are firewalls acting on a higher level, these are complicated to maintain and require a lot of knowledge about the protection domain, which possibly has to be reengineered.

Security maintained by an external component has its downsides. Imagine a castle with well defended walls. A spy slipping by these walls undetected (possibly by digging a tunnel or disguising, etc...) could wreak havoc as he pleases (or is told to) if there are no sentries inside the castle.

To put this metaphor into the IT context, some kind of analogy to sentries, i.e. security checks, must be put into the operating system. Unfortunately the current coarsely grained structure makes it disproportionately hard to insert security checks into operating systems anywhere besides the protection boundaries.

A well known technique for breaking into a computer system [3] is even named after its application in history: The greeks using the *trojan horse* to get past the well defended walls of Troy, which they ultimately managed to conquer.

Now if putting sentries inside our today's operating systems would be so easy. According to their structure (see 2.1) sentries could only be positioned at the protection boundaries.

### 2.3. Countermeasures

Unfortunately in the last past years, the situation did not improve. In fact, in the last years, the complexity of software and operating systems did rise to a level where it is even difficult to handle normal operation correctly, let alone improve the security handling. Some software companies even did increase the complexity of their products

<sup>1</sup> Although it is debatable whether a firewall residing on the computer to be protected is a good method to increase security, since once the computer has been breached, the firewall might be accessible to the intruder.

intentionally to make it harder for their competition (see [2]). This situation makes it even easier for intruders to find weak points and subsequently exploit them. Additionally, should such a weak point be inside the kernel level, an intruder would gain complete and utter control over the attacked computer.

Furthermore, increased complexity makes the removal of security deficiencies a difficult task. Introducing new deficiencies while removing one is often the case.

From our point of view, it is imperative that the system itself gains a finer grained structure to contain security breaches more effectively. While we think that 100% security is only attainable under very high cost (if at all), it should be made clear that reducing the scope of damage should be a high priority. It is feasible to increase it through clever structural considerations.

On the following pages we therefore propose our concept of the nano protection domains which is thought to replace conventional operating system structuring in order to give it a much finer grain in which possible breaches can be more securely contained.

### 3. Structure

Like argued in [5] the current basic building blocks of operating systems have to be rethought. As mentioned in this paper, objects offer themselves to replace the structuring element of process local private virtual address spaces as new basic building blocks for a new operating structure.

Our goal in restructuring is to give an intruder a space as small as possible. Therefore the classic process related private virtual address spaces must be broken down further.

Similar to the concept of small protection domains[3], nano protection domains limit the scope of action to only a small and necessary set of objects. Nano protection domains limit this scope even further than small protection domains in more strictly controlling the environment a thread of execution runs in.

The scope of reference a small protection domain has are the capabilities it got when it was created. Thus also limiting the initial scope an intruder could get by breaking into the thread of execution. But a cleverly designed attacker could adapt and try to collect capabilities. Since it is hard to revoke a capability (one of the weaknesses of capabilities), this type of attack is obviously feasible. Nevertheless it requires a lot more expense on the side of the intruder.

Now, part of this deficiency comes from the fact, that capabilities can be exchanged freely, so any object could get into possession of one, even if it is not meant to.

### 3.1. Design

For unification purposes, in our approach *everything is an object*. Using only objects as operating system basic building blocks [5] adds a finer grained structure. As part of this restructuring new possibilities for access checks are enabled by regulating the access to objects and their interfaces.

**Object Types.** The objects that are used, are grouped into two types:

**primary object** is directly implemented in hardware and cannot be further split into more specific objects. It is atomic. Primary objects directly represent values stored in memory or the state of any other hardware. These objects always have the implicit methods read and write, whereas write changes the state of the object, and read does not. Usually the basic types like `int`, `float`, `bool` etc. are primary objects. Also any other hardware device is thus represented by primary objects and has to have the read and write methods which qualify an object state change.

**aggregate object** consists of other objects. They correspond to the classic notion of objects used in programming languages like Objective-C, C++ or Java. Thus, aggregate objects consist of other aggregate objects and/or primary objects.

Using only objects enables the operating system to qualify each memory reference as an object reference, thus unifying the way the user interacts (via the operating system) with the raw hardware.

The horizontal and vertical boundaries of current operating systems, which are now implemented by kernel mode and private virtual address spaces are being replaced by only using objects and their attributes. When looking at the execution tree, the process notion is therefore implemented by associating a branch of this tree to a task. A side effect of this protection is, that there is no direct read/write access to executable code in memory from within an object.

**Object Relations.** An important part in creating a nano protection domain is to qualify the different types of object relationships. This leads to an object classification based on their scope of reference.

**local object** This object is local to the current thread of execution. It usually lives as long as its thread. It must not be referenced elsewhere. Local objects can be classified further:

**auxiliary object** Object is strictly used as an auxiliary variable and therefore is created and destroyed within the thread of execution of its containing method.

**turn over object** Object is instantiated for the purpose to hold return values. This object can leave its thread of execution and enter another one. It may only be possessed by one method at a time.

**instance object** Object belongs to the instance of its containing object and may be referenced from all the methods of this object. It may not be communicated outside of its instance and is therefore only viewable from inside the methods of its object.

**base object** Object, which has a global scope of visibility. However access to such an object has to be requested. Also it may be possible for a base object to be made persistent.

Based on this object relationship, one-step capabilities ensure the enforcement of these rules.

### 3.2. Nano Protection Domains

Since the primary goal is to make nano protection domains as small as possible, we have to establish clear rules of which objects may reference what and when. Also it is of importance how to regulate the hand-over of the thread of control, since through this action it travels between nano protection domains.

The entry point of each nano protection domain is a method call. By entering a nano protection domain, a new scope of reference is opened, which consists of the following:

- user capabilities to the local instance methods
- user capabilities to *turn-over objects* that got handed over
- creator capabilities to objects that got copied for hand-over
- creator capabilities for any object that got instantiated in the local context
- user capabilities to base objects that got requested and granted

### 3.3. Rules of Access and Hand-over

The normal way of calling another method is call-by-value<sup>1</sup>. This confines any overlap of nano protection domains to the minimum possible, facilitating distribution of nano protection domains, keeping actual interdependencies small between domains, and especially facilitating error recovery through possible rollback

<sup>1</sup> Since memory is not that scarce and expensive any more, you can now copy data when you need it. Internally it very well may be managed by copy-on-write

Turn over objects may be passed by reference between two methods. This means, that the sender must have a creator capability, and the receiver will get a user capability.

When an object instance is created, the creating context gets a *creator capability* for this object, which it may pass on as a parameter or return value.

Access to base objects is handled differently. Since base objects are - as a matter of principle - visible from everywhere. Access to them has to be specially qualified. For this purpose a name and permission server has to be set up. This server holds creator capabilities to all base objects and manages the corresponding user capabilities. Upon successful request, the server hands over a user capability for the base object. Consequently, access to base objects is always restricted to the method which requested access to it. It also may be possible for this thread of execution to relinquish control of the base object before its termination by releasing the user capability to the base object, further diminishing the scope an intruder could get.

### 3.4. Intrusion Detection

A possible method of intrusion detection would be to stop the thread of execution in the case of a permission exception. This permission exception is then treated specially, i.e. it must not be caught in the local context, but either in the calling context of the current method or in a special execution handler. These then resume the execution and may then handle this permission exception. Thus preventing an intruder from just checking out some permissions.

### 3.5. Communicating Base Objects

To communicate the usage of a base object the system may either use call-by-name or a special type of pointer, called a weak pointer, which only contains a hint to the base object, but no permissions to use it, like a capability which only contains the pointer, but no access rights. Using a base object as parameter automatically converts this strong pointer into a weak pointer upon hand-over.

### 3.6. Lifetime

The discrete structure of the objects and their relationship allows some statements about object lifetime.

**local object** Obviously, since this object can only be referenced within its local scope of execution, its lifetime is also constricted by it.

**auxiliary object** If it is not released before the end of the thread of execution, it will then be released.

**turn over object** The lifetime of this object is extended to the runtime of the receiving thread of execution

**instance object** Its lifetime is constricted by the lifetime of its owning object. If it hasn't been already, it will be automatically released when its owning object is released.

**base object** This object will only be released, when it is told to. There is no automatic release. Since it is possible for base objects to be referenced by multiple other objects, some kind of reference counting or garbage collection may be set up.

## 4. One-Step-Capabilities

### 4.1. Overview

To achieve the properties of nano protection domains, we devised one-step-capabilities. Using this special type of capability allows the easy realization of nano protection domains. Like traditional capabilities [1], one-step-capabilities act as surrogates for addresses and qualify the type of access made using these addresses. Therefore the technique of replacing pointers with capabilities is often described as capability based addressing.

Capabilities are owned by the caller object and mask the called object to which they are intrinsically tied. Thereby adding plausibility checking to the call, deciding that only qualified objects (i.e. those in possession of the correct capability) may execute the call. It is possible to reduce the access rights of a capability. Traditional ones may also be passed freely around.

Two distinguishing features separating traditional capabilities from one-step-capabilities can be identified:

**object orientation** Traditional capabilities qualified access to memory ranges, whereas one-step capabilities are used to qualify access to objects. Memory only has three distinguished access methods: read, write and execute. Objects have an arbitrary set of methods, by which they are accessed. One-step capabilities accommodate for this difference by extending their access part to accommodate the corresponding object methods of their qualifying object. This set of methods can be reduced to accommodate for the given context.

**rules for capability hand-over** The possibility of traditional capabilities to be freely distributed is a two edged sword. On the one hand it is desirable and necessary to communicate references, on the other hand unrestricted capability distribution may render the whole capability system ineffective. As previously mentioned, a possible attack might be that an object collects capabilities for later misuse. In this context, capabilities act like a key for an object. Anyone owning the key gets access to the object. While there are plausibility checks at a call to an object, there are no

checks for the capability hand-over. Therefore the security through traditional capabilities may be circumvented at another point, namely the capability hand-over.

To ameliorate this deficiency, one-step capabilities introduce rules for capability hand-over by limiting the scope in which a capability is valid.

### 4.2. Functionality

Like traditional capabilities, one-step capabilities qualify access to an object. The former to a memory object, the latter to a real object. While a call is made, the system uses the list of allowed operations on the object contained in the capability to decide whether the call is valid or not. This clearly resembles the procedure already employed by traditional capabilities with the subtle difference that one-step capabilities now qualify object access.

One-step capabilities also handle capability hand-over. For this purpose, they are distinguished between two types:

- creator-capabilities
- user-capabilities

The main intention of these two types is that a capability may only be handed over one step (hence the name one-step capabilities). This means that the receiver of a capability, which resides in another context must not pass it on again. Context in this reference means the actual thread of execution inside - and only inside - the current method.

This is, where the two capability types come in. The creator-capability is created with each new object instantiation; it is thereafter owned by the creator of its associated object. This creator may now pass on the capability to another object. When arriving at its call target, the capability has then automatically been converted to a user-capability. The owner of a user capability is not able to pass it on into another context. The user-capability must only be used for calling its object.

In some cases it is necessary to relinquish control of the creator capability, e.g. to convert an object into a base object by giving its creator capability to a capability server. If a creator capability has to be passed on, its owner automatically releases control of it and the receiver becomes the new owner of the creator capability.

Recapitulating:

- Only owners of creator-capabilities can pass on capabilities.
- User-capabilities are only valid inside their context.
- Upon passing on, a creator-capability is converted into a user capability.

- There must be only one creator capability on an object, while there may be multiple user capabilities.
- If a creator capability must be passed over, ownership of the creator capability is also passed over.

### 4.3. Composition

A traditional capability consists of a memory pointer and a list of allowed operations on the memory object. One-step capabilities have some additional attributes, respectively a type identifier and an owner. To recapitulate, one-step capabilities consist of

- Object identifier
- List of allowed operations on object
- Type identifier
- Owner

The *object identifier* points to the object protected through the capability. For the whole capability system to work, it must not be possible to access an object without going through a capability. The system must make sure that this rule is never breached.

The *list of allowed operations on object* contains the identifiers of the methods this capability allows access to. Only a call which is approved through the list is declared valid and executed.

The *type identifier* specifies whether the capability is of the *owner* or *user* type. According to the type, the capability may be passed on to another context or not.

The *owner* is the object in whose context the capability resides. It is mainly important for denoting the creator object. It may only be changed in a creator capability.

### 4.4. Effect

The functionality of one-step capabilities directly represents a principle of locality. This principle of locality is also employed by the nano protection domains, which are the conceptual counterpart to the one-step capability. By applying this technology, we support object usage in a local domain. Such it is made difficult to take an object out of its context, therefore supporting more coherent objects.

Coherent objects in this sense mean that objects have a more clearly defined connection to their containing and contained objects. Through the one-step principle these connections become close, encouraging the use of local objects. The capability aspect gives the operating system the possibility to recognize these dependencies and use them for protection. By keeping objects close together, we attain a more treelike object structure. In contrast, today's object systems act like a vast pool of objects with may freely point to each other.

## 5. Conclusion

This document concerned itself with the issue to introduce a new mechanism to improve operating system security. As a well defined and intended side effect this brought a new basic structuring concept with it.

The standard approach to increase security in a system today is to add something new to it which then should accomplish additional checks and verifications. As much as these solutions strive to accommodate the general case, they often can cover only a special one. So a lot of these add-ons are made and appended to the system. As a consequence, systems tend to get a lot more complex[2] and unmanageable. And this even does not accommodate that different add-ons may not work correctly with each other.

Conversely, we argued that to really improve security, operating system basics have to be rethought. We introduced nano protection domains, which help to partition the operating system into small protected units. As a consequence intruders now have a lot more barriers to overcome if they want to gain complete (or at least sufficient) control over a system. Nano protection domains use the locality of reference principle and expand on it by limiting access primarily on local objects. Further access has to be explicitly requested. We then introduced one-step capabilities which support this locality principle by allowing only the creator of a capability to hand it over to potential users of the accompanying object, i.e. the object creator receives a creator capability for it, which it may pass to object users. In passing this capability, it is reduced to a user capability that may only be used, but may not be passed on, respectively may not leave its thread of control. All access to objects outside of the nano protection domain has to be specially requested.

Since attaining 100% security is attached to such high costs (for now - or even perhaps in the future) we argue, that it is not in our current scope of reach. For that, our goal should be to improve operating system security as much as possible. With nano protection domains we have come closer to this goal by reducing the scope of damage an intruder can inflict. We also opened a possibility for intrusion detection through the separation of permission exception generation and handling.

## 6. Future Work

The dynamics of one-step capabilities and nano protection domains have to be further researched. Especially in the field of passing on capabilities.

It should be examined if it is feasible to create a special object, which may be passed on and further on, i.e. the creator loses its creator (and also user) capability when passing it on also no user capabilities must be made from this special object. So enabling the capability to closely follow the object through the different threads of control.

Modeling concurrency is also an issue which will require further attention.

## References

- [1] D.Gollmann. *Computer Security*, volume 2. Wiley Press, 2004.
- [2] Dan Geer, Rebecca Bace, Peter Gutmann, Perry Metzger, Charles P. Pfleeger, John S. Quarterman, and Bruce Schneier. *Cyberinsecurity*.
- [3] Theodore A. Linden. Operating system structures to support security and reliable software. *ACM Comput. Surv.*, 8(4):409–445, December 1976.
- [4] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, 1974.
- [5] Martin Uhl. On operating system basic building blocks. In *Proceedings of the 2005 international Conference on computer design CDES'05*, volume 1, pages 175–181. CSREA Press, 2005.