A Practical UNIX Capability System

Adam Langley <agl@imperialviolet.org>

22nd June 2005

Abstract

This report seeks to document the development of a capability security system based on a Linux kernel and to follow through the implications of such a system.

After defining terms, several other capability systems are discussed and found to be excellent, but to have too high a barrier to entry. This motivates the development of the above system.

The capability system decomposes traditionally monolithic applications into a number of communicating actors, each of which is a separate process. Actors may only communicate using the capabilities given to them and so the impact of a vulnerability in a given actor can be reasoned about.

This design pattern is demonstrated to be advantageous in terms of security, comprehensibility and modularity and with an acceptable performance penality.

From this, following through a few of the further avenues which present themselves is the two hours traffic of our stage.

Acknowledgments

I would like to thank my supervisor, Dr Kelly, for all the time he has put into cajoling and persuading me that the rest of the world might have a trick or two worth learning.

Also, I'd like to thank Bryce Wilcox-O'Hearn for introducing me to capabilities many years ago.

Contents

1 Introduction **1**

2 Terms **3**

2.1 POSIX ‘Capabilities’ 3

2.2 Password Capabilities 4

3 Motivations **7**

3.1 Ambient Authority 7

3.2 Confused Deputy 8

3.3 Pervasive Testing 8

3.4 Clear Auditing of Vulnerabilities 9

3.5 Easy Configurability 9

3.5.1 Filter 9

3.5.2 Revoker 10

4 Existing Systems **11**

4.1 The Burroughs B5000 11

4.2 Dennis and Van Horn 12

4.3 Hydra 13

5 Contemporary Designs **15**

5.1 EROS 15

5.2 E 15

5.3 Coyotos 16

5.4 Asbestos 16

5.5 Others 16

6 Design **17**

6.1 Why a new design? . . . 17

6.2 Basic Design . . . 17

6.3 Representing Capabilities in Linux . . . 18

6.4 Master Model . . . 20

6.4.1 Distributing Capability Systems . . . 21

6.5 Connecting Processes Together . . . 21

6.6 Limitations Imposed . . . 23

7 Capability based web-server **25**

7.1 A more complex design . . . 27

8 On the Importance of Users **29**

9 Writing Capability Aware Programs **33**

9.1 Concurrency Orientated Programming . . . 34

9.1.1 Error Handling . . . 35

9.1.2 Flow Control . . . 36

9.2 Related Work . . . 38

9.3 Implementation Details . . . 39

9.4 Where we went wrong . . . 39

9.5 Parsers for Network Protocols . . . 40

10 Conclusions **45**

10.1 Reflections & Further Work . . . 46

A Specification **49**

A.1 LSP Wire Protocol . . . 49

A.2 Master Interaction Protocol . . . 49

A.3 Master Protocol . . . 50

A.4 Interhost Communications . . . 50

A.5 Manifest Format . . . 51

Chapter 1

Introduction

"Don't forget the quotations at the beginning of each chapter"
– Etienne Pollard

When you go to the corner shop, do you hand the cashier your wallet, and ask him to take out what it costs?

No? Then why can your screen-saver open your private key and email it across the world?

The security model on modern UNIX systems makes it possible to restrict such programs, but not easily. Efforts such as SELinux[LS01] have been used to try to secure some distributions of Linux[Cor05]. However the SELinux configuration files for a typical setup now run to some 250,000 lines[lwn04] in a language which very few people understand.

This report presents a vision for a security model based around capabilities and documents a prototype implementation that demonstrates both its benefits and feasibility.

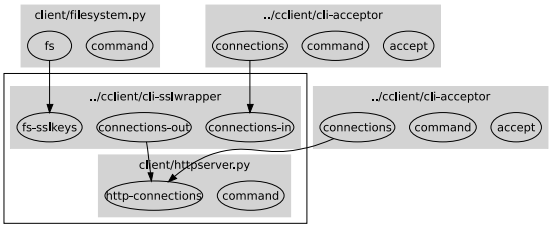
- We develop an object capability model within the framework of a POSIX kernel.
- We develop implementations in Python and C++ to demonstrate feasibility and practicality.
- We propose and implement a design for a web server which is significantly more modular, reusable and has scope to be more secure than popular designs.
- We show how the modularity of this design allows the web server to be transparently distributed across computers for load balancing.
- We show that such a design can be clearly audited and simply specified which we suggest are important aspects of any security system.

This report proceeds in three stages. Chapters two through five define terms, explain motivations and review previous work.

Next, chapters six through eight present the high-level design of this report and argue that most of the achievements listed above can be brought about by this design. Importantly we claim that this can be a gradual process and needs no clean start.

The last section concerns itself with the implementation of software which fits into our framework and we argue that this requires fundamentally more concurrent programs. We review our solution to this problem.

By the end we understand how to write and combine processes such that we can reason about the security properties of individual components and automatically extract diagrams like the following for auditing:



Chapter 2

Terms

In this chapter we will tackle the first task of this report; which is to define the term capability. The term has has been given so many meanings that it is almost worth inventing a new one. Hopefully the different meanings and their distinctions will become clear.

A capability is an opaque channel which names an object and authorises its holder, an agent, to use it.

The meaning and implications of the word *opaque* are important. In this definition it means that, at least:

- it cannot be forged
- it cannot be serialised or printed
- it cannot be copied, except via explicit copy methods

As an approximation to hold in mind, one can think of a capability as an object reference in a language such as Java. Object references in Java are opaque as one cannot cast from, say, integers to a reference.

Also, they both name and authorise the use of that object. An object reference is necessary and sufficient, no other access checks are required. (Not strictly true in Java unfortunately). Other object references may be passed down the capability by giving them as arguments to a method call.

We will now discuss several other systems that have used the term capability for very different entities so that the distinctions may become clear by example.

2.1 POSIX 'Capabilities'

POSIX defines an object called a capability in an attempt to fragment the omnipotence of the root user on UNIX systems. The powers of the root account are split into an enumerated set of '*capabilities*' where each capability authorises the use of a fragment of root's authority. For example, from the `linux/capability.h` file:

```
/* Overrides the restriction that the real or
   effective user ID of a process sending a signal
   must match the real or effective user ID of the
```

```
   process receiving the signal. */

#define CAP_KILL 5
```

If a process holds CAP_KILL it may perform an action usually reserved for root. Thus CAP_KILL authorises actions against a number of objects, but it would not be a capability under our definition as it does not name the objects. A separate namespace (in this case, the process's PID) is needed to identify the object which is to be acted upon.

2.2 Password Capabilities

Password capabilities were implemented in the Amoeba operating system[TA90]. Quoting from the introduction paper:

> When an object is created, the server doing the creation constructs a 128-bit value called a capability and returns it to the caller. Subsequent operations on the object require the user to send its capability to the server to both specify the object and prove that the user has permission to manipulate the object. Capabilities are protected cryptographically to prevent tampering. All objects in the entire system are named and protected using this one simple, transparent scheme.

Clearly, in this scheme a capability does both name and authorise the use of a resource, but the capabilities are not opaque. An Amoeba capability is only 128-bits of information and may be transfered over any information channel.

Our definition require that capabilities may only be transfered over existing capabilities, thus the graph of capabilities may become more highly connected over time but, without intervention, disconnected regions of the graph may never become connected.

It may appear that, even in a password capability system, disconnected parts of the graph may not become connected as there is no channel of information flow between them. This is to ignore the practical reality that we are dealing with processes running on a real computer and that there are a multitude of side-channels available to cooperating processes. Indeed, according to [KS74] these side channels exist and we are unable to remove them.

To summarise the above we can consider the various combinations of the properties listed below and identify an example of each combination to highlight how our definition differs from many others (see figure 2.1)

Confusion between password capabilities and true capabilities leads Boebert to mistakenly conclude[Boe84] that capabilities are weaker than ACL based systems. This misunderstanding has, unfortunately, survived in the literature[Gon89].

Boebert considers a two-level Multics-like security system where a privileged process, Bob, can read both Secret and Top Secret data, but may only write Top Secret data. Thus information from the Top Secret area may not leak to the merely Secret one. This is enforced by means of capabilities, thus Bob has READ capabilities for both areas, but only a WRITE capability for the Top Secret one.

Also running is another processes, Alice, which has a lesser privilege and may only read and write to the Secret area. Boebert imagines that Alice (who holds a WRITE capability to the Secret area) could serialise

Names	Y	N	Y	N	Y	N	Y
Authorises	N	Y	Y	N	N	Y	Y
Opaque	N	N	N	Y	Y	Y	Y
	File name	Password	Password capability	?	?	UID	Capability

Figure 2.1: Summary of different properties of security objects

this capability and write it to the Secret area where Bob could read it and use it to leak information to Alice by writing using his new capability.

Boebert asserted that this showed that capabilities are weaker than ACL systems as they cannot enforce security in this system, which he called the *-Property.

Of course this assumes that the capabilities can be serialised, which violates our definition of capabilities as objects which are opaque. Under our definition of capabilities the *-Property is indeed enforced since no capability carrying channel exists between Alice and Bob.

From the other direction we could well imagine an ACL system where the UID of a process could be serialised and written to the Secret area such that other processes could read it and assume the identity of that user. Under this similar misconception it could be shown that ACLs, also, cannot enforce the *-Property.

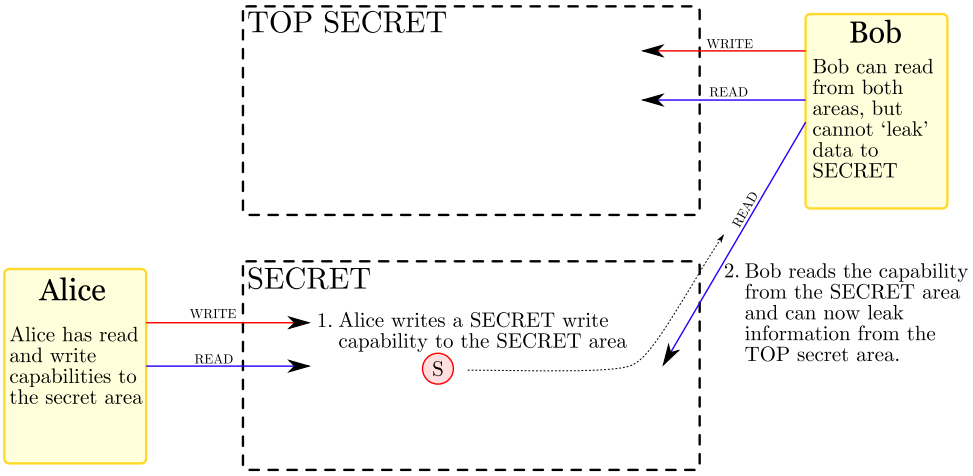


Figure 2.2: Boebert's 'proof' that capabilities are weaker than ACLs

In this chapter we have defined our use of the term capability and shown how it differs from several other uses which have appeared in the literature. We have also shown how at least one misunderstanding about capabilities has arisen from confusing these different definitions.

From this point onwards we do not consider any other uses of the term. The next chapter expands upon some of the reasons why a capability based system is desirable.

Chapter 3

Motivations for a Capability System

At the moment the state of computer security falls far short of what we would wish and this chapter outlines how capabilities could improve the situation in several ways.

The CVE group[Gro] seeks to give standard names to reported security problems so that groups involved can communicate more effectively. A quick search of the list of security problems cataloged by CVE during 2003 (CVEs and CANs, less rejects and reserved entries) shows 322 entries. Microsoft suggests[VT04] that security problems cost businesses about $55 billion in 2003.

The conclusion from this can only be secure software is very difficult (and therefore costly) to build. Given sufficient resources it may be possible to produce highly reliable software[nas05] but this may push the cost of producing software above the utility advantage of using it. Therefore, by economic and historic arguments, we suggest that buggy and exploitable software must be accepted and planned for.

No number of software methodologies, toolkits, languages, libraries nor IDEs have produced the promised salvation from security problems.

Since programs cannot be trusted to perform correctly we must investigate how to control their interaction in order to limit the damage that an exploited component can cause. We must eliminate complexity where we are vulnerable in order to make our security explicit and obvious.

We now discuss several common patterns of vulnerability consider how capabilities can help solve them.

3.1 Ambient Authority

Ambient authority exists when security domains are larger than they need to be. The term *domain* isn't strictly defined but, in the case of UNIX, domains are isomorphic to user accounts.

The drive to eliminate ambient authority is called the *principle of least authority* which states that an object should have all the authority it requires, but no more. This is an obvious statement yet current systems violate it routinely.

As an example, the text editor that this report is typed with (`vim`) is perfectly able to open `.gnupg/secring.gpg` and email it to someone. Since `vim` is running under a user account, `vim` has all the authority that that user has, which includes the ability to create outbound, port 25 TCP connections (email) and to open the users' secret key-ring.

The requirements of the principle of least authority are stated in [MK] as: "(1) split applications into smaller protection domains, or 'compartments'; (2) assign exactly the right privileges to each compartment; (3) engineer communication channels between the compartments; (4) ensure that, save for intended communication, the compartments remain isolated from one another; and (5) make it easy for themselves, and others, to perform a security audit."

Common security models do not allow a user to create limited security domains which can be used to restrict the access granted to processes. The only commonly provided mechanism for this is a `chroot` jail but these can only be created by the superuser and require that many libraries and system-wide configuration files be duplicated for each jail.

Ambient authority amplifies the impact of any exploited component by failing to contain it.

Further to software being flawed, people are flawed too. If all programs have total (from the scope of the user) authority, a simple misconfiguration can lead to any amount of damage up to the maximum that the user is capable of.

In a capability system authority is explicit, not implicit. Creating sufficiently small authority domains involves selectively passing on any subset of capabilities to a child process. This authority, by virtue of being explicit, is also obvious - allowing us to consider the impact of a security failure.

3.2 Confused Deputy

The Confused Deputy[Har88] is a pattern which arises where there is a separation between names and authority.

The `sendmail` process on some UNIX systems (we would hope that `sendmail` would have been replaced by now, but it still hangs on) is responsible for delivering email to users. Thus it has three sets of authority - one from the user which invoked it, one as a `setuid` root binary and one from the user to whom the email is being delivered. The `sendmail` process is expected to juggle each of these and only use the correct authority when performing certain operations.

For example; the user to whom the email is being delivered can setup a `.forward` file which may contain commands to be executed. Obviously, this file must be interpreted with the correct authority otherwise a user could execute any process as `root`.

However, as expected, `sendmail` has got it wrong many times in the past[Ber04]. Amongst numerous bugs, `sendmail` has managed to get all three authorities confused at one time or another.

In a capability system there is no possibility of having multiple authorities at once. If a process has two different levels of access to the same object, then it has two separate capabilities to that object. It is still possible for a sufficiently poor program to misuse the capabilities that it has been given, but this possibility is greatly reduced by avoiding the need to to switch authority correctly using calls from the `setuid` family.

Even in the case of a bug, capabilities allow for least authority so that the impact of the bug is less than the total system compromise that most bugs in `sendmail` allow.

3.3 Pervasive Testing

Unit testing has gained wide acceptance in recent times but is often frustrated by the inability to isolate components to be tested. Although data structures are easy to test (since they have no external effects)

few people are sufficiently motivated to structure their code so that more troublesome areas can be tested (such as network interacting code).

Although component isolation in a capability system is motivated for security reasons it serves excellently for testing purposes too as it requires a system to be broken down into message-passing actors. Copy-everything message passing is a gift for test writers as a test framework can easily be dropped around a component without needing to restructure the component.

3.4 Clear Auditing of Vulnerabilities

The flip side to explicitly granting authority is the ability to introspect a system to determine which processes currently have any given authority. We would wish to be able to do this for every capability in the system and we demonstrate in chapter eight the ability to export the graph of capability connections as a graphic which we believe provides excellent introspection.

3.5 Easy Configurability

We believe that a general framework for plugging together modular systems leads to a more intuitively obvious method for access control than many dissimilar, disintegrated configuration files.

Several patterns occur in developing capability systems which are worth detailing at this point. This may answer some questions that the inquiring reader may already be asking and demonstrates how modular systems can be constructed.

3.5.1 Filter

The filter pattern can be used when a capability needs to be restricted. See figure 3.1.

Figure 3.1: The capability filter pattern

In specific cases it may be possible to request that a resource create a new capability with less access than the capability that was used to request it. This may not be possible either because the exact level of access cannot be specified to the resource, or because the resource doesn't provide a means to make new capabilities.

The filter pattern involves a proxy object which presents the same interface as the resource that it is proxying. Certain messages may be ignored or rewritten by the proxy to enforce any restrictions whatsoever.

Since a capability names, as well as authorises, the use of a resource the child process is unaware that its requests are being proxied and so no special support is needed in the client. In this way a filter is slightly

different to a proxy pattern in that it's highly polymorphic and a single filter can be reused to filter any process - regardless of the interface (so long as the configuration is sufficiently flexible).

3.5.2 Revoker

The revoker pattern is similar to a filter in that both involve proxying all requests through an object which presents the same interface as the protected object.

The difference is that a revoker proxy does not alter the messages passing through. It waits for a signal from a parent process (or whomever the parent process hands the capability to) and destroys itself.

Since the capability to the protected object is only held by the proxy object this has the effect of revoking it.

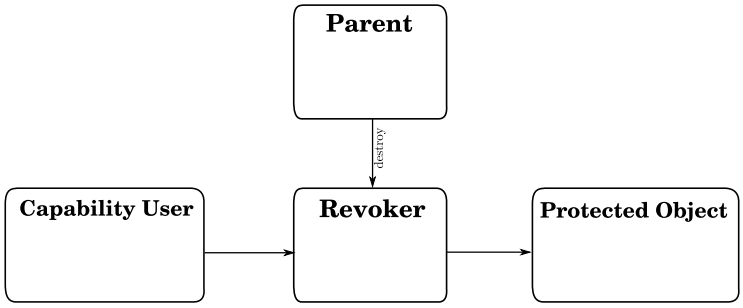


Figure 3.2: The capability revoker pattern

This chapter has suggested why capabilities may be a better tool for shaping security than those which are commonly used. The next chapter reviews some of the landmark works which advanced the state of the art of capabilities.

Chapter 4

A Review of Previous Capability Systems

In this chapter we describe a number of previous capability systems. In doing so we hope to inform later discussion about the merits of different approaches.

When describing them we will be updating the terminology to a more modern style. For example; older papers often use the term *meta-instruction* where as, now, we would use the term *system call*.

4.1 The Burroughs B5000

The motivation for the early capability work came from the, then novel, problem of protecting different programs on a single computer.

Early work in this field produced the Burroughs B5000[Bur61] computer which was the first to use segmented memory and here we can see the beginning of capabilities.

In order to allow the operating system to manage memory effectively, programs addressed memory with a *segment* and an *offset* in that segment. The segment was, itself, an index (called a descriptor) into a per-process structure called the Program Reference Table (PRT) which acted much as a segment table acts in modern processors. Thus the PRT determined the complete set of memory which the process could address and so limited its ability to interfere with other processes. The extra level of indirection also allowed the operating system to reorder segments (to avoid memory fragmentation) and to swap them to secondary storage.

PRT entries could also be program descriptors. These caused a remote procedure call to the described code segment when indexed. The index in this case served as an argument and the called procedure could use the caller's PRT.

In this system, the PRT indexes were the capabilities. They are distinguished objects (or rather pointers to distinguished objects, as a process cannot directly manipulate its PRT) and they name and authorise access to the object (segment) which they protect.

Although the B5000 (and later, the B5500) was the first to introduce many features, its capability support was the most revolutionary:

From [May82]

> "The descriptor was one of the most novel features of the B5000.... Indeed, Burroughs published a description of the B5000 system and titled it 'The Descriptor' "

(information for this section was obtained from secondary sources [May82][Lev84])

4.2 Dennis and Van Horn

Dennis and Van Horn first described[Jac83] a modern capability system in 1966. Although they were defining a *design*, not describing a working computer, most of the facets of a modern capability system can be found in their seminal paper.

Dennis and Van Horn defined many terms, including *capability* and described it for the first time:

> "We think of a computation as proceeding within some *sphere of protection* specified by a *list of capabilities* or *C-list* for short. Each capability in a C-list locates, by means of a pointer, some computing object and indicates the actions that the computation may perform with respect to that object."

A partial list of types of objects protected by capabilities in the paper:

- Segments
- I/O devices
- RPC entry points
- Protection spheres (C-lists)
- Suspended processes
- User consoles

Each process has a C-list that lists all the capabilities which that process can use. Like UNIX file descriptors, system calls are given an index into this C-list to specify the capability to be invoked.

They also define what is now called a thread as a process which shares its C-list (and thus its sphere of protection) with other processes.

Dennis's capabilities have four permissions bits which are interpreted according to the type of object that the capability references. The permissions bits are **R** (read), **W** (write), **X** (execute) and **O**. The **O** bit indicates that the holder of the capability owns the referenced object and this allows it to perform operations such as deleting it.

Capabilities, in Dennis's system, may be selectively granted to child processes. The CREATE SPHERE system call returns a C-list which may be populated by GRANT calls before being passed to START, which creates a new process with that C-list. The GRANT call can copy any capability which the parent holds into the new C-list with any lesser set of permissions bits.

The START call returns a capability to the child processes with the **O** bit set. This allows the parent to start, stop and manipulate the C-list of the child.

Capabilities also protect RPC calls in Dennis's system. The CREATE ENTRY system call returns an RPC capability which may be passed to other processes in order for them to make RPC calls. During the call both the caller and the callee are protected from each other - neither can use the C-lists of the other. Communication is via a number of capabilities which can be passed as arguments to the call. (Recall that memory segments are protected by capabilities in this system, so the passed argument can be a pointer to a memory buffer and this will work across processes).

Dennis's system also has a single level store. Segments which were still referenced by a process could be moved to secondary storage and the kernel was expected to garbage collect these segments. Thus there was no filesystem in the modern sense, there were just long-term objects which were still referenced by capabilities.

In order to manage this Dennis and Van Horn described a naming system, but this is extraneous to a discussion of capability systems.

Dennis and Van Horn's paper laid out many of the ideas for capabilities and these ideas were implemented, to varying extents, in several systems. Notably the PDP-1[AP67], the Chicago Magic Number Machine[Fab67][She68] and the CAL-TSS system[LS76].

4.3 Hydra

The next significant step for capability systems was the development of the Hydra system at Carnegie-Mellon.

Hydra was developed as a system for experimentation with two central features: it was microkernel based to allow experimentation in areas which were usually the domain of the kernel, and it pushed the idea of object orientation into the kernel.

Thus, in Hydra, objects were first-class kernel entities and not just within the domain of the language in which a given process was written. The kernel concerned itself with the creation of new types, with virtual call dispatch and with the management of the references to these objects (which were, of course, capabilities).

Objects in Hydra have four parts:

Object ID This is a 64-bit number which is unique for the entire lifetime of the system.

Type This is the 64-bit ID of the type object which describes this object. For type objects themselves this is a sentinel value.

Data The object may maintain state in a number of bytes of memory

C-list The object may also hold capabilities to other objects

Unlike modern object systems there was no concept of public and private data (member variables), the level of access to the object which you had was dictated by the permissions bits in the capability for it that you held.

The type objects are responsible for creating new instances of objects of their type (and thus for allocating the data and C-list areas) and they hold method capabilities for the methods which may be called on the object. Type objects hold a special capability called a *type capability* which allowed (amongst other things) the creation of new objects of that type.

When a method call is made the activation record for the call is kept in an object called the *Local Name Space* (LNS). This object holds the capabilities which the method has access to. (These capabilities may have been arguments, or may be stored in the procedure object). Since this is an object system, the caller is protected from the callee and vice versa; that is, neither have access to the other's C-list.

A problem now presents itself. In order to maintain encapsulation, the capabilities for a given type of object which are given out to the world should not allow the holder to modify/inspect the data or C-list of that object. However, when a method call is made, the method needs to be able to do these things even though the caller is unable to pass a sufficiently powerful capability as the *this* argument.

This is resolved by *amplification templates*. These are a special type of capability, which can only be created by the holder of a type capability, and are inserted into the C-list of a procedure capability. They allow a capability argument of the correct type to have its permissions amplified. The amplified capability lives in the LNS of the method call and it thus destroyed (generally) upon return.

The permissions in a capability are more complex than is generally found in such systems. These permissions are designed to be able to implement *const* objects and to prevent the called method from leaking information which may be contained in the object. These permissions are described below:

*x*DataRts	The ability to **Get**, **Put** or **Append** to the data part of the object.
*x*CapRts	Likewise for the C-list of the object
DeleteRts	The ability to delete the *capability* (not the object) from a C-list
KillRts	The ability to delete capabilities from the C-list of the referenced object, providing that those capabilities have **DeleteRts**
ModifyRts	The ability to modify any part of the object. *Cannot be gained by amplification*
EnvRts	The ability to copy the capability from a LNS object. *Cannot be gained by amplification*
UncfRts	The ability for a method to maintain state. *Cannot be gained by amplification*

The three permission bits which cannot be gained by amplification deserve special attention.

The MODIFYRTS bits is used to implement a *const* object. When it is cleared the method may not modify the object even if it has PUTDATARTS etc.

ENVRTS allows the caller to prevent a capability from leaking out of the method. When this bit is cleared on any capability argument the method may not copy that capability out of the LNS object. Thus it must loose it upon return. A capability without ENVRTS may be passed to another method, but that method will likewise find that it cannot save it outside its LNS.

UNCFRTS forces the method to be functional. When a method is called on an capability which lacks this permission bit all capabilities in its LNS have UNCFRTS and MODIFYRTS removed. Thus the procedure cannot modify any external objects, except its LNS. Thus the method must be side-effect free.

Hydra, like the Dennis & Van Horn system, implemented a single level store. Objects could be stored on disk and reloaded when referenced. Garbage collection deleted objects which could no longer be reached.

(information for this section was obtained from secondary sources [Lev84])

This chapter has reviewed some older designs which used capabilities at a fundamental level. The features of these systems are used to evaluate our design in chapter six (page 23).

The next chapter reviews current work in the field with a view to explaining why the work presented in this report is needed.

Chapter 5

Contemporary Designs

Although the use of capabilities has declined in modern operating systems from the levels shown in the previous chapter, several current efforts are using them. This chapter will briefly outline contemporary work in the area.

5.1 EROS

EROS[SSF99] is another research operating system, which came out of the University of Pennsylvania. It is a refinement of KeyKOS[Har85] which was a previous work by many of the same authors. Many of the facets of the early capability designs have been disregarded in the past few decades as operating systems such as the POSIX family have risen to dominance. EROS is very much in keeping with those early designs, but updated for modern computer architecture.

Early capability systems introduced the idea of segmentation of memory, which still exists in modern architectures but is largely vestigial compared to its previously central role. Paging is now the ubiquitous model of memory management and EROS reflects the structure of page tables as capabilities. Processes hold a capability to the root of an address space. A tree of capabilities identifies every page in that address space and the EROS microkernel walks this tree in the case of a page fault. Capabilities for pages can be shared between processes to elegantly implement shared memory.

EROS also extends the idea of a single level store to one of orthogonal persistence. The *entire* state of the system is check pointed in a copy-on-write manner at intervals such that, in the event of a power failure, the system can return to the exact state of the last checkpoint - running processes, in-memory state and all. Applications need not concern themselves with serializing their state onto a persistent medium (which is why it is described as *orthogonal* persistence).

Sadly, EROS has had very little take up outside of the group of people developing it. It's difficult to build, has poor hardware support (by modern standards) and application support is practically non-existent.

5.2 E

E[Lan] is not an operating system, but a capability language implemented on the Java Virtual Machine. It concerns itself with the construction of highly concurrent, secure programs and thus is probably closest in scope to this project.

E defines a protocol for capability security over networks and provides a number of libraries designed to make the development of secure network applications as easy as possible in E. Since this protocol is open it's quite possible to implement programs which use E's network security architecture in any language. Like this project, E eschews the idea of concurrent mutable shared state (but that is kept for a later chapter).

More originally, E extends the idea of capability protection down to the level of single objects. By using the features of the JVM (namely the unforgeability of object references) E builds a structure where untrusted code may safely be used within a program. This has required a large scale redesign of most of the Java standard libraries since they were never designed with this level of security in mind but used a weaker sandboxing pattern for containment.

5.3 Coyotos

Coyotos[coy] is the next iteration of EROS and is in very early stages at the moment. It is mentioned here in order to provide a complete set of references to the interested reader who wishes to survey the state of the art.

Coyotos currently includes a number of changes to the foundation design of the EROS kernel which reflect lessons learnt from the development of EROS. Its most radical departure from EROS is that the new kernel is to be built in a language, BitC, developed for formal reasoning while keeping the low-level hardware control of C.

5.4 Asbestos

Asbestos is another project in its very early stages. The introductory paper[MK] was published in the final weeks of writing this report and will be covered in greater detail later as the first half describes a design very similar to that which we have used.

The final section outlines their future plans which are to develop a capability based operating system which includes mandatory access control for additional security. Their current work is very brief but it appears that the MAC elements are a performance optimisation.

5.5 Others

The above is not an exhaustive list and other works exist which have not been included in because we felt that either they were not sufficiently different to those above or they are tangential to this report. These include Plan9[pla], Plash[Sea04], Polaris[MS] and Capdesk[MSMS].

Chapter 6

Design

6.1 Why a new design?

Over the previous two chapters we have outlined many previous works in this area but have left open the question of what we believe that we can do better.

We are developing this project because previous systems have had too high a barrier to entry. All but one of the previous contemporary designs have been operating systems and it is our belief that common PC hardware is now too diverse for a fresh-start operating system to ever gain widespread acceptance. The complexity in writing hardware drivers, which are vital if people are going to be able to even boot an operating system, is now prohibitive for all but the most well funded efforts.

Possibly more importantly, it's vitally important that current applications continue to run without modification and any new work can interact with this existing corpus of programs.

The E project's network capability protocol is closest in design to what we are attempting but was very much designed for a single language (namely E itself). It also fails to address some of the motivations for a capability system which we outlined previously.

This leads to our decision to build a framework in which capability programs can be built in any language and can interact with the rest of the world. Security in this system will be enforced at the operating system level rather than by language-based security.

Thus elegance and beauty in the design will be sacrificed in the name of practicality.

6.2 Basic Design

We constrain our design to one which will work on a Linux kernel with minimal modifications. As a rule of thumb, we should not require any patches which would break any existing applications, or which could never be considered for inclusion into the mainline kernel.

This design will develop in three stages. Firstly, the basic design of this project will be outlined. Then common elements of previous work will be highlighted and contrasted to this design. Finally, a more detailed design will be given.

6.3 Representing Capabilities in Linux

The role of capabilities in our design will be played by UNIX file descriptors, since they meet all of our requirements.

They both name and authorise the use of an object. Once a file descriptor is held no further authority is required. (To convince yourself of this, change the permissions of a file after you have already opened it and try reading.)

They uniquely name an object: A file may exist, and be named by a file descriptor, even when there is no other name for it in the filesystem. Many classes of objects (for example, sockets) can only be named by a file descriptor.

They are opaque: To a user-land process they are small integers. Communicating this integer to another process does not grant that process the ability to use the file descriptor. They may only be exchanged with the co-operation of the kernel.

Finally, a file descriptor may be passed along another file descriptor. This is only true when the file descriptor refers to a UNIX domain socket, and the mechanism for doing this is little known. But it is fully specified by POSIX and supported by all POSIX kernels.

Our domains of protection are UNIX processes. Within this domain, capabilities (file descriptors) are global, but no capabilities from other domains can be used. Different processes are also protected from each other and the vectors of manipulation are explicit, controlled and well known.

Using UNIX processes helps greatly by providing confidence in some of the axioms which will be required when it comes to reasoning about the system. At some level, assumptions about the correct working of certain core parts of the code need to be made and Linux kernel has had more scrutiny than anything we could hope to produce in the scope of this project.

Over UNIX domain sockets, which are our capabilities, we need a form of lightweight RPC. Although we could allow the communications to be free-form, suggesting a standard is a good idea in terms of code reuse and interoperability. It should be noted that nothing enforces our standard protocol and co-operating processes could, if they wish, implement any protocol over capabilities.

Our standard protocol, called LSP, is simple and an existing protocol could have been used except for one very important feature - capability passing. No widespread RPC protocols have support for passing capabilities and the kernel interface for doing this requires that the protocol be tokenisable without backtracking.

LSP messages have a tree structure and it is greatly influenced by BitTorrent's `bencoding` (which is, itself, based on Mojo Nation's `mencoding`). There are few data types:

1. Symbols (8-bit clean strings of, at most, 65K bytes)
2. Dictionaries (the keys of which may only be symbols)
3. Lists
4. 64-bit integers
5. Capabilities

LSP messages are written in this document somewhat like Python structures. For example: `[this is a list {this is a dict} 3454]` would be expressed in Python as `['this', 'is', 'a', 'list', {'this': 'is', 'a': 'dict'}, 3454]`.

Although the above outlines, in brief terms, how a capability system might work, it doesn't stop any capability process from exploiting all the current ambient authority which UNIX processes have.

In order to do this we require a kernel patch to deny marked processes from making many system calls. (How processes become marked will be explained later). The system calls allowed are specified as a white-list. The full list is below:

- `accept`
- `alarm`
- `brk`
- `close`
- `clone`
- `dup`
- `dup2`
- All `epoll` system calls (Linux only)
- `exit`
- `exit_group`
- `fcntl`
- `fork`
- `futex` (Linux only)
- All assorted get UID/GID/PID calls
- `sockopt` calls
- `ioctl`
- `mincore`
- `mprotect`
- `munmap`
- `nice`
- `poll`
- `select`
- All I/O read/write calls
- All thread calls (Linux only)

- All signal calls (except `kill`)
- `socketcall`
- `socketpair`
- All calls to get the current time
- All assorted wait calls

In addition the `open` and `access` system calls are allowed in a restricted sense: write access is not allowed and only certain paths are allowed to be used. The ELF dynamic linker (`ld.so`) uses these in a very fundamental sense, as does the Python interpreter. This is yet another example of elegance being sacrificed for practicality. We would wish that no aspect of the filesystem be directly exposed to marked processes, but we must remain compatible with the rest of our software universe.

System Principles

At this point we outline a few principles of the above design. We do this so that the reader will recognise them when they appear again:

- Processes have "share nothing" semantics.
- Message passing is the only way to pass data between processes.
- Isolation implies that message passing is asynchronous.

6.4 Master Model

In order to communicate with running capability processes we must possess a capability to them. This must be a UNIX domain socket since we might need to pass other capabilities in order for them to function.

Thus we could either have a master process which holds capabilities to all running child processes (for a given user) or we could give each child process a listening domain socket bound to the filesystem which we would connect to when we need a capability for a given process.

We chose the former because it allows for the caching of some process information and because it saves littering the filesystem with domain sockets (which aren't deleted when a process dies). Thus the master process acts as a de-multiplexer for messages to running processes and can be reached by a single, well known, domain socket in the filesystem.

Interested readers are directed to the appendices for full protocol documentation, but the most important master commands are:

`exec` Fork and exec a new client process

`connect` Create a new socket pair, pass one end to the specified client and port and return the other

`biconnect` Create a new socket pair, pass one end to one client and port and the other end to a separate client and port

Those commands account for nearly all the requests to the master. Note that a process issuing these commands has managed to open a socket from the filesystem, suggesting that it is an unmarked process. We call these processes *controller* processes and they are not children of the master process.

6.4.1 Distributing Capability Systems

Since capability processes work on a basis of copy-every message passing they lend themselves to transparent distribution over a network of computers. Capability channels can be sent across network connections to processes running on a different computer and neither side should be any the wiser. However, dealing with passing capabilities in this situation requires a little more work.

Within a single host, capabilities are passed using special kernel calls to transfer file descriptors between processes. This only works within a single kernel. When capability channels are carried over the network any capabilities passed down that channel must be *pinned* at each end. This is done by the master process.

A `client-biconnect` or `client-connect` within the same computer causes the master to create a new socket pair and pass one end of it to each client. When the clients are on different hosts the master creates a socket pair and keeps one end itself. It then proxies all data from that socket over a new TCP connection to the remote host. The remote host is told which client and port the connection should end up with and proxies it to that client.

When a capability is received from a proxied connection a new TCP connection is setup and the new capability is proxied at each end. In the data stream of the first connection, the capability object is replaced by a marker denoting which TCP connection is to carry the data for this new capability (see figure 6.1).

When a `client-biconnect` command specifies two separate remote hosts, data is relayed via the master process. Thus the flow of data is always a star topology - it never flows between remote hosts without going via the master process. (This keeps the design simple and removes the possibility of different parts of the network having conflicting views of which hosts are reachable.)

Separate TCP connections are used for each capability since they all need independent flow control. An alternate transport, SCTP, was considered since it has the ability to multiplex many kernel flow controlled streams down a single connection. SCTP was dismissed since it is poorly supported on Linux systems, the number of streams is limited and the sockets API is incomprehensible.

The use of many TCP connections is troublesome since each requires separate authentication. In this protocol authentication remains unimplemented but future work will need to consider how to avoid rekeying each connection. TCP connections also need to be opened ahead of time and cached in order to remove latency from inter-host communications.

6.5 Connecting Processes Together

Traditional UNIX command line processes have had three channels over which they communicate (`stdin`, `stdout` and `stderr`). This limited model has worked very well in many situations, but fails in several respects when the processes are long lived. Firstly, it's generally very difficult to introspect the

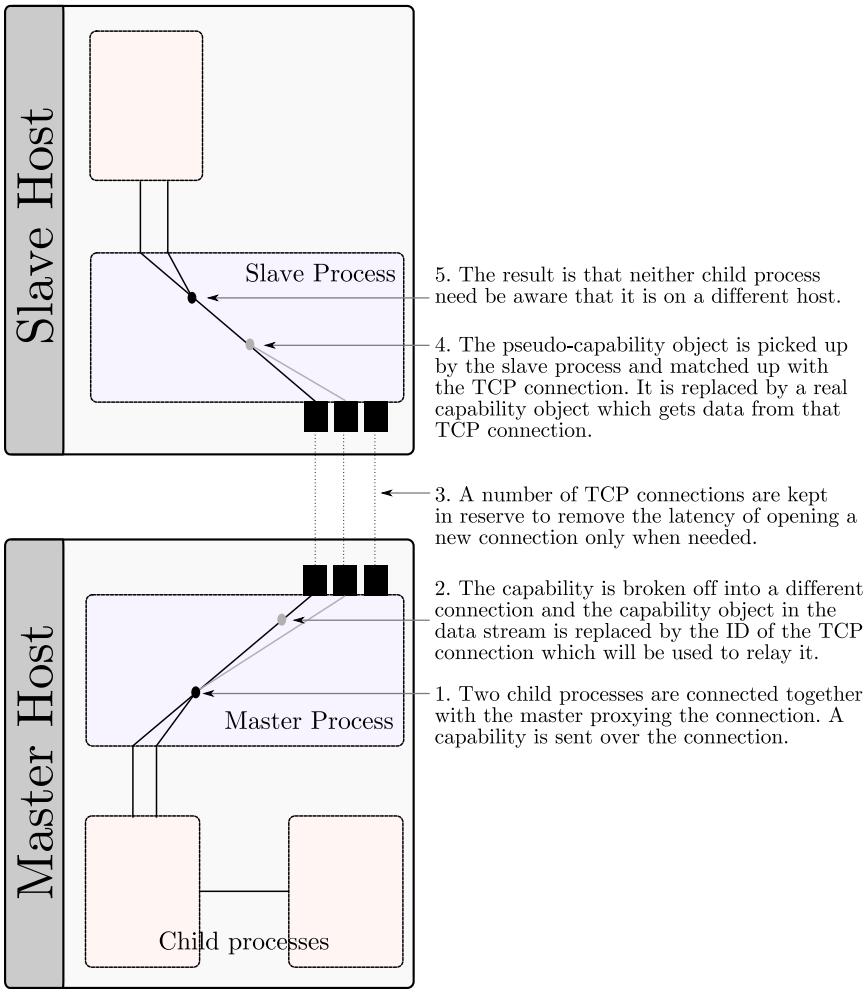


Figure 6.1: Transparent handling of capabilities across hosts

connections between running processes without a lot of digging in `/proc` and manual matching up of socket inode numbers.

Secondly, the number of channels is too small for many tasks, leading to lots of `tee` processes buffering to the filesystem where following tails pipe the data into another process.

Because of this we have designed a simple system whereby capability processes advertise themselves with a number of ports to which capabilities can be attached. An example of this is given in the next chapter which should help to make it clear.

Each port has an optional type (a free-form string) and a nominal direction which are metadata for user tools and not enforced. The master process caches the port data for every child.

To enable introspection of the graph of connections between ports, the master process is able to enumerate the end points of every connection between ports (with the help of another kernel patch). See chapter eight for details.

6.6 Limitations Imposed

In this section we will try to distill some common themes from the previous work described and contrast them with the basic design above.

Uniform Memory and Single Level Store

Many previous capability systems have used capabilities to address memory and have provided a single level store (e.g. no difference between disk and memory, just capability addressed blocks of data). In every respect we fail to deliver this.

Fundamentally, memory management is a core part of any kernel and altering this, in any significant way, would violate our requirement that kernel patches not break existing applications. Also, previous systems had hardware support for this and the vestigial remains of this on the IA32 platform (segment registers) are all but disabled in Linux.

This is not to suggest that we do not consider single level store ideas to be important. The checkpointing ability in EROS[SA02] is very impressive and useful, but we are simply unable to do anything so radical in this project.

Sealing data

Most capability systems have the ability to seal data in a capability. This sealed data is protected and allows a called process to evaluate the rights of the caller.

In a traditional object-orientated language all references to an object are equivalent. If one has a reference to object A, then one has all the access rights that every holder of such a reference has.

With sealed data this isn't true. The most restricted and common aspect of sealed data are the permission bits that are associated with each capability. But many systems allow extra process defined permissions bits and some, like Hydra, allow arbitrary sealed data.

Our design is certainly different to other systems in this respect.

In a traditional system a process serving a capability call has to inspect the permissions bits or sealed data in order to determine how it should behave. In our system each distinct possible caller is presented as a different channel. Thus the callee can maintain any amount of per-caller state without any special data sealed in the capability.

This design also gives us explicit partial ordering of requests and flow control 'for free', by virtue of the channel based communications.

Security by Language

Some systems have sought to achieve security by building on a virtual machine[Lan] and thus have defined a more comfortable environment in which they can achieve distinguished capabilities, not by any kernel mechanism, but by restrictions imposed on the VM.

Indeed, some systems have sought to achieve this by static inspection of VM code to ensure that code doesn't attempt to violate the security framework[GEK+02]. This allows 'untrusted' code to be run in the same address space and eliminates the overhead of task switching.

We have rejected this path in the name of interoperability.

As designed above, a capability marked process, in our system, may be implemented in any language available on Linux (that is to say, nearly all of them). Although some languages are clearly superior, it seems remarkably difficult to reach agreement on which ones they are.

This chapter has communicated the bulk of the design decisions involved in this project. We are now able to discuss some of the ramifications of this, which we do in the next chapter.

Chapter 7

Capability based web-server

Rather than give a shopping list of the more subjective parts of the design we will work from an example and reach obvious design choices where we can. Remarkably all these obvious design choices are exactly what we have implemented.

Our motivating example will be that of a web server. Web servers are both commonplace and an example of a monolithic design where a modular one would serve better.

At a very macro level we may consider the basic event loop. It accepts TCP connections, on one or more sockets, then talks either HTTP or HTTPS. After parsing the HTTP header it determines how the request is to be handled - generally either by returning a file or passing the request off to some special handler (like a PHP interpreter) which are often linked in. All of which is determined by a complex configuration file for which a special parser etc is required.

Initial designs for a capability adjusted version of a web server may be little different. It would need a capability to accept connections from and some method to read named files. But this would forsake most of the advantages of a capability system. By increasing the amount of modularity we also increase code-reuse, confinement and comprehensibility.

These are the processes that our example web server will consist of:

Acceptor This process will hold a capability from which it can accept connections. (In reality this will, in fact, be a bound IPv4 socket, since this will also appear as a file descriptor and thus a capability to the process.) It will pass these connections onwards.

SSL Wrapper This will accept raw capabilities (raw capabilities aren't expected to use LSP) wrap them in SSL, and produce corresponding raw capabilities.

HTTP Parser This will accept raw capabilities and parse an HTTP request. It can be configured, using string matching on the requested path, on how to pass the request onwards.

Demo Processor This will accept parsed HTTP requests from the HTTP parser and generate some demo webpages to pass back.

In the above many possible details have been omitted. For example, many servers need to be able to blacklist/whitelist hosts on an IP address basis. Generally this is reimplemented for every program which

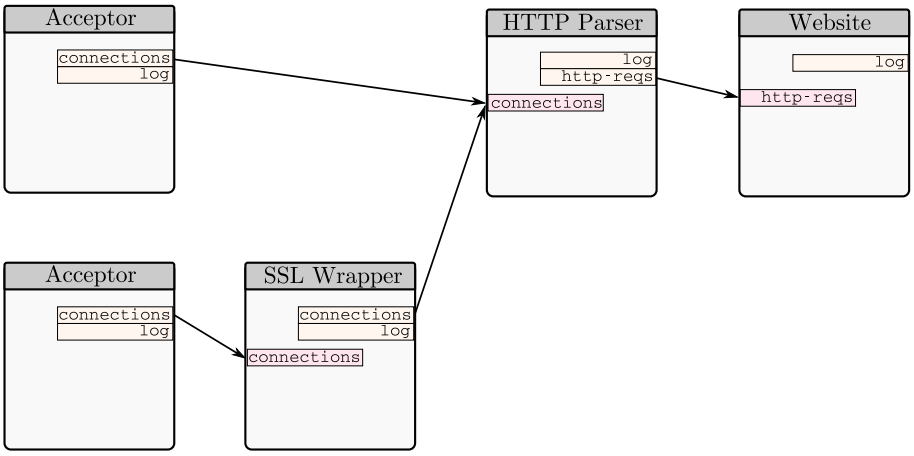


Figure 7.1: One possible capability deconstruction of an HTTP server

needs to accept a TCP connection. We could implement that into a general Acceptor process once, and have any server that we choose to plug together benefit from it.

We can also reason about the impact of a compromise in each process. In the above design the most sensitive part is probably the SSLWrapper. SSL libraries are very complex and implemented in C for performance. Despite much peer review, the common `openssl` library suffered several security problems in 2004.

Assuming that an attacker managed a complete compromise of the SSLWrapper they could:

- Fake connections
- Snoop HTTPS connections before encryption
- Mount denial of service attacks on the host and server
- Steal the SSL key

Of all of these, we consider last one is the most damaging if it could be performed covertly.

It's also instructive to consider what an attack could *not* do, which would be possible in a monolithic server:

- Start a backdoor shell
- Use the host as a staging post to attack the network from the inside
- Use the host as a DDoS zombie
- Walk around the filesystem, reading or destroying sensitive information

If the possibility of loosing the host SSL key is still too much of a risk we could consider splitting all signing functions into a separate oracle process which holds the key, but does not parse complex ASN.1

data from the outside world. With a limited interface, the correctness of the code becomes much more obvious.

A detailed cryptographic investigation into the merits of this approach are out of scope for this project, but its gratifying that such a design could even be considered without changing the web server at all.

A recurring issue when designing modular systems is of the flexibility of interfaces. Common programs have very limited interfaces which cause impedance mismatches when they are used for anything other than that which they were intended.

To illustrate, consider an SSL proxy which sits in front of a normal web server. It's perfectly possible to write a proxy which accepts connections on port 80 and forwards them (after SSL) to another port on the local machine where a web server could listen. In this way the web server can do SSL without any explicit support.

However, when the web server logs events all the IP addresses will appear to be `127.0.0.1` and the web server cannot know which connections are SSL wrapped and which are not, so directives like Apache's `SSLRequireSSL` cannot be handled.

This is a consequence of the very simple interface (a listening socket) which is used to connect the two programs. In the example above both the Acceptor and SSLWrapper output messages in the `connections` format. This format is just an informal protocol for anything which requires connections.

Connections from an Acceptor look like this:

```
[connect <Cap> {from: "192.168.0.1", type: "inet"}]
```

And connections from a SSLWrapper look like this:

```
[connect <Cap>
{from: "192.168.0.1", ssl_protocol: "TLS 1.0", ssl_kx_algorithm: "DHE
RSA", ssl_mac_algorithm: "SHA", ssl_compression: "NULL",
ssl_cipher: "AES 256 CBC", "type": "ssl"}]
```

Note that the original IP address isn't lost. SSLWrapper knows that it's just wrapping connections and so should pass on the IP of the original connection. However, it also adds many extra headers which an SSL aware process can check. Non-SSL aware processes simply ignore the extra headers.

7.1 A more complex design

A more complex design for a distributed web server is shown in figure 7.2. This illustrates several features.

Firstly, a filter pattern has been used after the Acceptors to monitor possible attacks. Although this is fairly standard practice these days it should be noted that this change requires no modification of any other component in the system.

Secondly, transparent distribution has been used to split the work load across many different hosts (the dashed lines show a host boundary). The exact decomposition of course depends on the workload of any

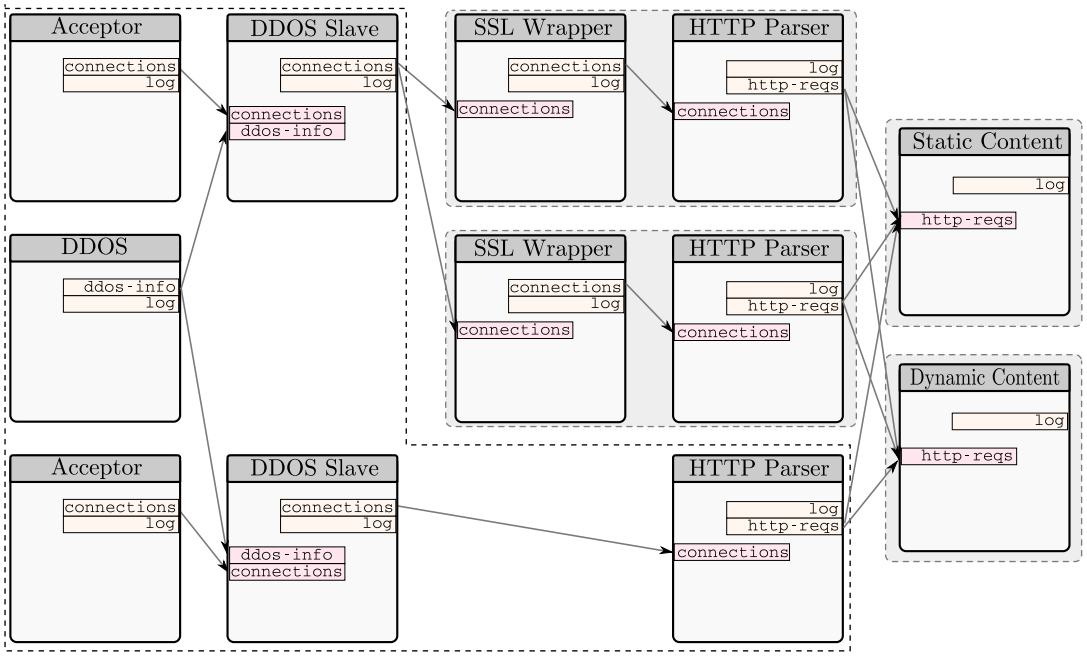


Figure 7.2: A more complex capability deconstruction of an HTTP server

specific site, but in this example two hosts have been dedicated to handling public key and symmetric cryptography which might be sensible for a site which handles a lot of SSL traffic.

It's also worth pointing at at this point that designing the correct building blocks is a very complex matter. In figure 7.2 the top most DDOS Slave has connections to two different objects. If this were implemented with the libraries developed for this project that object would round-robin the connections between those objects.

The SSL protocol has the ability to quickly resume sessions with clients which have connected recently so long as the server still has the correct context cached. However, if the connections were distributed in a round-robin fashion about half of the connections which could have been resumed will be routed to the wrong host.

The solution to this is either to share the context cache between the SSL Wrappers or to have the connections distributed in a manner related to the source IP address. Neither is too taxing but the greater the code reuse the more this issues will appear. It's certainly possible to imagine situations where the the problem isn't so easily remedied.

This chapter has hopefully shown that capability designs can be modular and reasoned about. It has demonstrated how reusable components can combined into systems more complicated than a shell pipeline.

It has also spelt out that the design of these components needs to be thought about carefully and that the cost of reuse is foresight.

The next chapter expands on how our design tackles the problems of usability and auditing.

Chapter 8

On the Importance of Users

Security is only as good as the user who is using it. The user is the ultimate source of authority and, if they can be tricked into authorising something untoward, then the security has failed. This chapter will outline several features of our system which make it ease to use and audit.

That observation introduces the complex field of secure interface design. Several projects known to us are exploring this space with respect to capabilities [MS][MSMS]. As an example, a capability application such as a word processor need not have any read access to a user's documents. Asking the word processor to open a document can trigger a trusted file open dialog which returns a capability to the single file selected to the word processor.

In such a system there are two levels of user interaction - interaction with an untrusted process (the word processor) and interaction with the trusted kernel (such as a file open dialog). These interactions must be clearly deliminated and this is a problem for user interface design.

Such problems are not covered by this project, we stay within the confines of back-end server processing (HTTP servers and the like). As such we have different problems - we need to be able to weave complex networks of capability processes and we need to be able to audit the graph of running processes in order to understand which are the weak points of a system.

To the first end we have created a very simple domain specific language for starting capability systems. Everything important about this language can be gleamed from an example:

```
process rmtfilesystem
→       code client/filesystem.py
→       - base-path-set fs
→       slave ice
→       unsecure

process acceptor
→       code ../cclient/cli-acceptor
→       grant inet-accept port 8000 as accept
→       connect connections http-parser.http-connections

process ssl-acceptor
→       code ../cclient/cli-acceptor
```

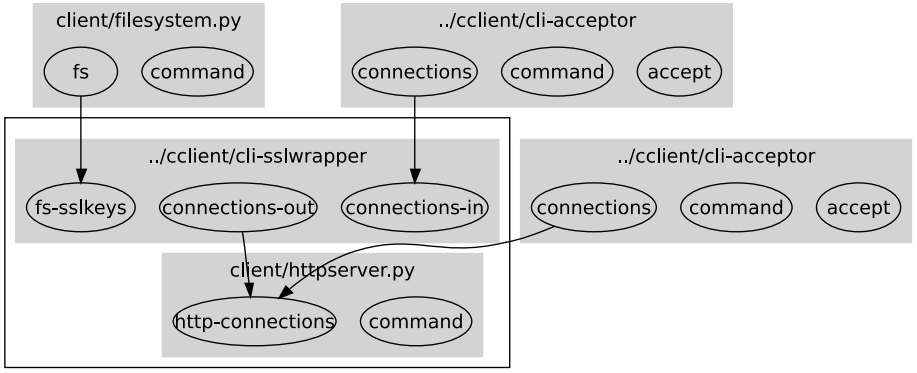
```
→       grant inet-accept port 8001 as accept
→       connect connections ssl-wrapper.connections-in

process ssl-wrapper
→       code ../cclient/cli-sslwrapper
→       connect connections-out http-parser.http-connections
→       slave ice

process http-parser
→       code client/httpserver.py
→       slave ice

connect
→       rmtfilesystem.fs ssl-wrapper.fs-sslkeys {path sslkeys}
```

The example sets up a capability system similar to the HTTP server example in the preceding chapter. The processes are split over two computers, the default host (being the one that the master is running on) and a slave host called `ice`. We can clearly see which processes are granted special authority (with the `grant`) command and how they are connected. We can also see that some processes are not restricted by the kernel (denoted by the keyword `unsecure`) as, in this case, they need to access the filesystem directly.

We claim that this form of configuration file allows easy auditing of capability systems and easy construction of systems from preexisting components.

It's also important to be able to discover the structure of the capability graph when the system is running. The configuration format above specifies an initial graph but processes may connect within themselves or a user may manually mutate the graph at some point after starting it.

Thus, with the help of another kernel patch, the currently active capability graph can be extracted. The result of doing this with the example above is the following:



The importance of this diagram is not that it can be produced. Indeed, such a diagram could be produced on existing systems and could reflect the structure of shell pipelines and other common structures. The importance of this diagram is that one can know that it is complete. When a process is shown connected to only a single other process then we can be sure that there is no ambient authority and that it cannot effect anything else by manipulating the filesystem etc.

This chapter finishes this stage of the report. The next chapter considers the problems of writing capability processes.

Chapter 9

Writing Capability Aware Programs

In previous chapters we have presented a design for a modular, secure and comprehensible framework for building servers. During which we have specified the behaviour of a number of processes. These processes are expected to correctly juggle input on a number of different capabilities and to maintain per-capability state. The software engineering required to do this is, unfortunately, still non-trivial.

The separation of processes requires that message passing be asynchronous. If this were not the case then an untrusted component could disable the rest of the system by simply ignoring all messages. Other process would soon block on sending and then yet more processes would block on sending to *those* processes, and so on.

As a concrete example of this find a UNIX server which is setup to log messages to a virtual console. This setup is useful for monitoring but the entire server can be completely disabled by pressing scroll lock. After a while the syslogger will block on writing to the console, then every process which logs messages to syslog will block on writing to the syslog daemon etc. Within a few minutes the only remedy (unless someone removes the scroll lock) is a power cycle.

Asynchronous message passing is also more than simply buffering messages, otherwise the buffers would overflow the memory and cause the same problems. Processes must be able to detect to flow control back pressure and respond in a way which doesn't cause problems in the face of disabled or hostile processes.

The usual method for doing this involves basing the program around a polling system call (on Linux this is one of `poll`, `select` or `epoll_wait`). This notifies the program of which I/O channels are ready for reading/writing and the program then can perform I/O without blocking and return to the polling call. If a program is reading three chunks of data from a capability, this results in a stack graph like the following:

In this, every I/O operation requires a return to the bottom of the call stack (assuming that the kernel delivers all available data in one operation). This precludes any state being kept on the stack, or implicit state being kept in the current instruction.

The alternative is to keep the stack across I/O operations and have the I/O operations 'block' the progression of the thread of control:

The synchronous design is not without its flaws. Threads are often implemented as preemptive kernel threads which may be run concurrently on different processors and be task switched at any time. This leads to the use of classic concurrency primitives: mutexs, rwlocks and condition variables. It also leads to deadlock, livelock, missed notifications and a fundamental axiom of our concurrency design:

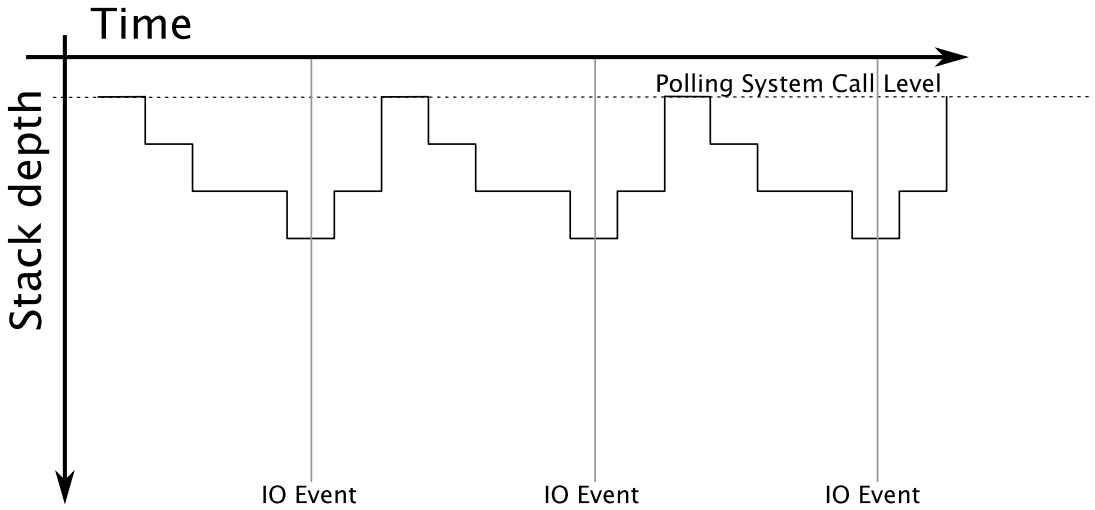


Figure 9.1: Stack graph for asynchronous design

We're too stupid to do locking correctly

Locking leads to bugs which are nearly impossible to track down since they are manifest only in very rare timing conditions. Locking, or rather the lack of it in the right places (or in the wrong order), also leads to race conditions and other security problems.

These problems, and failure in past projects to overcome them, has lead to us abandoning shared mutable state between threads. Instead we follow a set of principles from the language Erlang[Armb]:

9.1 Concurrency Orientated Programming

(The title comes from Jon Armstrong in his thesis[Armb])

Concurrency orientated programming is based around three principles:

- Threads have "share nothing" semantics.
- Message passing is the only way to pass data between processes.
- Isolation implies that message passing is asynchronous.

The careful reader should notice that these are exactly the same principles which were enumerated as the principles of capability systems. We propose exactly this: capability processes should be built exactly as capability systems are, for many of the same reasons.

Capability processes mark the edge of a domain of protection. They cannot interfere because the kernel isolates them from each other. Threads share the same address space so clearly they can interfere and a security break in any of them breaks the whole process. So we do not seek to reason about security breaks within a process and assume that a successful attack compromises the whole process. But we do suggest that this is a superior pattern for building concurrent programs.

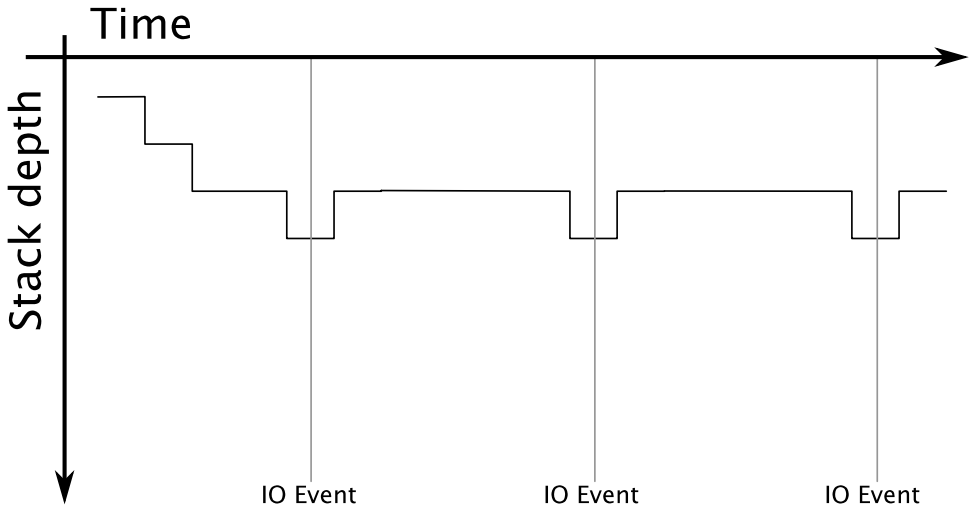


Figure 9.2: Stack graph for blocking design

This pattern of concurrency leads to a large number of cooperating actors, each of which is a small state machine. There is no hard and fast rule as to where the boundaries of different actors lie - it is often possible to conceive of several reasonable designs for dividing a problem up. This same situation also exists when determining how code should be broken up into functions and, with a little practice, a good decomposition can usually be found.

One extreme of the decomposition spectrum would be to formulate the whole process as a single state-machine. This corresponds to the asynchronous design presented above. We would suggest that, like writing a whole program as a single function, this is a very poor point in the design space.

We have developed two implementations of this, one in C++ and one in Python. We stated that a goal of the concurrency design was to be language agnostic and the best way to show this is to, in fact, have several working implementations.

Many of the choices for this design have been taken from Erlang. For those familiar with it, differences will be highlighted in parenthesis.

Messages consist of a name and a payload. In Erlang the name is an interned symbol, however both Python and C++ lack symbols and so we name messages using strings. This means that there need not be a central registry of all messages in the source code but does throw up bugs where a message name has been mistyped. Message names always follow a colon-and-hyphen format, such as `:io-flow-stop`.

Message payloads are either fixed structures (in C++) or free-form key-value dictionaries (in Python). Having a reference to an actor is necessary and sufficient to send a message to it.

9.1.1 Error Handling

Cooperating actors can be linked together. When an actor dies it causes a `:thread-death` message to be sent to every actor which has linked to it. The usual error handling pattern in most actors is to log the error and die, leaving it up to someone else to recover. This error handling strategy is very simple and very robust. When attempting to cover all possible errors there's always the likelihood that some case is missed. But in this design one can always rely on the fact that some higher level will manage the recovery when writing almost every actor.

Linking is very important to robustness. Most actors, upon receiving a `:thread-death` message, die immediately. Thus whole networks of actors tend to die at once. This is exactly what is required most of the time. For example, a failure in an `IOReader` actor, causing it to die, will kill all the other readers which process its data, the interaction actor and all the writers which that interaction actor used. Once all those actors are dead the garbage collector (or reference counter in C++) will close the file descriptor that the `IOReader` was using.

(In Erlang all links are bidirectional, so that linking to another actor will also cause it to be informed when you die. Links in our implementation are one way.)

Higher level actors handle `:thread-death` messages to perform cleanup without killing themselves. In the example above the `IOReader` actor will have been linked to by the core event handling actor which will note its death and delete any I/O handlers that it may have installed.

9.1.2 Flow Control

There is also a second network of actor linking - flow linking. Like regular links, flow links only run in one direction but are used for explicit flow control messages. Since sending a message is asynchronous (thus doesn't block), an actor which has to process that message (e.g. send it over the network) has to have a way of signaling that it is already busy. It does this by sending an `:io-flow-stop` message along all its outgoing flow links.

For example, an `IOWriter` actor sends data across the network. All I/O calls are nonblocking (otherwise the actor would be blocking on I/O most of the time and wouldn't be able to respond to any messages). Thus it has to queue up data messages which haven't been sent yet. When this queue reaches a certain length it emits an `:io-flow-stop` message. Hopefully this message will be passed around the network of flow linked processes to an `IOReader`. The `IOReader`, upon receiving a `:io-flow-stop` message pauses reading from the network. This gives the `IOWriter` a chance to send some of the data until its queue falls below a given length and it emits an `:io-flow-start` which causes the `IOReader` to resume reading.

(Flow linking doesn't appear in Erlang.)

There is no provision at the moment for an actor to be part of multiple, separate flow networks. If, at any point, this is needed it's probably a sign that the actor is doing too much and needs to be broken down.

Edge driven I/O

As mentioned above, in order for actors to be sensitive to messages they do not block on I/O calls. A single, special actor in the system is responsible for all I/O events. Threads which wish to do I/O register their interest in events pertaining to a specified file descriptor and the event actor sends them `:io-event` messages when that file descriptor is ready.

Traditional I/O multiplexing system calls (such as `select` or `poll`) are level triggered and this presents a problem.

Imagine that the event actor is using `select`/`poll` to wait for events. Data arrives from the network, sending the descriptor high, and the poll call returns. It fires a message off to the correct actor and continues.

However, the next poll call will return immediately because the other actor probably hasn't had a chance to perform a `read` and empty the kernel buffers yet. So the only option is to remove the descriptor from the set of 'interesting' ones and force any actor which wants to do I/O to re-enable it with a message as soon as they have finished reading.

This causes large numbers of internal messages to constantly signal events and re-enable them.

Recent multiplexing calls (`epoll` under Linux 2.6, `kqueue` under FreeBSD and realtime signals under Linux 2.4) have an edge triggered mode. In this mode the call only delivers events on a rising edge. So if a socket *becomes* readable it will tell you once. `select` and `poll` will tell you *whenever* the socket is readable.

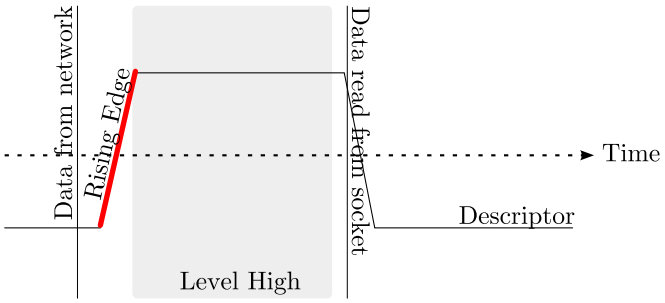


Figure 9.3: Event driven I/O notification

This is clearly ideal for our model. The event actor waits for edge notifications and sends messages. The descriptor doesn't need to be removed from the interesting set and another actor can read the data at its leisure.

In the case of flow control, even, the descriptor need not be removed. A actor may ignore the edge message if it doesn't currently want more data. In the future it can read until `EAGAIN` and then start waiting for edge messages.

Advantages of Message-Passing Concurrency

Firstly, message-passing concurrency eliminates locking. Since no mutable data is shared between process there is no need to lock anything. Syncronisation is managed through message passing and any shared data structure should be owned by an actor who's job it is to handle update and query requests. This is a pattern similar to object-orientated programming but we don't couple data with the functions for managing it, but rather with the sole actor which will update it.

Most importantly, message-passing concurrency fits perfectly with the idea of pervasive unit testing. The advantages of unit testing were elaborated earlier in this report and this project would not have worked without it.

This pattern, although rare, has been used very successfully (with Erlang) to build the Erikson AXD301 - Erikson's flagship telephone switch which manages 9-nines reliability (on average only 30ms of downtime per switch per year).

Helpful Patterns

Several patterns pertaining to the design of actors proved themselves useful in this project.

Firstly, *actors should not read any globals*. The `IOReader` actor needs a reference to the core events actor in order to receive notification when network data is ready (see above). The core events actor is a singleton and therefore a reference is available from a global variable.

However, in order to allow unit-testing to isolate an actor from the real world the IOReader should always expect the reference to a global actor to be passed at construction time.

Secondly, *actors should not build other actors where possible*. An `LSPWriter` accepts LSP messages, serialises them and sends the resulting data messages to an `IOWriter`. Thus a possible design is for the `LSPWriter` to create the `IOWriter` itself at construction time.

This distributes information about how actors are linked together over many different actor constructors and makes it more difficult to use actors in odd situations (or to unit-test them). Thus structural information (such as the common relationship between `LSPWriter`s and `IOWriter`s) should be limited to as few places as possible. Actors should not reference other actors by class name unless absolutely necessary.

9.2 Related Work

Clearly this work is very much based on the Erlang language, but it's worth contrasting it with other concurrency designs.

There is much work on concurrency which is unrelated to this: better traditional concurrency primitives in languages, static analysis of those primitives, transactional memory. These are only mentioned to point out that they are following a different path and so we do not consider them here.

SEDA[WCB01] (*staged, event-driven architecture*) is a family of frameworks built in Java which seek to tackle the (increasingly inaccurately named) C10K problem. Thus they are very much focused on high performance, something which has not motivated us.

Their frameworks are built on Java and based on the NIO APIs which are level-triggered asynchronous APIs (like `select` and `poll`). Their webserver design passes the standard benchmark of performance in that it's faster than Apache.

The most fundamental difference is that their graph of threads is static. Each step in handling a request is decomposed into stages and the resources are intelligently allocated to each stage needing processing. However, the number of stages does not change at run-time. There is a stage for reading the data of a HTTP request from the network and it's a single stage for all incoming requests. Our, Erlang like, design would spawn a number of IO threads for each connection and so mutate the graph at run time (see graph in next section). We also have nothing akin to their resource allocation algorithms.

However, a SEDA graph is very much like the graph of *capabilities* shown in chapter eight. The capability graph is indeed static and something very much like it appears in the SEDA paper. This suggests future work could fruitfully investigate resource allocation across a capability graph for better performance.

Although we don't concern ourselves with performance too much (most of the code is written in Python) we are confidant that actor based systems have no fundamental weaknesses in this area as an Erlang webserver can vastly outperform Apache[Arma] for highly concurrent loads.

9.3 Implementation Details

C++ Implementation

For the C++ code actors were mapped onto full operating system threads for ease of implementation. OS threads are usually considered too heavy weight to be used lightly, but recent NPTL Linux kernels can perform thread creation in 100 microseconds and with 20K of address space. (Measured on a PII 450Mhz with Linux 2.6.11). This suggests that a default 32-bit address space of 2GiB can support 100,000 threads and that we could create them all in 10 seconds.

(Sadly, a lesser limit is imposed in `kernel/fork.c` based on the number of memory pages. This reduces the number of threads to about 32,000 with 512MiB of memory.)

Message passing was implemented using a per-thread queue built using the usual POSIX thread functions. Only pointers to messages are passed about as an optimisation, but this does not violate the rule of no mutable shared state since the messages are read-only once sent.

Using true operating system threads for the C++ implementation meant that processes like the SSL wrapper (see chapter seven), which are CPU bound in many places, could be built with low latency and could take advantage of multi-processor hosts.

The C++ implementation of the Acceptor and SSLWrapper processes runs to about 4500 lines of code.

Python Implementation

Python handles operating system level threading very poorly so actors in the Python version are cooperative and use stack manipulation to save themselves and yield to other actors. The stack manipulation code was taken from Stackless[Tis].

The Python implementation of everything (including the master process) runs to about 8000 lines of code.

As a demonstration of the similarities between the graph of threads, and the capability graph we extracted such a graph from a running master process. The arcs between threads are references, not counting the link of flow networks. See figure 9.4.

9.4 Where we went wrong

Both of our implementations involve implementing actors as threads in the sense in that the actors can store state in their stack. If you refer to the stack diagrams in chapter nine then our actors can be considered synchronous designs. This design was chosen because it mirrors that of Erlang and, at the time, we didn't have enough experience to know better.

However, it has become clear that this is not necessary and, in fact, is wasteful as a default style for actors.

Almost all the actors end up following this general format:

```
def run():
    perform-startup-operations()

    while True:
```

```
message = self.recv()

handle-message(message)
```

(That is clearly Python code, but most C++ actors look similar modulo language.)

This is wasteful since the stack need not be preserved between messages. If the actor could be implemented with a single `message_received` callback then they could be much lighter in both implementations.

Once actors are callback based they can be split into thread pools where the number of threads is based on the number of CPUs in the host system. Actors also know if they are CPU bound (e.g the handshaking thread in an SSL wrapper which performs many public-key operations). These actors could be split off into their own threads at construction time.

There were two notable exceptions to this pattern: the LSP and HTTP parsers. Both of these actors were written in a style in which the stack held state across messages and reflected the structure of the LSP or HTTP message being decoded.

If we were to build callback based actors in a future system we would either need to provide the option of stackful actors or we need a good way to transform these stackful designs into callback based ones.

9.5 Parsers for Network Protocols

Both of the examples of stackful actors, above, were parsers for network protocols. Stack-free parsers are well studied and understood, yet are almost never used for parsing network protocols for a number of reasons:

- They are often designed for parsing files which are parsed all at once. Although the parser may be left-to-right based they produce an entire parse tree at the end with no option of emitting partial parse trees when an interesting section of the input has been parsed.
- Generated parsers are designed to work with a tokenising pre-processor. This doesn't work for network protocols since they often mix binary and textual data together. The information required to tokenise it is in the parse tree which, by definition, isn't available until parsing.
- Generated parsers tend to produce foul right-recursive parse trees which are very hard to use. To make it worse; without a tokeniser the parse tree is right-recursive for every byte.
- Writing a grammar for a parser generator is hard and requires knowledge of how the generated parser works to avoid writing obvious, yet unusable, grammars. The EBNF, provided in many RFCs, is unusable in this respect.

With this in mind we built an SLR parser generator in Python which was designed to address these issues and allow us to write a stack-free HTTP parser as a proof of concept.

Our grammars allow any reduction to be marked as emitable - which causes a partial parse tree, rooted at that reduction, to be emitted as soon as it's parsed. This allows, for example, the URL in a HTTP request to be checked before the full header has been received. The emitted section of the parse tree can also be omitted from the full parse tree to save memory.

Our parser has no tokenising stage and its terminals are single bytes. It's 8-bit clean and can work with both text and length prefixed binary data. The length information can be taken from the partial parse tree.

For example, a HTTP `POST` request involves a text based header followed by a series of chunks of binary data where the length of each chunk of data is given in hex just before it. Our generator can produce a parser which can read the length, validate it, convert it to a number, expect that many bytes of binary data, emit the data as soon as it's read and not include it in the final parse tree to save memory.

Our grammar is also specified as Python code and provides many helpful shortcuts:

In general a string would be specified as:

```
TokenString = TokenChar
TokenString = TokenChar TokenString
```

For the input `POST` this would produce the parse tree `['P', ['O', ['S', ['T']]]]`. For this case we provide a helpful function `OneOrMore` which can be used like:

```
TokenString = OneOrMore(TokenChar, _string)
```

The `_string` parameter causes the resulting parse tree to be a string, not a right recursive list of bytes. If no predefined functions are suitable then *reduction functions* can be defined in place:

```
Token = OneOrMore(TokenChar)
def Token_reduce(values):
    return ''.join(values)
```

A reduction function receives all the terminals and non-terminals for a given reduction and returns the result to the parse tree. It can also cause a partial parse tree to be emitted, can reset the parser and can switch it to a different grammar.

Reduction functions are the key to making the parser framework work. Since the parser is SLR it cannot backtrack and so checking strings is a problem. Consider the rule:

```
Request = "PULL" | "PUT"
```

That rule is unworkable with an SLR parser generator because it will read the `P` and then be at a loss as to which path to follow - is it a `PUT` or a `PULL`? The trick is to parse it generically as a string of uppercase letters and make the reduction function check that it's a valid command:

```
RequestString = OneOrMore(uppercase-letter, _string)
Request = RequestString
def Request_reduce(values):
    if values[0] not in ['PULL', 'PUT']:
        raise SyntaxError()
    return values[0]
```

We concede that this requires more knowledge of the working of the parser from the grammar writer than we would like, but we believe it to be the best solution given a non-backtracking parser.

Since the parser is written in pure Python it is a little slow. On a 450Mhz PII it manages to parse about 200 HTTP headers/second which is a rate of about 82 KB/s.

Future Work

A parser generator is simply a way of automatically generating state machines. But parsing isn't the only state machine involved in dealing with network protocols. Protocols also often need a higher-level state machine to manage their request-response nature:

```
220 imperialviolet.org ESMTP
MAIL FROM: foo@foo.com
250 ok
RCPT TO: bar@bar.net
250 ok
DATA
354 go ahead
Subject:  testing

Hello there!
.
250 ok 1114338782 qp 5232
QUIT
221 imperialviolet.org
```

In SMTP, a valid conversation can't have a `DATA` after the `RCPT TO` if the `RCPT TO` failed. Currently, the only option would be to have to parser working line-by-line and have a higher level state machine tracking the set of valid commands at each point. (Although a parser for each line of SMTP would probably be better written by hand.)

So let us introduce two new terminals: $\top$ and $\bot$ and let them stand for "command successful" and "command failed". We can inject these into the parser when we're finished processing a command and then define a conversation thus:

```
RCPTTO = RCPTTOCommand
RCPTTO = RCPTTOCommand, RCPTTO
SMTPConversation := HELOCommand, ⊤, MAILFROM, ⊤, RCPTTO, ⊤, DATA, ⊤
```

(That's not at all valid according the the RFC, but we are just illustrating a point.)

A client may send as many failed `RCPT TO` commands as they wish, but can't send a `DATA` command until one has completed. Thus, if the parser parsed it, it's a valid command and you don't need to keep track of the state at a higher level.

(This section is speculative and has not been implemented in this project.)

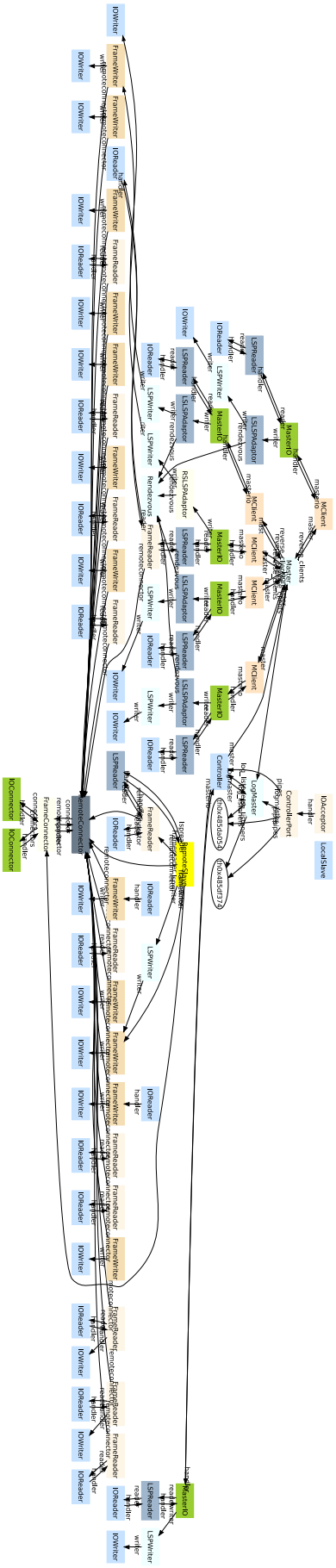


Figure 9.4: The graph of threads for a running master process

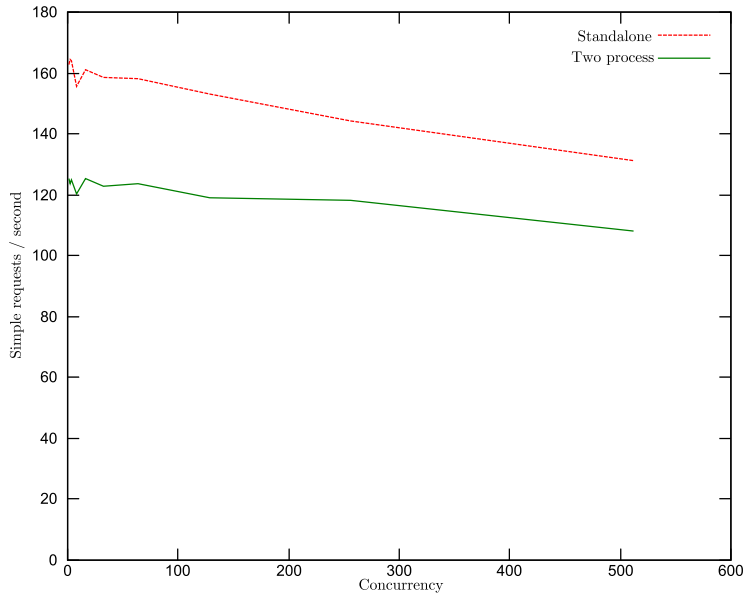
Chapter 10

Conclusions

Performance has not been a major consideration in this project, but it's important that capability designs don't impose too high an overhead.

The graph below shows the transaction throughput for two configurations of the HTTPServer module. In the 'standalone' configuration the HTTPServer opens a listening socket directly. In the 'two process' configuration is it attached to an Acceptor process. In both cases processing the HTTP request just involves returning a static (in memory) web page which is less than 100 bytes.

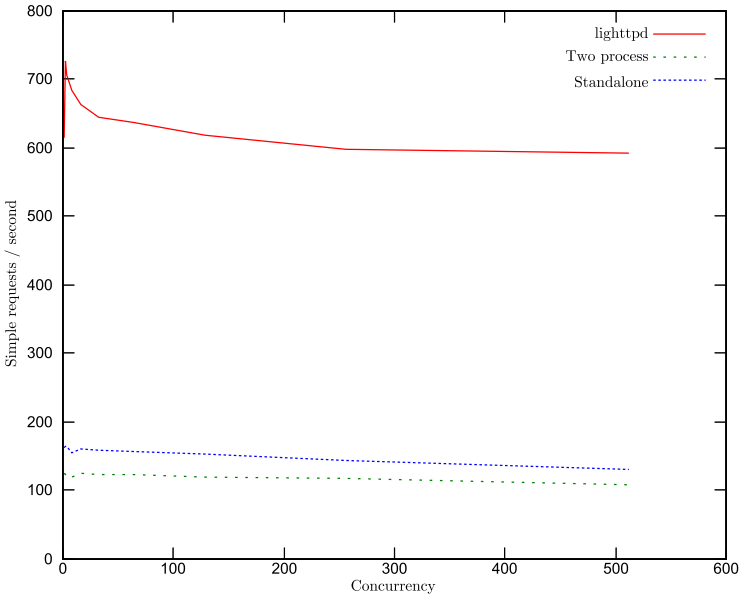
This setup should show the overhead of message passing to the greatest degree. The actual processing involved in the request is minimal and the the HTTPServer is written in Python which is not suited to processing large amounts of binary data. We have chosen this setup because it sets an upper bound to the overhead.



Neither process performs very well since it's written in Python. The overhead of the two process configuration is almost all due to the Python parsing of the LSP messages. The Acceptor process is almost

completely idle. The importance of the overhead can be seen in the next graph which includes the results of running the same test against a pure-C server (`lighttpd`). Clearly the trick is not to write high-performance servers in Python.

Although there is a performance overhead of using a modular, capability design it isn't very large even in this test, which should show it off to the greatest degree. In situations where the LSP parser can be written in a compiled language (or even as a C-Python module) and where the processing of a request involves more than returning a static string, the overhead should be acceptably small.



10.1 Reflections & Further Work

In reflection there are several points which suggest themselves as obvious places where improvements need to be made. These are listed below. In general we feel that the basic idea of implementing capabilities with file descriptors in a way which allows for their gradual introduction is a sound one. However, on top of that there are many, many details with no clear answer and it's in these that we find that we are uncertain about our decisions.

The libraries, for both languages, could do with a major redesign in light of the lessons learnt in the course of this project. These issues are very difficult to explain in the course of writing a report since their understanding requires a familiarity with the code but many small decisions in this area turned out to be wrong.

At a higher level the Master process works very well but highly distributed systems will need the ability to communicate directly between hosts. Having the master process on a single host proxy the traffic was a simplifying decision and was correct for the scope of this project but is too restrictive in general.

LSP messages are already size limited in the specification to avoid clients having to receive (and buffer) extremely large structures. Since this is so it was silly not to length prefix them. This would remove the need for any parsing of partial LSP messages making this (critical) function faster.

The question of robustness needs to be addressed. At the moment a process which fails has its capabilities closed by the kernel. Likewise, a host which fails will timeout the connections to the master host. At the moment, neither can be restarted without user action. Building robust systems is a perennial problem but some way to restart failed processes and set them up correctly, even when a whole host has failed, will be needed if any critical services were ever to run on a capability system such as this.

As ever, user interfaces provide fertile ground for future work. We have cited several other projects which are working on this problem[MS][MSMS] but the features outlined in chapter eight will not do for a general non-technical user. There is no reason why capability ideas shouldn't be applied across the desktop but the problem of communicating these ideas to a general user is a significant one.

There are many other places in most systems where a more sensible architecture is sorely needed. SUID binaries, for one, have been a source of many security problems over the years and are a gratuitous violation of the principle of least authority.

Although an alternative to that design might have little relation to the design presented in this report I would hope that the spirit of it could be carried over. Certainly the idea of using UNIX domain sockets to communicate with trusted/untrusted actors immediately suggests a reasonable design for doing away with the need for any SUID binaries in most cases.

If this project can introduce people to the ideas of capabilities I feel that it has done its work. More than anything a change of mind set is required to produce reliable software. As good as 'best practices' are in reducing security problems (and the introduction of safe langauges also helps in this respect) the principle of least authority is still vital, and still ignored.

Examples of software which plan for security problems and are structured accordingly are still the exception; `qmail` and `openssh` are the only ones which come to mind. Mistakes have been made in the past and should be corrected given that we now have the hindsight to do so.

But, although there are many willing to correct these problems their work will be in vain so long as they fail to plan for the change. Work which starts from a clean slate will always be limited to mapping out the territory far away. Real security benefits will come from ugly, impure work which moves in small steps.

Appendix A

Specification

A.1 LSP Wire Protocol

Nearly all communication between capability aware processes uses LSP to serialise data structures. It's such a trivial protocol that it's almost not worth standardising on, but then so is the format of `/etc/passwd`, and that has enabled many different tools to inter-operate.

This wire protocol is a little special because it allows capabilities to be passed. These aren't bit-patterns but are special out-of-band element handled by the kernel.

```
LSP Message := Element
Element := List | Dict | Symbol | Num | Cap
  List   := <u8: 0> Element* <u8: 6>
  Symbol := <u8: 2> <u16: LENGTH> <LENGTH bytes of data>
  Num    := <u8: 4> <u64>
  Cap    := <u8: 5> <out-of-band file descriptor> <u8: 0x42>
  Dict   := <u8: 1> (Symbol, Element)* <u8: 7>
```

LSP Messages may not be longer than 256KB. Any LSP based protocol which needs to transmit large amounts of data must break it into several LSP messages.

A.2 Master Interaction Protocol

Abbreviated to MIO, the master interaction protocol is so called because it's used by the master to communicate to child processes. It's a very simple layer on top of LSP which specifies a request-reply series of messages.

Either side may send a request which may start with the symbol `fire-and-forget`, in which case no reply should be sent. Replies start with the symbol `ok` or `error` and are always returned in the same order as the requests which triggered them.

The MIO dispatcher (on the receiving end) must not dispatch a request until the reply of the previous request has been generated (unless it's marked `fire-and-forget`, in which case the requests need only be in the same order).

A.3 Master Protocol

The master interaction protocol specifies how requests should be sent while the master protocol specifies exactly what requests the master process may send to its child processes. At startup, child processes should expect to find a file descriptor in slot three, which is their socket to the master process. At this time all requests are made by the master and all child messages must only be replies to these requests.

If the master socket disconnects the child should exit.

Command	Arguments	Description
`query-ports`	*none*	The client should reply with a list of dictionaries, where each entry in the list describes a single port. Each dictionary should have the keys `name`, `type`, `direction` and `pipes`. `name` and `type` are free form strings, `direction` should be one of {0, 1, 2} which mean {incoming, outgoing and bidirectional} respectively (these are non binding and only serve as metadata for the user). `pipes` should be a list of inode numbers of sockets which are currently connected to this port.
`connect`	*port name*, *capability* and *extra*	This is sent when a new connection is made to a port. The *extra* argument is optional but must take the form of a dictionary if provided.

A.4 Interhost Communications

All interhost traffic is carried over TCP without any additional security for the moment. All connections are started by the master process to the slaves. The master writes a 9 byte header on each new connection:

```
SlaveHeader := IncommingHeader | OutgoingHeader | ControlHeader
  IncommingHeader := <u8: 0> <u64: connection id>
  OutgoingHeader := <u8: 1> <u64: connection id>
  ControlHeader := <u8: 2> <u64>
```

The master assigns connections as incoming, outgoing or control. Control connections carry LSP data which follows the slave control protocol (detailed after). If a slave accepts a new control connection it should be considered a reset - old children should be killed etc.

Incoming and outgoing connections are structured as:

```
Chunk := DataChunk | CapabilityChunk
  DataChunk := <u8: 0> <u64: LENGTH> <LENGTH bytes of data>
  CapabilityChunk := <u8: 1> <u64: connection id>
```

For example, a message from master to slave which contained a capability in between two chunks of data would generate three chunks along a TCP connection. The capability chunk would specify the connection id of a connection (which would be ‘incoming’ from the point of view of the slave). The slave would pass on the first chunk of data then create a local socket pair and pass on one end before passing the final chunk of data. The slave must then forward data between its end of the socket pair and the connection specified in the capability chunk. That new connection may also carry capability chunks and so on. The

slave must also deal with the fact that the specified connection may not currently exist and so should wait until a connection with the correct id is created.

Command	Arguments	Description
`exec`	*filename*, *cid*	The slave should fork and exec the filename given. The slave should return a message containing the same *cid* and a capability to the new client process.

A.5 Manifest Format

A manifest file consists of a series of stanzas. Each stanza is either 'simple' (consisting of a single command) or 'complex' (consisting of a main command and a number of subcommands).

The only simple stanza is `registered`, which takes a single string as an argument and denotes that a process with that name should have been registered with the master. That process name can then be used in future stanzas.

The complex stanzas are `process` and `connect`. The `process` stanzas takes a single string which is the name of the new process and number of subcommands:

Command	Arguments	Description
`code`	*filename*	Specifies the location of the code for this process. The code can either be a binary or a Python source file (which must end in `.py`).
`slave`	*slavename*	This process should be run on the given slave host
`grant`	*type*, [*options*] `as` *name*	Grants a named capability of the given type to the process. The type should be `inet-accept` and the options should be a dictionary with `port` given.
`connect`	*portname*, *destination*	Connect the given port name on this process to the destination. The format of the destination should be `processname.portname`. The process need not have been specified already.
`register`	*name*	Register this process with the given name
`-`	`command`	Connect to the process and run the given command. The format of the command is specific to the process.

The `connect` stanza takes a number of subcommands like `connect` except that the command name is omitted and both process names have to be given.

See the example in chapter eight.

Bibliography

[AP67] William B. Ackerman and William W. Plummer. An implementation of a multiprocessing computer system. In *SOSP '67: Proceedings of the first ACM symposium on Operating System Principles*, pages 5.1–5.10. ACM Press, 1967.

[Arma] Joe Armstrong. Apache vs. Yaws.

[Armb] Joe Armstrong. Making reliable distributed systems in the presence of software errors. Master's thesis.

[Ber04] Dan Bernstein. http://cr.yp.to/maildisasters/sendmail.html, December 2004.

[Boe84] W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. *Proceedings of the 7th DoD/NBS Computer Security Conference*, September 1984.

[Bur61] Burroughs. Definition of the B5000 Information Processing System. Technical report, Burroughs Corporation, 1961.

[Cor05] Red Hat Corp. http://fedora.redhat.com/, May 2005.

[coy] Coyotos Microkernel Specification.

[Fab67] R. Fabry. A user's view of capabilities. *ICR Quarterly Report*, November 1967.

[GEK+02] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceno, Russell Hunt, and Thomas Pinckney. Fast and flexible Application-Level Networking on Exokernel Systems. *ACM Transactions on Computer Systems*, 20(1):49–83, February 2002.

[Gon89] Li Gong. On security in capability-based systems. *SIGOPS Oper. Syst. Rev.*, 23(2):56–60, 1989.

[Gro] The CVE Group. http://www.cve.mitre.org/about/.

[Har85] Norman Hardy. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985.

[Har88] Norm Hardy. The Confused Deputy. Technical report, Key Logic, Inc., 1988.

[Jac83] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 26(1):29–35, 1983.

[KS74] P.A. Karger and R.R. Schell. Multics Security Evaluation: Vulnerability Analysis. *ESD-TR-74-193*, 2, June 1974.

[Lan] The E Language. http://erights.org/.

[Lev84] Henry M. Levy. *Capability-based Computer Systems*. Digital P.,U.S., December 1984.

[LS76] Butler W. Lampson and Howard E. Sturgis. Reflections on an operating system design. *Commun. ACM*, 19(5):251–265, 1976.

[LS01] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[lwn04] Security. *Linux Weekly News*, November 18^{th} 2004.

[May82] Alastair J. W. Mayer. The architecture of the Burroughs B5000: 20 years later and still ahead of the times? *SIGARCH Comput. Archit. News*, 10(4):3–10, 1982.

[MK] Cliff Frey et al Maxwell Krohn, Petros Efstathopoulos.

[MS] Ka-Ping Yee1 Mark Miller Marc Stiegler, Alan H. Karp. Towards Virus Safe Computing for Windows XP.

[MSMS] Bill Tulloh Mark S. Miller and Jonathan Shapiro. The Structure of Authority.

[nas05] They write the right stuff. *Fast Company*, (6):95–, 2005.

[pla] Plan 9 From Bell Labs. Technical report.

[SA02] Jonathan S. Shapiro and Jonathan Adams. Design Evolution of the EROS Single-Level Store. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 59–72, Berkeley, CA, USA, 2002. USENIX Association.

[Sea04] Mark Seaborn. http://www.cs.jhu.edu/ seaborn/plash/plash.html, December 2004.

[She68] J. H. Shepherd. Principal design features of the multi-computer. *ICR Quarterly Report*, November 1968.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. *SIGOPS Oper. Syst. Rev.*, 33(5):170–185, 1999.

[TA90] A. S. Tanenbaum and Et. Al. The Amoeba distributed operating system, 1990.

[Tis] Christian Tismer. Continuations and Stackless Python.

[VT04] Worms Viruses and Trojans. http://www.bcentral.co.uk/issues/security/windowssecurity-/maliciouscode.mspx, December 2004.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM Press.