

Security-oriented fast local RPC

Security-oriented fast local RPC

Copyright © 2006 Michael Hearn

This report is submitted to the Board of Examiners in the Department of Computer Science, University of Durham as part of the degree of Michael Hearn.

It consists of 17,478 words.

Abstract

Computer security is a subject that increasingly concerns everyone in our society. This project addresses the challenge of keeping systems secure in the face of malicious programming. It considers a new approach to the problem by analysing current approaches and evaluating their usefulness. It explains the historical context and summarises the issues. By examining the underlying principles of damage minimisation, the criteria for successful security enhancement are clarified.

The report documents a new form of remote procedure call (RPC) designed to make splitting software components out into separate processes easy. These separated processes can then be confined using the kernels privilege separation framework. Included is an overview of how it works, and a description of the design tradeoffs involved in its development.

The key criteria for success are improved raw performance and improved ease of integration with existing systems. These are tested and the implementation developed in this project, *FastRPC*, is shown to meet both criteria. Test results are presented that demonstrate a ten-fold improvement in performance compared to the closest equivalents, and a case study is used to show improved ease of integration. Finally ideas for further work are presented.

Table of Contents

1. Introduction	1
In the beginning	1
Mandatory Access Control	2
Summary	4
Definitions	5
Objectives	6
Criteria for success	7
Example	8
Implementation Plan	8
Choice of software	8
Guide to contents	9
2. Fast RPC in the literature	11
Performance and ease of integration	11
Performance in the literature	11
SHRIMP RPC	12
QuickLPC	13
LRPC	14
Generic optimisation	14
3. New Approach	16
A new RPC design	16
Resource sharing	17
Address space management	18
Precise MMU based access control	20
The heap	23
The stack	24
Designed for performance	25
Overheads in traditional RPC frameworks	25
Overheads in FastRPC	28
Designed for ease of integration	29
4. Implementation	31
Developer API	31
Basic steps	31
Memory management	34

Miscellaneous APIs	35
Master-side engine	36
Slave-side engine	37
Testing scheme	39
5. Results and Evaluation	41
MS-RPC	41
Performance results	41
Evaluation of performance results	43
Code impact results	44
6. Image Viewer Case Study	48
The threat	48
The code	48
The RPC framework	49
7. Conclusions	51
Problems encountered	52
Integration with third party libraries	52
Limitations of Linux kernel security	52
ASLR	53
Ease of debugging	53
Areas for future work	54
Improving framework security	54
Multi-threading support	55
Handling callbacks and re-entrant code	55
Developing an automatic partitioning algorithm	56
Windows support	56
Overall conclusion	57
Bibliography	59
A. Example code	60
FastRPC initialization code	60
MS-RPC Client/Master code	60
MS-RPC Server/Slave code	62
MS-RPC IDL	64

List of Figures

1.1. The single process, un-separated model	3
1.2. The multi-process, privilege separated model	4
3.1. The address space of the less privileged code starts as a subset of the more privileged code	22
3.2. RPC in a DCOM-style framework	27
3.3. RPC in FastRPC	28
5.1. FastRPC vs MS-RPC timings from the "perftest" program	43
5.2. Lines of code comparison	46
6.1. The image viewer after a crash	49

List of Tables

5.1. Raw timings for FastRPC vs MS-RPC	42
5.2. Lines of code in RPC-less and FastRPC perftests	44
5.3. Lines of code in MS-RPC perftests	45

Chapter 1. Introduction

Crime is everywhere. Russian mafias blackmail online businesses,¹ remote-controlled viruses steal credit card details², and spam kings make millions³ by sending untraceable email via home computer systems that can be bought on the black market⁴. Put simply, the internet has been subverted by organized crime.

How did we get here, and why? Can we do anything about it? And if not, is it too late to stop it happening again in the future? These are the questions asked in the introduction to this dissertation. Later chapters will cover a detailed proposal, named *FastRPC*, the production of a prototype, evaluation of its effectiveness and ideas for further work.

A definitions section is provided towards the end of this chapter, which defines various terms used throughout the paper.

In the beginning ...

There was networking, and there was C. Computers were not very fast and as a result the languages which became popular were those that focused on performance. Both C and C++ had minimal safety features, like array bounds checking, and both allowed for and promoted what we now call "unsafe" coding - for instance, reading into buffers allocated directly on the stack.

This made things easy for software developers and gave them a high degree of control, but it had an unintended side effect. It became easy to accidentally incorporate many different flaws into software, which could then be exploited by malicious programmers to gain control of the program and through it, the system. Historically the most common type of exploit has been the "buffer overflow" [Darpa2000], but other types of exploits involve integer or heap overflows and taking advantage of race conditions.

The C language was originally written for the UNIX operating system [Ritchie93] under development at Bell Labs⁵. UNIX featured a reasonably standard security system based around the processors memory management unit (MMU) and security levels keeping processes separate, communicating with each other and the hardware only via the kernel. Every running process ran in the security context of a user, and privileges were assigned to users not programs. If a user ran a program, that program could do anything the user could do. As a bonus, an all powerful administrator user (the "root" user) was available on every system.

This security system is called Discretionary Access Control (DAC) [Moffett94] and made a lot of sense when it was designed, as the primary purpose was to stop users interfering with each other on a multi-user timesharing system. Software was seen as a tool that was used by competent professionals - the concept of protecting the system from malicious software was unheard of.

As networking, and the Internet in particular, became prevalent the issue of compromised programs became bigger. People began to abuse DAC to sandbox vulnerable applications like FTP or web servers. By creating a "user" that did not actually correspond to a real world person, it was possible to limit the damage caused by a compromised process. For instance, it is common for Linux servers to have a dedicated "apache" user that the web server runs as which has very limited permissions. That way, even if Apache is hacked, the attacker cannot get access to other programs or directories.

Unfortunately, this system was not as flexible as it could be. For maximum security it is desirable to sandbox every application to have only the privileges it needs and no more but it is not practical to have a separate user account for every application. Another problem is that user/group security is very coarse grained, as it only restricts access to files and directories - for instance, you can not say that program A may not communicate with program B using DAC.

Mandatory Access Control

To solve these problems, Mandatory Access Control (MAC) was developed. The most widespread implementation of this type of security system is SELinux, developed by the NSA⁶. The SELinux security system centralises access control decisions in a system "policy". Because policy is all kept in one place, it can be analyzed using automated tools and controlled by the system administrator. In other words, this is quite different to other forms of security such as DAC in which application privileges are determined by the access level of the running user, or Java/.NET style Code Access Security (CAS) in which applications ask for the fine-grained permissions they need, as they need them.

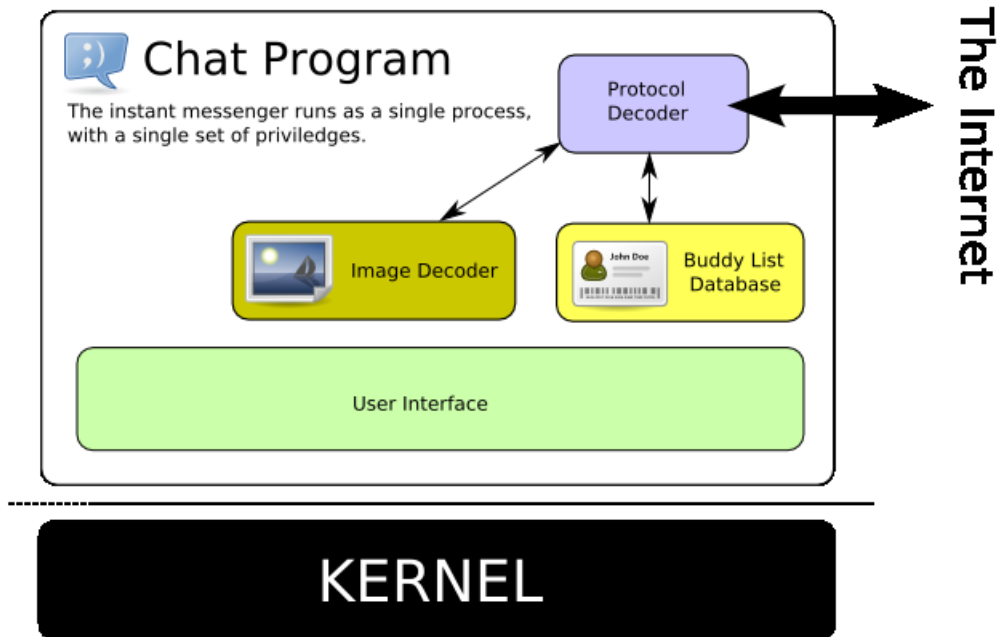


SELinux is not the only MAC framework in existence. Novells AppArmor⁷ framework is a simpler realisation of the same ideas, and Core Labs CoreForce⁸ is an implementation for Windows.

The NSA has published a paper, entitled "The inevitability of failure" which outlines the motivation for MAC security[NSA98]. They argue that given the currently high usage of uncheckable

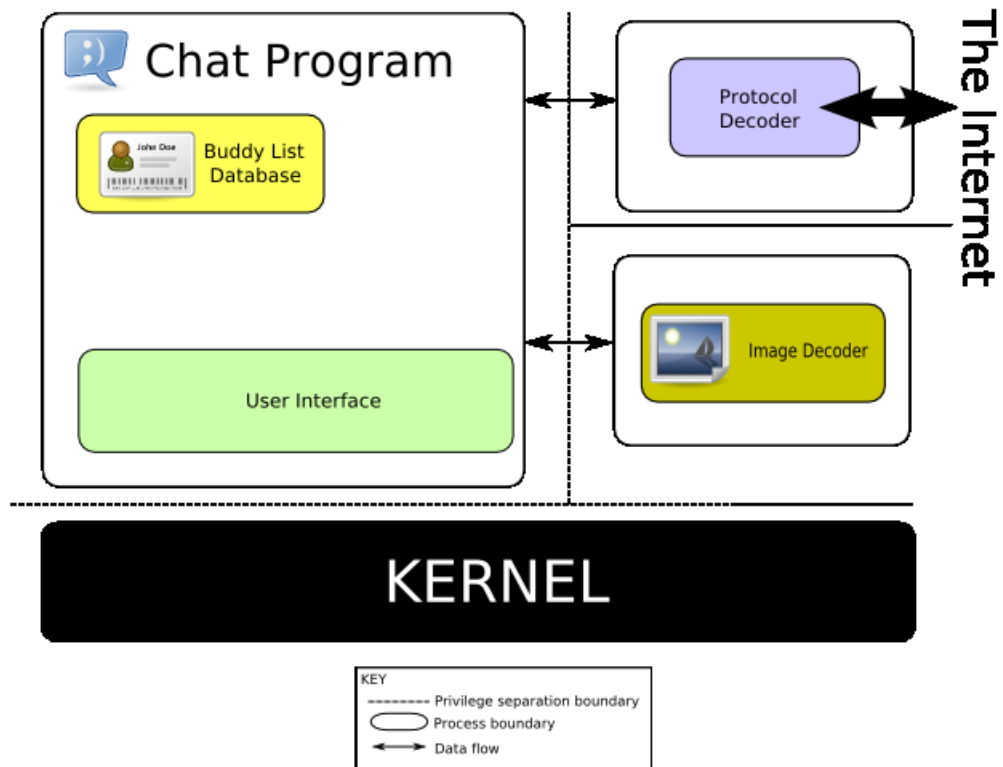
languages like C and C++⁹, software being hacked is inevitable. Their conclusion is therefore that security work should focus on privilege separation so when the inevitable does happen, damage is limited.

Figure 1.1. The single process, un-separated model



The NSA's SELinux MAC system allows for very precise specification of what privileges software does and does not have. However, it suffers from a problem – security contexts when applied to code have process scope not module scope. In other words, whilst you can stop the Mozilla web browser trying to wipe your hard disk, you cannot stop it modifying the files in your home directory as web browsers need this privilege so you can download files. As an example, consider the image loader exploits that started circulating around the start of 2005. Software that loads images does not need any privileges, as it is a pure data transformation – normally, decompressing or resampling image data into raw RGBA form. Because it involves complex data structure manipulation in C/C++ this code is a prime target for exploit writers, and as such an exploit against a popular decoder like libjpeg¹⁰ or GDI+¹¹ can mean the ability to compromise many applications.

If the image loader modules of popular programs like web browsers and chat applications could be easily isolated into a unique process, a compromise against an image loader would not give the attacker any useful privileges.

Figure 1.2. The multi-process, privilege separated model

MAC security must be integrated at the operating system kernel layer, as it relies on leveraging the hardware security features provided by most modern CPUs such as privileged instructions and an MMU. Because process separation as enabled by fast local RPC only makes sense when you can sandbox individual processes, the technique requires MAC security. Because of this, it will be assumed for the rest of this project that the RPC system is running on a Linux host with either the SELinux or the Novell AppArmor security system active. As currently only Fedora Core and Novell SUSE Linux have these systems integrated, they will be used for development. MAC security for Windows has recently become available through the Core Force project, and the impact of this will be discussed in the conclusions chapter.

Summary

The challenge of how to easily and cleanly separate sub-modules of monolithic software programs into separate processes is the focus of this project.

Something that is explicitly not a goal is analysis of how to modularize software in the first place. That is an active research area in the field of software engineering, and this project looks only at the technical mechanisms allowing process separation of existing modules. The goal is to confine software as much as possible, following "the principle of least privilege". This principle was first laid out by Salter and Schroeder [Saltzer75] in their paper "The protection of information in computer systems".



Saltzer and Schroeder's paper has been hugely influential on the computer security community over time and many key ideas in the security community can be traced back to it.

The method used is a fast remote procedure call transport - FastRPC - optimised for performing calls between processes running on the same machine (and therefore sharing the same physical memory chips). RPC is a well known technique in the software community, and discussion of it dates back to the early 70s [Xerox84].

Definitions

- **RPC: Remote Procedure Call.** This is a layer over a message based communications system that strives to make communication between programs transparent. Typically this is done by allowing the programmer to call a function on the client and have the function actually execute on the server.
- **Master:** the main process, which has the greatest privilege level of any component. This is typically the program that the user will run, for instance "Evolution Email" or "Gaim Instant Messenger".
- **Slave:** a process containing a separated sub-module. This process has a lower privilege level than the master process. For instance it may deal with image decoding, network protocol handling or executing untrusted code.
- **Marshalling:** This is the process of writing any information an RPC may require into a packet suitable for transmission to a remote host. For instance, function parameters must be sent, and if any of them are pointers, what they are pointing to must also be sent. Marshaling can be a very complex (and therefore expensive) process, especially once complex data structures such as doubly-linked lists must be marshaled.

- **Process:** A task within the system. Each process runs in its own address space, and so cannot access or interfere with another process. The kernel can dictate what a process can and cannot do.
- **Mandatory Access Control (MAC):** a security system in which the relations between each system object are controlled by a centralised policy – security based on what you are, not who you are.
- **Discretionary Access Control (DAC):** security based on user identity, this is the default model for Windows, MacOS X and most Linux distributions.
- **Principle of least privilege:** the idea, as pushed by the NSA, that software can be made more secure by ensuring that it has only the privileges it needs to do its job and no more.
- **IDL: Interface Definition Language.** Traditional RPC frameworks use these to define the RPCs available and provide marshalling hints to the RPC system, sometimes also to provide language neutrality.
- **Stack:** region of memory used by the computers CPU to track return addresses of nested function calls, and to hold temporary buffers and state.
- **Heap:** region of memory used to hold data not suitable for the stack (typically data that exists over the lifetime of a program)
- **Security context/profile:** a set of privileges that are given to any process running within the profile. Typically programs are not allowed to do anything not specified in their security profile.

Objectives

To write a remote procedure call (RPC) system that is:

- Fast enough to sit between internal components in C/C++ based software which exchange traffic regularly without causing user-visible slowdown.
- Easy enough for developers that separating their software into multiple processes is an attractive option.

- And by achieving the previous two goals, improve the security of desktop software and make it more resistant to attack from hackers, viruses and bots.

By separating software into multiple processes that only interact through carefully controlled gateways, it is possible to leverage mandatory access control (MAC) based security like the SELinux system to give each component least privilege.

These goals have the following basic, intermediate and advanced subgoals:

- *Basic:* Exchange pointer and primitive types as function arguments/return values with hard coded RPC wire-ids and dispatch tables.
- *Basic:* Allow for pointers to inside the stack.
- *Basic:* Allow for pointers to inside a shared heap.
- *Basic:* Demo of an application being hacked and once RPC protected, the hack failing.
- *Intermediate:* Address space management: ensure that the slave process only has the memory mappings it needs.
- *Intermediate:* Develop a re-usable framework for easy integration with other programs.
- *Advanced:* Stack barriers for additional security
- *Advanced:* Multi-threading support
- *Advanced:* Allowing for multiple slaves under the control of a single master

Criteria for success

The project shall be considered successful if it succeeds in producing a system that is faster and easier to integrate with existing codebases than MS-RPC, which is probably the most widely deployed RPC framework in the world, being as it is a part of every Microsoft Windows install.

To measure this, the speed of FastRPC will be compared to MS-RPC, and the size of the code impact will likewise be measured.

Example

The Gaim¹² instant messenger program is a potential entry point for malicious code. It requires a high level of privileges (for instance, to read/write from the file system and window server) and contains complex code which parses data from external sources like MSN/AIM/Yahoo Messenger servers. That is risky as by sending malformed data to the client from another computer, it would be possible to completely compromise the users computer by working up from the Gaim protocol plugin or image decoder to the rest of the users system.

By separating the at-risk parts of Gaim like the protocol plugins, crypto engines and image decoders (for buddy icons) from the non-risky parts like the user interface code, a compromise can be contained and damage controlled so reducing or eliminating the risk of the exploit being leverage to gain full control.

Implementation Plan

The system shall be known as FastRPC. By combining FastRPC with the SELinux MAC implementation, a proof of concept system will be implemented and a demonstration of a real exploit attempted being stopped by it will be shown. The patch against the case study application will be as small as possible. The RPC system itself is unlikely to bear much resemblance to traditional RPC, instead for speed and ease of use it's likely to rely on controlled shared memory segments. For instance, rather than rely on detailed type information to build skeletons and stubs, the contents of the stack and parts of the heap will be directly mapped into the server processes address space. The client is responsible for verifying the correct state of the data transferred from server to client and vice-versa. The framework will ensure that malicious stack and heap manipulations cannot occur, and the kernels security system will prevent a compromised component from doing things like accessing the filesystem.

Choice of software

Because mature MAC security frameworks are only available on Linux, either Fedora or SUSE Linux shall be used for development of the software.

The compiler used was assumed to be the GNU C Compiler. The code should be portable to other compilers with minimal effort but this has not been tested.

The system and its documentation, along with this report, was maintained using the Subversion¹³ version control system. As each feature or bugfix was made the changes were committed to the repository. Whilst using version control for a project with only one person working on it may seem wasteful, it allowed for a kind of "global undo" - back-tracking of changes that in hindsight turned out to be a mistake - along with precise identification of exactly what changes had been made.

Guide to contents

- *Chapter 1* - The basic theory, background and rationale
- *Chapter 2* - An overview of previously published literature, covering RPC optimisation and applications to security
- *Chapter 3* - Design overview, discussion of security sensitivity of different parts of a running program
- *Chapter 4* - Implementation and a discussion of implementation tradeoffs, issues arising and how it works
- *Chapter 5* - Results of a comparison of FastRPC against MS-RPC
- *Chapter 6* - Case study of integrating FastRPC with an image viewer program
- *Chapter 7* - Conclusions and directions for further work
- *Appendix: References* - Papers and documents cited throughout the report

Endnotes

¹Network IT Week, 12th November 2003: <http://www.networkitweek.co.uk/2123707>

²PhatBot trojan analysis, <http://www.lurhq.com/phatbot.html>

³Microsoft press release, REDMOND, Wash. - Aug. 9, 2005, <http://www.microsoft.com/presspass/press/2005/aug05/08-09MSRichterSettlementPR.msp>

⁴"Have hackers recruited your PC?", BBC News Thursday, 17 March, 2005, <http://news.bbc.co.uk/2/hi/technology/4354109.stm>

⁵<http://www.bell-labs.com/>

⁶<http://www.nsa.gov/selinux>

⁷<http://www.novell.com/products/apparmor/>

⁸<http://force.coresecurity.com/>

⁹<http://www.securitydocs.com/library/3222>

¹⁰<http://www.ijg.org/>

¹¹<http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx>

¹²<http://gaim.sourceforge.net>

¹³<http://subversion.tigris.org/>

Chapter 2. Fast RPC in the literature

Performance and ease of integration

The two most important aspects of an RPC system designed to split monolithic programs into multiple processes are performance and ease of integration.

Performance is essential because it is likely the software involved makes many cross-calls between the components (as RPC won't have been a part of its original design).

Ease of integration is essential because traditional RPC frameworks are very invasive, requiring language-specific tools and code generators to work. This has been a big barrier for past attempts to modularise software using RPC. Using a C/C++ object oriented framework like CORBA or DCOM often leads to huge complexity and difficult code modifications.

It is believed that by eliminating the marshalling overhead from each RPC, performance can be greatly increased over existing network transparent RPC frameworks like CORBA. By eliminating language/machine abstractions such as a platform-independent interface definition language (IDL), ease of integration can be increased so it becomes near painless to use the framework.

In this chapter, previous attempts at improving performance will be examined, and then the approach my system takes will be described.

Performance in the literature

Because of the importance of RPC to the software industry, much research has been done into ways of improving its performance. The Microsoft RPC implementation has introduced successive optimisations throughout the years, such as interpretivemarshallers which reduce the working set of proxy/stub code. Other RPC implementations have focused on optimising the marshalling process by simplifying proxy/stubs, or eliminating them altogether.

Generally, there are several discrete components of performing an RPC that can be optimised independently:

- Marshaling/Unmarshaling

- Proxy/stub code speed/size
- Latency at the transport layer (network card/cable, or IPC system)
- Context switch overhead, in the case of same-machine RPCs
- Contention for server side resources (typically RPCs will be serialised as they are received)
- Buffer copying as marshalled data is copied into and out of kernel space/network driver chipset buffers etc.

Some approaches such as RPC on the SHRIMP virtual memory architecture [Felten97] eliminate the concept of marshalling altogether by providing network transparent shared memory regions. SHRIMP will be covered in detail in a later section.

In Microsoft Windows NT prior to version 4, the "user" subsystem, formally known as the client/server runtime subsystem (CSRSS), handled graphics and the widget toolkit. In keeping with the NT microkernel design, it ran in a separate process and a highly optimised form of RPC [WinNT] called QuickLPC was used to allow applications to communicate with it.

Other common approaches are to employ traditional optimisation techniques on pre-existing RPC codebases. For instance, Microsoft found that a major overhead in its RPC implementation (MS-RPC is a derivative of DCE-RPC) was the code size of the generated proxy/stubs. Increased swapping slowed down RPCs significantly – if only one page fault occurred whilst executing proxy/stub code then the time taken to go to the hard disk and back would instantly eliminate any savings gained from other optimisations. For this reason code size is often targeted for reduction. Automatic program optimisation either at the compiler level or using code specialisation has also been effective [Muller98].

SHRIMP RPC

The SHRIMP network interface ¹, developed at Princeton, is a custom network card that plugs into a regular PCI bus. By sniffing memory read/writes as they travel across the bus, the card can make it appear that certain regions of memory are shared with different computers on the network. As a program writes to one shared memory area, the contents appear in the equivalent memory region remotely. In the SHRIMP architecture, processes can “export” a region of their address space. This makes it available to other processes on the network so they can “import” them. Exported regions have some basic access control on them. It supports both explicit and

automatic send/receive modes. In the automatic mode, every write to an imported/exported region is snooped, the written values are turned into packets and send across the network. This mode optimises for low latency at the expense of network traffic.

By providing an easy to use, low latency transport the Princeton team were able to significantly improve the performance of RPCs on the same hardware. An implementation of SunRPC (similar in design to DCE RPC) based on this new transport was found to be several times faster than the traditional equivalent based on Berkely sockets. An implementation of a new RPC system that did not conform to any standards, called ShrimpRPC, was found to be much faster still: a null RPC (with no arguments) took only a microsecond more than the hardware minimum.

QuickLPC

QuickLPC was designed only for inter-process communications, and was virtually the most optimised form of RPC you can get. Arguments were copied into a shared memory buffer, and then a special event pair object was used to force a context switch from the client to the server. The event pairs optimised out the bulk of the context switch overhead, which arises from the server not being scheduled immediately after the client relinquishes control of the CPU. The shared memory eliminates marshalling overhead. Finally, the CSRSS process eliminated contention by starting a new thread in the server process for every connected client thread.

Event pairs are an interesting concept that could definitely be applied in this project. However, their use must be carefully considered. For one, it is not at all clear that with modern Linux schedulers it would make any difference – generally processes communicating via pipes get an “interactivity bonus” which means that if they write to a pipe then block waiting for a reply, the process listening on the other end will be immediately scheduled. Event pairs have another problem – by allowing you to force a context switch it opens up a security hole as any application can now take 100% cpu time by constantly switching between two co-operating processes.

Many of the concepts such as reducing or eliminating copying are common themes throughout all fast RPC systems. The overhead of copying memory between buffers can be significant, especially in low memory situations as is common on modern desktop computers. For inter-process RPCs shared memory segments are an easy way to avoid the userspace->kernelspace->userspace copies though synchronization is still necessary. Also the marshalling code must be specifically written to marshal into pre-allocated buffers and avoid absolute pointers within that space, otherwise you just replace a copy into the kernel with a copy between two regions of user address

space. These issues often complicate implementing shared-memory solutions on top of pre-existing RPC frameworks like SunRPC or CORBA.

Unfortunately the QuickLPC subsystem as implemented in older versions of Windows NT is unsuitable for splitting monolithic programs into smaller processes as it relies upon holding the event pair handle in a fixed location in the thread environment block (TEB)². Because of this design decision, it is impossible to do RPCs to more than one destination from the same thread. Worse, in most cases the single slot was already taken by calling out to the CSRSS. QuickLPC also failed to meet the other goal of this project, which is that the system should be easy to use and integrate with existing codebases. QuickLPC did not have any form of IDL compiler, nor did it provide any support for writing the marshalling code by hand. Only a few raw (internal) APIs were exposed.

LRPC

LRPC was developed at the University of Washington in response to the desire of the OS research community for faster micro-kernel designs.³ From the opening of the paper:

LRPC is perhaps the closest equivalent FastRPC around. The nature of a micro-kernel based system requires that many RPCs are made between components running on the same system, so fast RPC is essential for such systems. To provide this LRPC used variants of the same techniques used for FastRPC although it did not focus on ease of integration, and indeed is written mostly in Modula2 making it unsuitable for integration with most existing codebases.

Generic optimisation

A common strategy, probably because it is easy, is to apply generic optimisation techniques to pre-existing RPC codebases. When analyzing performance bottlenecks in early versions of Microsoft Office, the developers found that the RPCs between Word and Excel in the case of an embedded spreadsheet in a word processor document were harming performance. The code itself wasn't slow, but the sheer size of the generated proxy/stub code was increasing the suites working set (the set of pages that are held in RAM) dramatically.

The solution was the development of a “format string interpreter”. This is a program that accepts sequences of numbers representing a program for a small virtual machine. Because they aren't generic CPU opcodes, they take up far less space. Whilst there is the added overhead of the RPC

interpreter, because this is shared code that can be used by all processes the overall memory usage drops.



The format strings are generated automatically by the Microsoft IDL (MIDL) interface compiler ⁴, replacing automatically generated calls to the marshalling APIs. They can also be introspected by programs such as Microsoft Transaction Server to gain basic information about the objects being marshalled, although the value of this feature is somewhat dubious seeing as the IDL compiler can generate type libraries at the same time as generating marshalling code.

Endnotes

¹www.cs.princeton.edu/shrimp/Srpc/

²<http://www.windowstlibrary.com/Content/356/08/5.html#5>

³ A micro-kernel is an operating system design in which the kernel is shrunk to the smallest possible size by migrating code into userspace *servers*. For instance network stacks, filing systems and other modules that would have traditionally been in-kernel are run as programs communicated with via local RPCs.

⁴http://msdn.microsoft.com/library/en-us/midl/midl/_oi.asp?frame=true

Chapter 3. New Approach

A new RPC design

Because I am building a new RPC framework, instead of simply optimising a pre-existing one, it is possible to use a non-traditional design. The SHRIMP RPC framework is closest to what is needed performance-wise. SHRIMP is designed for network transparency and uses custom hardware however, which isn't a requirement of my system. The basic concept – shared memory instead of complex marshalling/message-passing – is however a good idea and the right direction to take.

I hypothesise that by eliminating the marshalling overhead from each RPC, performance can be greatly increased over existing network transparent RPC frameworks like CORBA. To do this elimination two techniques will be used:

1. Providing a shared heap facility so raw data structures can be allocated in such a way that they don't need to be copied for each RPC.
2. Blindly copying the contents of the stack directly on top of the other processes stack, such that function arguments and any on-stack structures which are pointed to are made available.

By operating at a much lower level than most RPC systems, complexity/size of the framework is reduced (hopefully increasing its speed) and perhaps more importantly integration is made easier. Because the framework just operates in terms of regions of memory, type information is not required.

For instance, in DCOM not only is it necessary to provide a separate type description of each interface involved separate from the code, it is also necessary to tag each object method with the [in], [out] or [inout] attributes [Box98] so the RPC library knows whether the method will be reading/writing to the given buffer. It needs that information both for memory management and so it knows whether to copy a buffer into the marshalled packet or not.

By simply joining processes together at the points they need, all this complexity can be removed. Pointers are implicitly assumed to be [inout], and the framework doesn't need to know the memory management or typecasting semantics for each function argument. This makes the programmers life easier – redundant keypresses are removed, the many obscure rules that DCOM

and similar systems impose disappear, and so the barrier to entry becomes correspondingly lower.

There are some subtleties involved with an RPC framework operating in this way. The remainder of this section will describe a few of them.

Resource sharing

There are several different types of resources that the developer may wish to share across process boundaries (and therefore security contexts). The frameworks job is not to share all of them automatically, but rather to make it simple to share only what is necessary. Share too much and security could be compromised, share too little and code will stop working.

Examples of resources which might need to be shared across RPCs:

- Memory regions
- Open files (file descriptors on UNIX)
- Security permissions (for this project, clearly we don't want to share these, but some frameworks like DCOM allow for this)
- Resources in another process, like an X server¹

This section will focus purely on memory. Sharing open files (which on Linux also represent connections to other processes like the X server) can be done using UNIX-specific facilities.

There are at least three critical regions of memory for any program: the program code itself, the heap and the stack. Multi-threaded programs will have more than one stack and most dynamically linked programs will have multiple regions of program code. Most programs will only have one heap, though some operating systems like Windows provide explicit support for multiple heaps and nothing on any OS stops you from managing your own heap (garbage collectors do this).

All of these memory regions are security sensitive. Program code generally can't be modified in a locked-down OS like Linux. So sharing program code is usually safe. The main reason to not share a piece of program code is because some exploits hackers use rely on finding particular code sequences in memory – if that sequence doesn't exist, they can't work. Therefore keeping the minimum amount of code in memory at all times is a good idea, as it minimizes the risk of exploitable code.

The heap is security sensitive for two reasons. Firstly, vital control information may be stored in memory – for instance, imagine a program that either deleted all the users files or made a noise depending on an option selected by the user. The selection the user made might be stored in the heap, so allowing indiscriminate access to this data could compromise the safety of the users data in the case of an exploited submodule. The other reason the heap is security sensitive is that it is possible to interfere with other pieces of code by overflowing it (by overwriting memory allocation control structures).

The stack is one of the most sensitive regions there is. Many exploits are based on buffer overflows, in which a buffer allocated on the stack is written to without checking the size of what's being written. The result is a "buffer overflow" or "stack smash" which can cause execution to jump to arbitrary code inserted into the program, or at the very minimum it can trigger execution of some function inside the program.

So, one of the major challenges of FastRPC is how to design a system that precisely and easily restricts access to the memory regions of the main "master" process.

Address space management

Most modern operating systems provide mechanisms to share memory regions between processes and they are quite straightforward. The hard part is controlling where the shared memory regions are mapped in the master and slave address spaces. Every process running on a modern desktop/server CPU has its own virtualized view of memory. This view makes it look as if the process is alone on the computer, even though in actual main memory chips the memory used by every process is interleaved.



This memory virtualization is achieved through the use of a hardware memory mapping unit (MMU). The MMU translates addresses from each processes address space to the hardware address space.

Each processes address space may contain several different types of memory. There is standard working memory such as the heap and stack, which is private to the program and stores user data, control information, temporary buffers and so on. Then there is code. Normally this is mapped from a disk file straight into memory, meaning that it will be loaded into and out of main memory automatically by the kernel. It is also read-only by default to add extra protection against programmer error, that is, attempts by a program to modify its own code will cause a

crash. To be able to do this the program must ask the kernel to adjust the permission settings in the MMU chip on the programs behalf, and as such self-modification can become a privilege like any other.

Typically, the exact location of these zones within the address space isn't guaranteed by the operating system. In Windows, the regions usually stay in the same place unless for some reason there would be an overlap. On Linux, a system called "ExecShield" randomizes the address space for additional security against buffer overflows.²

Because of this address space randomization, special care must be taken to synchronise the address space of master and slave. In traditional RPC systems, full type information is available so as pointers are demarshalled they are reset around wherever the heap happens to be in the client address space. Because FastRPC doesn't involve type information, this cannot be done – the only solution is to ensure that data exists at the same location in both processes.

This is particularly of importance for shared heaps and thread stacks. When a new thread is created, so is a new stack for it to run on. The stack needs to exist both in the master and slave – if they aren't the same size and location, RPCs will crash as the return addresses will be incorrect. If pointers are passed across an RPC, it is vital that they point to the same data in both processes, meaning that the heaps must also be in the same location.

There are several different ways to ensure the address space is controlled. On most operating systems the address space layout is not guaranteed even if it is predictable – writing code that assumes a particular layout is remarkably bad form and can lead to support nightmares for future developers. Therefore it is vital to somehow impose the required layout upon the slave process. Unfortunately, most operating systems don't provide any help with this, it must be done manually.

The easiest method is to use the UNIX fork() system call to produce a clone of the masters address space during the setup process. Because this is so easy, fork() calls are used in FastRPC. It suffers one huge disadvantage however – once the fork has taken place, the slaves address space doesn't follow changes in the parents. On one hand, this is correct and desirable as obviously a full mirroring would negate any security benefits FastRPC might have. On the other hand, it makes future support of multi-threading awkward as any threads created in the master process won't have any equivalent in the slave. It also means that as shared heaps are constructed and torn down, the likelihood of collisions increases dramatically.

Forking has another disadvantage – you must manually clean up the slaves address space before the slave code gets control, as otherwise the slave may be able to read information it shouldn't have access to (unencrypted passwords and so on).

A more advanced method is to start a new process with an empty address space, then build up the shared mappings one at a time. This avoids the need to "clean" the address space by unmaping unneeded regions, a process which risks missing parts, but is complicated by the behavior of the operating systems dynamic linker. On Linux and Windows, there's no way to ask the OS to load a shared library at a given address, instead, the best location is chosen by the linker itself. If you're trying to accurately reconstruct the address space of another process, this means the only option is to do dynamic linking manually: a time consuming and complex process.

Although not actually implemented due to time constraints, a compromise approach was designed. In this strategy, once the master has initialized itself and is ready to use the potentially insecure submodule a fork takes place. After the slave has been initialized, the RPC framework in the master hooks the `mmap()` and `munmap()` calls so it can observe changes to the address space layout. As each change takes place, the modification is sent to the slave which mimics the same action so it keeps the layout compatible. The slave doesn't have to copy the master exactly: for instance, the master may map a file into a particular location. Unless specifically requested though, mapping this into the slave as well could be an information leak so the slave would simply "block off" that VMA region with an empty mapping. This balances the books so if the slave loads something into its own address space, there's no chance of it conflicting with anything loaded into the master, but doesn't cause information leakage. Likewise, if the slave tries to map something into its address space (for instance, loading a file, a shared library etc) then it communicates with the master to try and locate a space that's free there too.

Precise MMU based access control

The easy integration of co-operating modules FastRPC provides is its biggest strength, but also its biggest weakness. Just because you *can* share something, doesn't mean you *should*.

Sharing too much information could lead to a compromised slave getting access to sensitive data like passwords or user details, or worse, being able to interfere with the internal state of the master. Sharing too little (or making that sharing too difficult) risks alienating the developers who want an easy way to make their software secure. Getting this balance right is key to the success of any system designed to split software into privilege-managed modules.

In general, it is not possible to automatically determine what data should be shared and what shouldn't: this is a task for the human programmer. The process of making software more secure by separating it into modules may even involve rewriting parts of the program to reduce the interconnections between those modules, just as the deployment of SELinux caused fixes to software that requested more permissions than it really needed.

Therefore, the task becomes making it easy for the human programmer to tell the FastRPC system what data should be shared and what should not be. By default, we should ideally assume that nothing is to be shared: starting from nothing and granting only the privileges that are needed is the core tenet of the principle of least privilege. By requiring the developer to explicitly grant access to what the submodule needs possible mistakes and oversights are avoided.

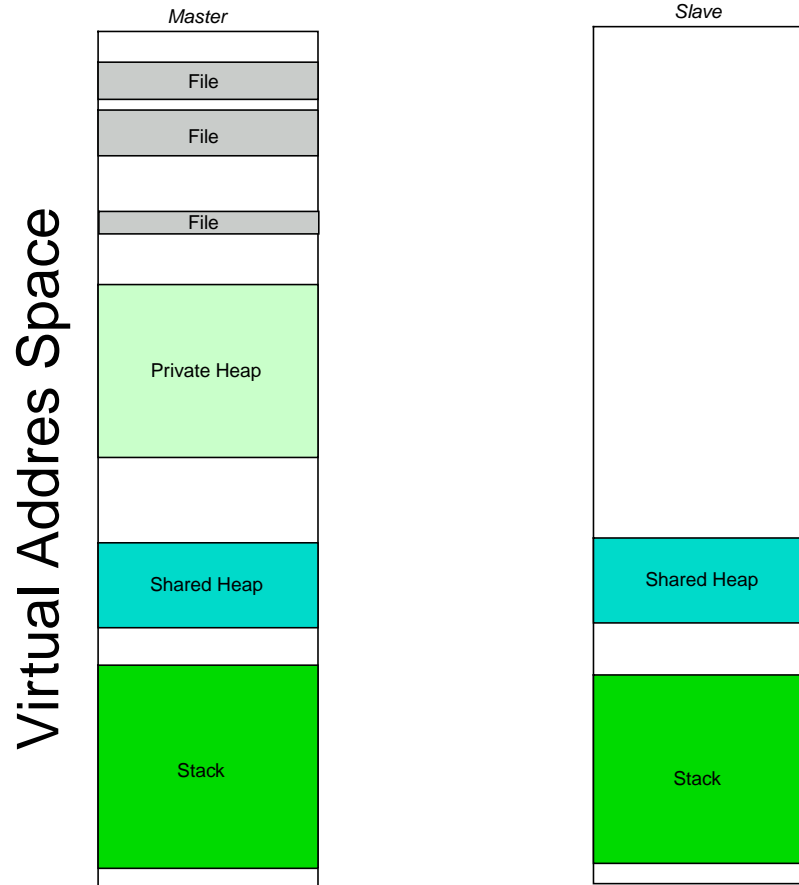
There are four different types of memory mapping active in any given process:

1. Loaded (executable) program code, from the main binary or dependent shared libraries
2. Heaps
3. Stacks
4. Non-executable file mappings, for instance, for translation databases.

There are also custom mappings which aren't clearly in any of these categories and which may be used for any purpose, but these will be ignored as they don't require special treatment.

These four types of memory mapping can be abused by malicious code in different ways, and all require different types of management as a result.

Figure 3.1. The address space of the less privileged code starts as a subset of the more privileged code



- *Executable code* can be exploited by “return-to-libc” type buffer overflow exploits. In this type of hack, it is not possible to inject arbitrary code into the running process but it is possible to control where the function returns to. If there is a sequence of bytes that correspond to what you want to do somewhere in memory, that can be leveraged to achieve the intended effect.

- *Heaps* are used to store the working state of a program. Typically, whilst many programs store sensitive information in obscured form on disk the same precautions aren't taken for data stored in memory³. The reason is partly that it is a lot easier to access data stored on disk so more care is taken to protect it, and partly because to operate on data with the CPU at some point it has to be decrypted anyway.
- *Stacks* contain small, temporary allocations and control information for a thread of execution. There can be multiple stacks inside a program. Being able to access the stack means you can control the flow of that thread, as well as accessing potentially sensitive information.
- *File mappings* contain arbitrary data backed by a file on disk. They are a more convenient way of reading the contents of a file, and so only files that the slave has been given access to should survive.

Usually by the time an application has started all the executable code that it needs has been loaded into memory by the operating systems dynamic linker. There are some notable exceptions to this rule - some very large C++ programs such as Firefox, OpenOffice and Microsoft Office make extensive usage of "lazy loading", in which code is constantly loaded and unloaded from memory under the direct supervision of the program. For the first version of FastRPC this factor is discounted, as many applications don't rely on it, and tracking the load/unload sequences to keep the address space synchronised is out of the scope of this project.

The heap

The heap and the stack require different treatment. The heap is a global pool of many small allocations, and the master (privileged) process will need to control access at the level of the individual allocation for best ease of use and security. To solve this problem, a *shared heap* will be provided. The shared heap is a region of memory that is mapped into both processes at exactly the same location in the address space (so ensuring pointers remain valid). It is separate to the standard, operating system provided main heap, but the two can be used interchangeably by the application. The FastRPC framework provides equivalents to common heap-using C functions like `malloc`, `free`, `strdup` and so on, which operate in the same way as the standard equivalents but fetch blocks of memory from the shared heap instead of the default application-private heap.

On Windows, a set of APIs are provided to allow for multiple heaps. On Linux no such API exists so FastRPC must supply its own simple multi-heap implementation. This implementation

is mostly transparent to the developer: it is initialized at the same time as the rest of the RPC framework and whilst the framework allows for multiple shared heaps, most people will need only one. Multiple heaps can prove more efficient in cases where the whole heap can be de-allocated in one go. When first created the shared heap is empty. Data that the slave process needs to access can be moved into it using the shared versions of the regular C library, so ensuring strict separation.

The stack

The stack is by default completely shared. Although this is a violation of the principle of least privilege, it is more convenient however and less likely to break software in subtle ways. Consider the example of a function that accepts a pointer, and which calls an RPC proxied function into the slave with that pointer:

```
bool function_a(char *str)
{
    if (strlen(str) > 3) return slave_function(str);
    return NULL;
}
```

If the stack frames above the `slave_function` call are not shared at all then this function will fail. But it is not even safe to simply share the stack up to the arguments itself, because "str" may in turn be a pointer onto the stack. Code can be made robust against such problems by duplicating the memory into the shared heap before calling into the slave.

To provide finer grained security, one of the intermediate goals was to allow the developer to set up a "stack barrier". A stack barrier prevents the slave process from reading or writing beyond that point, and FastRPC provides a simple API that can be used to set this.

The stack is vulnerable for another reason - return addresses are stored here. By modifying the return address a compromised slave could force the master to jump back into the stack and so gain privileges. This attack can be defeated by unwinding the stack to discover the locations of the return addresses and discarding modifications made by the slave to those locations.

To allow for this, the stack is not actually shared at the VM level, instead on each RPC the region below the barrier is written to the socket along with the RPC control data. The FastRPC framework on the slave side then reads the data off the socket and writes it to the correct location in memory. The net result is that the stack *appears* to be automatically shared, even though the sharing process is explicitly controlled by the framework (see the diagram below).

Designed for performance

One reason RPC for security has historically been considered infeasible is due to the overhead of marshalling calls between processes. As computers have got faster this has become less of a concern, still, RPC is often associated with "slow" by developers and it is easy to imagine scenarios in which the RPC overhead becomes significant. Because of this, it is important for FastRPC to emphasise speed.

The bulk of the overhead in any RPC system is made up of the marshalling and the context switch. The marshalling is entirely under the control of the framework, and this is where FastRPC is faster than the alternatives. The context switch overhead is made up of two subcomponents: the time taken by the kernel to actually perform the switch, and the "gap" time until the kernel decides to schedule the other side of the RPC for execution. Linux is very fast at context switching, but there is no way to force a process to be scheduled right after your own timeslice is up. To allow this would be a security hole as two co-operating processes could use all the CPU time, freezing the computer.

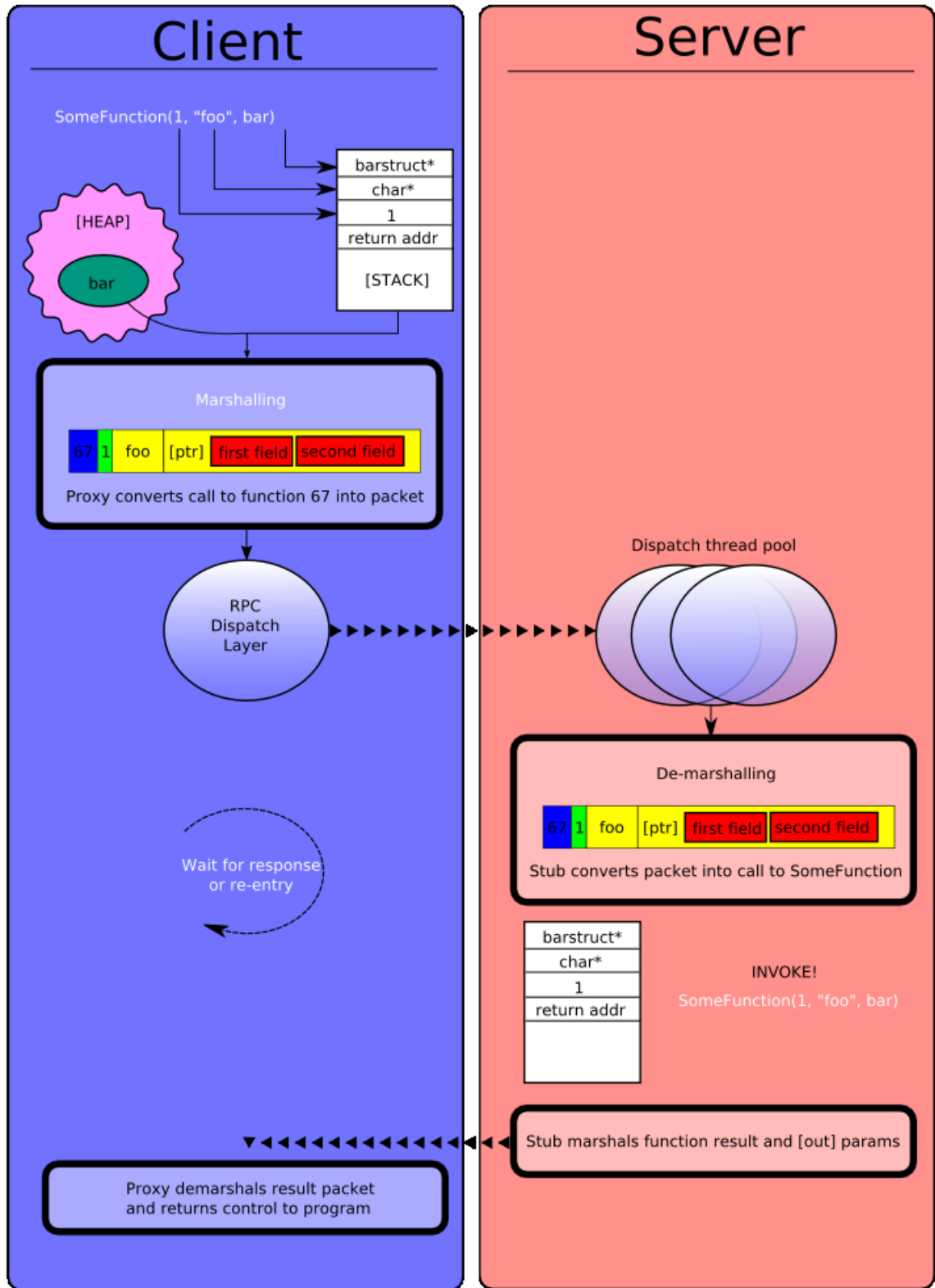
It may be possible to implement an event-pair primitive inside the kernel itself, similar to the one used by Microsofts QuickLPC in the early 90s. The semantics of this are that triggering the event pair causes an instant context switch to another process but maintains the current scheduler timeslice. If the RPC and context switch can comfortably fit inside a single timeslice then this optimisation would be worth doing, and it'd mean an RPC consists of not that many more instructions than a regular function call. This technique is not explored in this paper but is an avenue for further work.

Overheads in traditional RPC frameworks

Marshalling - the process of packing a function call into a packet - is expensive relative to the cost of a standard function call, which is only a few instructions.

It is expensive because it involves running a large amount of code, which dirties the processors instruction cache, and because that code is often complex - in the case of DCOM or CORBA it will involve allocating memory, looking up data tables, and manually walking the data structure graphs to pack them into a serialized form.

Figure 3.2. RPC in a DCOM-style framework

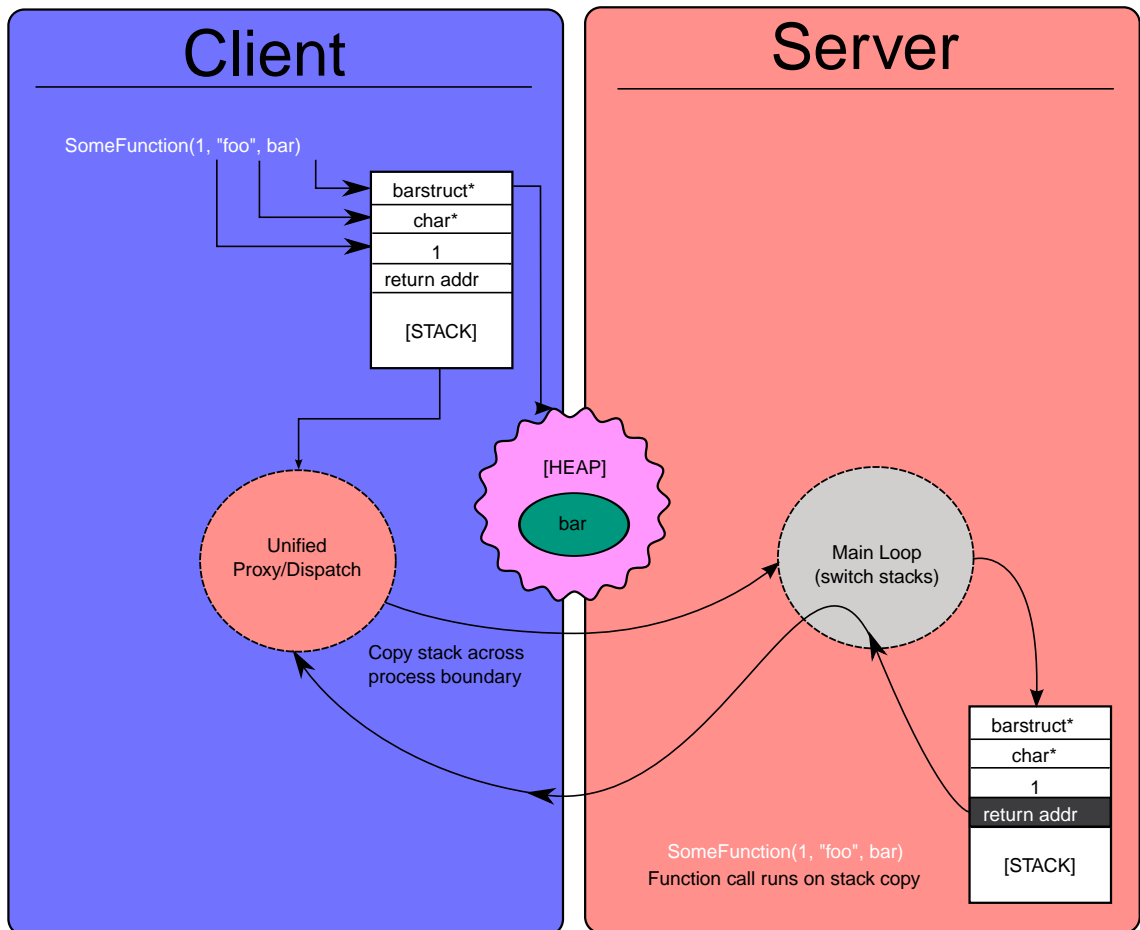


As can be seen from the diagram, each RPC involves at least 4 [de]marshalling operations: marshalling the call, demarshalling the call, marshalling the result, demarshalling the result. The arrows in the middle of the diagram represent the transmission of the packet from client to server: for the purposes of security this will always be across processes on the same machine, however, most RPC systems are network transparent and the packet can also go across network/machine and sometimes even language boundaries.

Overheads in FastRPC

FastRPC is comparatively simple:

Figure 3.3. RPC in FastRPC



The marshalling operation disappears in favour of simply writing the stack across the socket. The heap is shared, so no knowledge of which stack entries are pointers is needed. The arrow

on the right indicates the flow of control: when the function is finished executing it jumps back into the dispatcher which then captures the return value and writes it back across the socket.

As a result, less code is involved, less data is transferred, and so speed ups should be seen.

Designed for ease of integration

Apart from performance, another issue that has prevented widespread integration of RPC into existing programs is the difficulty of integrating them. Most RPC frameworks like DCOM or CORBA are designed to be language and even network transparent, and as such require you to mould the program around the framework and what it allows.

Sometimes, as is the case with SOAP⁴ or DBUS⁵, the RPC type system cannot represent everything in C so the program must be simplified to compensate. In most cases the interfaces that were already specified in code must be *respecified* in a framework specific "interface definition language" (IDL). Then a framework-specific compiler must be integrated with the build system to generate the proxy/stub/skeleton code which must then be linked into the program. And after that, usually the components must be manually registered with the system and activated via a framework-specific activation API.

So these technologies require a lot of work even for simple use cases. One of the driving forces behind FastRPCs design was to use minimal code for the common case of C/C++ programs being protected. As such, the following design decisions were made:

- *No IDL* - IDL repeats a story already told in code. As FastRPC is designed primarily for C and C++ programs language independence is not a requirement, and so there's no need to create a new type system. As it doesn't attempt to marshal data into packets, it doesn't need information about pointer locations nor memory management metadata.
- *No activation framework* - FastRPC is not a component model nor a document embedding system. It focuses on one thing, and one thing only - getting a function call from one place to another as quickly and easily as possible. As such forking, establishing the connection, making the server/slave listen and so on are all done automatically. If the user wishes to customize this behaviour they are simply handed a socket and given responsibility to join the two sides together themselves. As a result they can choose between easy and automatic, or precise and manual. But they are never forced to write lots of code they don't really need to.

- *New terminology* - Instead of a specific client/server model, FastRPC uses the concepts of "master" and "slave" to reflect the separate privilege levels they have. Another reason for the new terminology is that if client and server were used then a strange situation would occur, in that the client would create and be in charge of the server. That's not what you would intuitively expect, so to avoid confusion different terms are used.
- *No separate code generation* - code generators such as the DCOM/CORBA/SOAP IDL compilers are hard to integrate with pre-existing build systems and can be fiddly to set up properly. Some people dislike code generators entirely as the generated code can be hard to maintain and debug. FastRPC uses only a handful of C preprocessor macros to generate its code.

Together these design decisions help create clean, simple code that can be easily understood by developers. The cost is that FastRPC is not as powerful or flexible as more traditional frameworks - however, its goal does not require that flexibility.

Endnotes

¹On UNIX systems like Linux, an X server provides clients with access to the graphics display and a basic windowing system.

²Red Hat Magazine, July 2005: <http://www.redhat.com/magazine/009jul05/features/execshield/#threat>

³Firefox is an example of that

⁴The W3C standard for web services RPC

⁵DBUS is an RPC framework used to co-ordinate elements of the Linux desktop code

Chapter 4. Implementation

There are four key subsystems in FastRPC:

1. Developer API
2. Master-side Engine
3. Slave-side Engine
4. Shared heap subsystem

The core RPC library is split into 3 parts, and each module exposes a small bit of API to the developer. The master/slave distinction refers to the most and least privileged part of the code respectively, and the code that interacts with the RPC framework from each side is different. Calls flow in one direction, from the master to the slave. In future this could be enhanced to provide support for re-entrancy and callbacks, this is given further discussion in the final chapter.

Developer API

The developer API consists of the code that developers interact with to integrate the system and their software. It consists of a handful of functions and some macros.

There are three steps involved with setting up FastRPC and a program.

1. Initialize the RPC library
2. Declare the slave-side function array
3. Declare the master-side proxies

Basic steps

The following code snippet shows how to initialize the library from C:

```
#include <rpc-engine.h>
```

```
extern rpc_call dispatch_table[];
int main(int argc, char **argv)
{
    if (rpc_setup(dispatch_table) < 0)
    {
        fprintf(stderr, "pngshow: could not initialize FastRPC\n");
        return -1;
    }
    .....
}
```

It is typical but not required to separate the code that will be in the highly privileged master side from the low-privileged slave side, and to put them in separate C files. That way the compiler will separate the namespaces when slave functions are marked as static, so there's no need to do it manually. The `dispatch_table[]` array is declared in the C file containing the low-privilege slave code:

```
static int my_low_privilege_function(char *str)
{
    /* do something with str */
}

static void some_other_function()
{
    ....
}

rpc_call dispatch_table[] = {
    (rpc_call) &my_low_privilege_function,
    (rpc_call) &some_other_function,
    NULL
};
```

This is an array of function pointers for each function that is callable via RPC.

An RPC is invoked by calling a normal function that is supplied by the framework. To create one, the developer must use the `RPC_PROXY` macro with the name of the function and the function id as parameters. The ID is a 1-based index into the `dispatch_table` array. In a future version of the framework it might be possible to hide the function IDs entirely.

```
#include <rpc-engine.h>
#include "my-program-internals.h"

extern rpc_call dispatch_table[];

RPC_PROXY("my_low_privilege_function", 1);

int main(int argc, char **argv)
{
    if (rpc_setup(dispatch_table) < 0) ...

    int result = my_low_privilege_function("Hello World");
    printf("result is %d\n", result);
}
```

This code is all that is required to use the RPC framework in a minimal fashion. The `rpc_setup` function will initialize the RPC library and use the UNIX fork system call to split execution into two paths. One will return to the main program, the other will enter into a loop waiting for RPCs to the slave. Immediately after the fork, the slave will drop its privileges using the Novell AppArmor `change_hat` API. The exact privileges it has can be defined by an external policy file loaded by the system administrator on an application-by-application basis.

Currently the security framework and profile in use is not configurable. Making it configurable would not be difficult and would be a useful extension of this work in future.

Memory management

Memory management must be carefully considered. You can pass pointers to values on the stack, but pointers elsewhere must be pointers into a shared heap. Multiple shared heaps are possible but one is created at initialization time and can be accessed through some convenience APIs:

```
// sharing strings is easy ...
char *shared_str = rpc_strdup(private_str);
low_privilege_operation(shared_str);
rpc_free(shared_str);

// generic malloc is available too
int *number_array = rpc_malloc(sizeof(int) * 10);
int stack_array[] = { 1, 2, 3, 0 };

low_privilege_operation(number_array, &stack_array);
```

The `rpc_barrier` function can be used to set the upper limit on the part of the stack that is marshalled. This is important to prevent potentially untrusted code from interfering with control data stored in the stack further up. For instance:

```
bool start_server = FALSE;

.....

call_untrusted_function();
// this could have set start_server to TRUE, with
// potentially catastrophic consequences.
```

The solution to this is straightforward and only requires a marker value on the stack:

```
bool start_server = FALSE;

.....

int stackmarker;
rpc_barrier(&stackmarker);

call_untrusted_function();
// This function can no longer read or write to
// start_server. Read attempts will yield the
// value zero, write attempts will be allowed but
// ignored.
```

It may be that you wish to allow a slave to access some data that is not allocated by yourself. For instance, you may wish to share structures managed by a shared library - the case study demo requires this as it decodes an image in the slave, but the master needs to use the decoded image which was allocated indirectly by the PNG decoder library. For this reason the API exposes the `rpc_redirect_to_shared_heap` variable. When set to `TRUE` FastRPC will override the systems `malloc` implementation and force all allocations onto the shared heap.

Miscellaneous APIs

Finally, sending file descriptors representing open files or sockets across the connection can be done by using `rpc_fd` like this:

```
int my_file = open("/some/file/slave/has/no/access/to.txt");
untrusted_file_opener(rpc_fd(my_file));
close(my_file);
```

This allows the master to control which files the slave has access to dynamically, as a file descriptor can be used to read that file but not access any others.

A common result of a failed exploit attempt is that the program crashes. By default, if the slave crashes, FastRPC will also terminate the master as this is the safest and simplest thing to do. For additional robustness the user can override this behaviour. The case study image viewer demo does this to present the end-user with nice "no go" artwork instead of simply terminating. To use this facility, the `rpc_set_crash_handler` call is used. Two handlers are provided for convenience - the default handler that exits the program, and another which simply causes the crashed function to return zero. Example:

```
rpc_set_crash_behaviour(rpc_crash_handler_return_zero);

int result = call_untrusted_function();
if (result == 0)
{
    // the call failed, this must mean it crashed
}
```

Master-side engine

The master side of the engine is responsible for creating the slave process, altering its privilege level and setting up the connection. It also relays calls to the slave and returns the response.

Setting up a connection proceeds as follows:

1. A local socket pair is created using the UNIX `socketpair` syscall. This operates like a network socket but is not available to other machines or processes.
2. Some internal variables are initialized.
3. The process forks into master and slave.
4. The master side allocates a shared heap and transmits it to the slave, then returns to the user program.

5. The slave side lowers its privilege level, receives the shared heap, switches to a new stack and enters the main loop.

Control is transferred from the program to the RPC framework via the proxies - small "imposter" functions written in assembly code that stand in for the real thing. These proxies, generated by the `RPC_PROXY` macro, store the current position of the stack pointer then jump to the generic `write_command` function.

The non-portable assembly is required because the act of calling a C function modifies the stack frame, and it's important to know the exact position of the stack at the moment the function was called, before the RPC framework gets involved and modifies it. If the exact stack position was not known, then the call could not be transparently resumed in the slave process. Getting this address is hard to achieve with pure C, hence the use of assembly code. Whilst it would be possible to duplicate in C, the resulting code would be no more portable than the assembly, but would simply *look* more portable on first inspection. The use of assembly also has the advantage of bypassing the compilers type checking - useful for reducing the amount of typing involved.

Slave-side engine

The slaves *main loop* looks like this:

1. Read a request from the socket (and wait for one if nothing has arrived)
2. If it's an internal "control" message then handle it accordingly and go to step 1, otherwise, proceed
3. The stack copy is read from the socket and written over the old stack location.
4. If part of the stack is blocked using a stack barrier, then the rest of it is set to zero ensuring the stack is of the right size and alignment.
5. A stack switch occurs and control is passed to the called function. The called function returns, triggering another stack switch *back* to the main loop stack.
6. The result is written to the socket, along with the new contents of the RPC stack.
7. Any file descriptors that were transmitted along with the RPC are automatically closed. This prevents leaks due to the duplication inherent in the `rpc_fd` call.

Control messages are used to communicate between the master and the slave, and have a function ID of zero. There are three control messages: stop, receive file descriptor and receive shared heap. They are called transparently by the framework and developers do not need to know about them.

The primary complexity in this procedure is the stack¹ switching process. As a stack is merely a region in memory pointed to by some special registers nothing stops a program having multiple stacks, and this is standard for multi-threaded programs where each thread gets a private stack (as is required by the definition of a thread - a separate flow of control through the program).

FastRPC uses multiple stacks without multiple threads because there are effectively always at least two flows of control - the RPC flow, which is the one being split across processes, and the RPC control/dispatch flow, which is where the engine is receiving and decoding requests. They must be kept separate, but don't need to run simultaneously.

Switching stacks on Linux/x86 is not a complicated operation; the basic outline looks like this:

```
static __attribute__((used)) void *new_stack(int stacksize)
{
    // parameter 1: don't care where the new stack is
    // parameter 2: how big it should be
    // parameter 3: read/write/not-executable permissions
    // parameter 4: flags that tell the kernel it is a stack
    // last arguments: unused for this kind of allocation

    // now allocate the memory for the new stack
    void *s = mmap(NULL, stacksize, PROT_READ|PROT_WRITE,
                  MAP_PRIVATE | MAP_GROWSDOWN | MAP_ANONYMOUS, 0, 0);

    // it may fail ...
    if (s == MAP_FAILED) print_and_quit("stack alloc failed");

    return s;
}

.....
```

```
asm( "    push $0x200000\n" // 2 megabyte stack
     "    call new_stack\n" // allocate it
     "    mov %eax, %esp\n"
     "    xor %ebp, %ebp\n" // set the stack registers
     "    call initial_function\n"); // call first function
```

The `__attribute__((used))` marker on the `new_stack` function prevents the compiler from optimizing it out.

Testing scheme

Testing was done using both unit tests and integration tests:

1. As each feature was completed, a unit test was added to the test suite inside the source code. The test suite checks each feature works correctly, and just as importantly, fails correctly. It uses both internal assertion checks, and prints output to the console which can be "eyeballed" to confirm that output is as expected. This is made easier by the test suite printing what the expected output should be before the actual output.
2. An integrated test was written, which demonstrates a simple "real world" application that loads an image from disk and draws it to the screen. The program uses an old version of the libpng image decoder library which contained a buffer overflow exploit (since fixed). A working exploit for this flaw was found on the internet and integrated into the test, such that a malicious image file was created.

The exploit opens a network socket and connects it to a shell, so that the computer can be remotely controlled using the telnet program. The demo shows how the software is split into the privileged master, which is responsible for allowing the user to select an image file and drawing to the screen, and an unprivileged slave which simply decodes the image into an uncompressed buffer - a raw data processing task that requires no privileges.

By enabling or disabling the security system, you can therefore see the attack working or being blocked.

This test might seem artificial. At first sight it may appear that the whole program could have been blocked from establishing network connections, and so FastRPC is not required to secure this program against attack. However, the following factors show that it *is* important to lock down software at a fine level of granularity:

- a. The exploit may not have required network access, but may have been designed to load files from disk and do things with them - privileges the program itself needs to do its job.
- b. An easy improvement that could be made to the image viewer program is to allow it to display images residing on other computers; this feature would require network access.

Endnotes

¹The stack is an internal structure used by the CPU and compiler to track program control flow and store temporary state.

Chapter 5. Results and Evaluation

Because there are two design priorities for FastRPC - performance and ease of integration - it was evaluated based on those criteria. To gain objective measurements for performance and usability, a comparison with Microsofts implementation of remote procedure call was made. The amount of code required to integrate each system was measured, and the raw speed of several simple RPCs was measured.

An evaluation of a real life case study (the image viewer program discussed in the previous chapter) is presented in Chapter 6.

MS-RPC

Microsoft RPC is a modified implementation of the DCE-RPC¹ specification that was standardised in the late eighties. It sits at the heart of the Microsoft Windows operating system and provides essential services - everything from Explorer to the clipboard relies on the ability for one part of the system to communicate with another via RPC. As a result it is highly optimised, with not only the transport layers being optimised for speed over all else (they are in-kernel) but code size is also optimised via the use of a format string interpreter, which runs small marshalling programs inside a virtual machine.

The DCOM system is layered on top of MS-RPC, and adds object orientation to the protocol. This test does not examine DCOM, because FastRPC itself is not object oriented, so it would be an unfair comparison.

MS-RPC is based on the same concepts as regular DCE-RPC and CORBA: an IDL file describes an interface using a C-like language. An IDL compiler is run on this file, which produces a C file containing proxies and stubs. These are linked in to the server and client programs respectively, and then the RPC APIs are used to set up the server main loop and establish the connection.

Performance results

The "perftest" test suite program consists of 3 simple functions. No usage of advanced features such as overlapping buffers, double indirection or re-entrancy is made. The functions simply

sum their inputs and return. This ensures that the timings reflect the RPC overhead and not the actual computation time. The three functions are as follows:

1. `int test1(int num)` - returns `num + 10`
2. `int test2(struct test *arg)` - returns `arg->a + arg->b + arg->c`
3. `int test3(struct test arg)` - does the same as above, but without the pointer indirection (ie tests copy by value)

Each test was run 10,000 times and the time averaged out across the runs. The same was done for one of the test functions called directly, to determine the overhead of a native internal hardware call, however, this was so fast that the average time was always less than a microsecond - below the minimum time that the tests were equipped to measure.

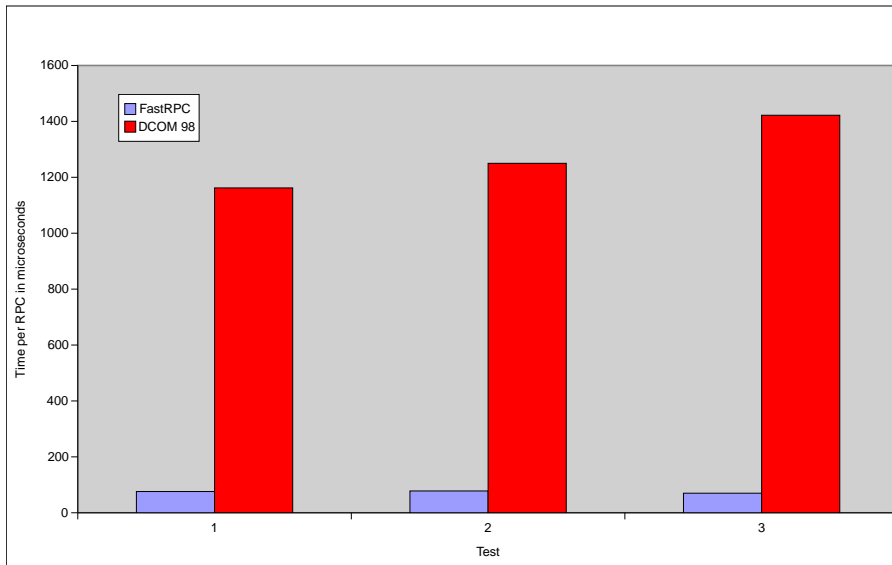
The MS-RPC code was run via Windows emulation on Linux so as to remove the kernel scheduling algorithm from the uncontrolled variables and to make direct comparisons easier. The Microsoft "DCOM 98" implementation that shipped with the Windows 9x series of operating systems was used. Despite the name this provides both DCOM and the underlying MS-RPC framework as well. Comparisons against the (slightly different) NT implementation were not possible on this setup, and would not be reliable anyway as the implementation can and does change with automatic security updates. DCOM 98 is a fixed redistributable provided by Microsoft that has not changed for some time, so it makes sense to test against this (even if it may not show MS-RPC in the best possible light).

Table 5.1. Raw timings for FastRPC vs MS-RPC

	FastRPC (microseconds per call)	MS-RPC (microseconds per call)
Test 1	76	1162
Test 2	78	1250
Test 3	70	1422

The timings are shown in the graph below for FastRPC vs MS-RPC. As can be seen FastRPC beat MS-RPC (DCOM) in all the tests, in one case it was 20x faster. For easier comparison a chart of the timings is below. It should be noted that with more time, the MS-RPC side of the tests could probably be tuned to make it faster, however the test was implemented in the most straightforward way possible.



Figure 5.1. FastRPC vs MS-RPC timings from the "perfctest" program

Due to time constraints CORBA², ZeroC ICE³ and DBUS were not tested, as they are similar in nature to MS-RPC and are unlikely to be significantly faster.

Evaluation of performance results

For a better picture of how FastRPC stacks up performance-wise to competing systems, a wider range would have to be tested. Better result gathering techniques would also be beneficial.

In this case, the 3rd test of FastRPC gave anomalous results: an RPC involving copy by value should increase the amount of data being copied over the socket, and so should have a higher time per call than the others. At only 70 microseconds however it was actually faster than the others.

This is likely to be because even though the time is averaged out over a large number of calls they still complete within a few seconds and as such are quite susceptible to glitches in kernel scheduling and other environmental conditions. Reducing the influence of these factors, either by redesigning the tests or using a larger number of samples, would give more accurate results. Given that statistical noise is unlikely to explain the large gap between FastRPC and MS-RPC though it may not be worth it.

The overall shape of the chart is in line with the design goals of the system. FastRPC was designed to do less work per RPC than traditional systems at the expense of sacrificing language and network transparency. The results support this design decision: 70 microseconds of overhead

is sufficiently low that it could be used for work where performance is important but not critical, such as on the desktop.

Whilst not measured in this test, memory overhead is also likely to be lower, and on modern desktop/laptop systems memory pressure is a large contributing factor to the overall performance equation.

Code impact results

The other key design tenet of FastRPC is that it should be easy to integrate with existing software, where "easy" is defined to mean simple and easy to use APIs, and minimal changes to the codebase.

To measure this, the first metric considered was the size of the differential between the RPC-less version, which used no RPC to perform the tests, and the version which had been integrated with an RPC system. The source code was measured using the Linux line count command (`wc -l`). All source code includes error handling.

The results for the RPC-less and FastRPC versions look like this:

Table 5.2. Lines of code in RPC-less and FastRPC perftests

RPC-less lines of code	FastRPC lines of code	File	Description
16	33	perftest.c	Sets up RPC (if any) and calls the functions many times, timing them as it goes
42	42	perftest-slave.c	Defines the test functions that will be called. Shared between the tests, no changes needed.
29	29	perftest.h	Header file that defines the [RPC] interface between perftest.c and perftest-slave.c
64	64	perfutils.h	Utilities for timing things, shared between the tests
151	168		TOTAL

For MS-RPC there are two possible measurements, one including generated code and one that does not. As generated code does not have to be written by the programmer it could be argued

that it should not be included - however, generated code can be a liability and is not always fully transparent. In this case the MIDL compiler produced code which violated the C specification and relied on a common lvalue assignment extension. Unfortunately as of version 4 the GNU C compiler refuses to compile this code, and as I did not want to introduce an uncontrolled variable into the test by changing the compiler this had to be fixed by hand.

Table 5.3. Lines of code in MS-RPC perftests

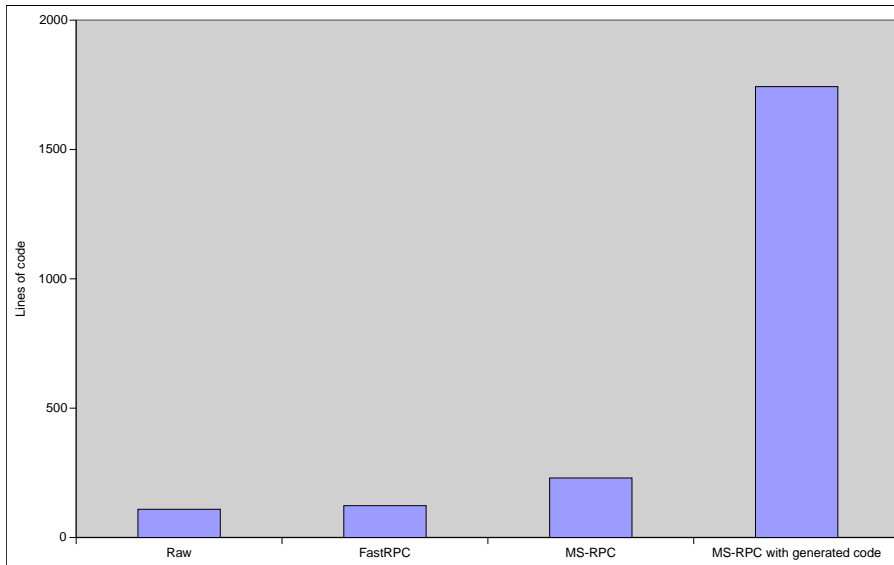
RPC-less lines of code	MS-RPC lines of code	MS-RPC with generated code	Filename	Description
16	53	53	perftest.c/comtest-master.c	Sets up RPC (if any) and calls the functions many times, timing them as it goes
42	96	96	perftest-slave.c/comtest-slave.c	Defines the test functions that will be called. Shared between the tests, no changes needed.
29	29	29	perftest.h	Header file that defines the [RPC] interface
0 (N/A)	30	30	comtest.idl	Interface Definition Language file, which is an MS-RPC specific form of perftest.h
64	64	64	perfutils.h	Misc utilities for timing things, shared between the tests
0 (N/A)	0 (N/A)	758	comtest_s.c	MIDL generated server stub/marshalling code
0 (N/A)	0 (N/A)	755	comtest_c.c	MIDL generated client stub/marshalling code
109	272	1785	TOTAL	

The separated -master -slave files are necessary because the MS-RPC API does not have any concept of forking a process to establish both sides of the connection, both sides must be started manually and meet at a common agreed rendezvous point: in this case a TCP/IP port.



This is clearly a large increase over the raw and FastRPC versions. The bulk of this code is in the auto-generated proxy/stub files (comtest_c.c and comtest_s.c respectively), which even for trivial RPCs are bulky. They also consist of complex, undocumented code which should not be overlooked.

Figure 5.2. Lines of code comparison



Of course, lines of code is a crude measurement. What really matters is the *complexity* of the code required. The code required by FastRPC can be found in Appendix A. It is straightforward and mostly boilerplate.

The MS-RPC code can also be found in Appendix A. Note that this is with MS-RPC in "fully implicit" mode, in which most of the work is handled by the generated code. The IDL looks mostly like C, with some extra annotations that must be learned. As can be seen, a GUID must be generated for the defined interface, and an implicit binding handle specified.

The complexity involved here is far greater than with FastRPC - to even get a basic "Hello World" RPC program up and running the developer must get to grips with binding handles, string bindings, NCA transport identifiers, MIDL memory allocation hooks, protocol sequences, endpoints and server lifecycles.

Endnotes

¹Open Groups Distributed Computing Environment; www.opengroup.org/dce/



²The Object Management Groups "Common Object Request Broker Architecture" (<http://www.omg.org/>); a similar specification to DCE-RPC

³A commercial RPC system by several of the CORBA designers; designed to be a cleaned up and simplified form of the system (<http://www.zeroc.com/>)

Chapter 6. Image Viewer Case Study

There are two sides to the FastRPC coin:

1. Being a fast and easy to use RPC system
2. Increasing security

The last chapter looked at the first side, by comparing its performance and ease of integration to MS-RPC. This chapter looks at how it was integrated into a simple image viewer program to increase security.

The threat

As discussed in the introduction, image decompressors have often been targetted by hackers in the last few years. The combination of complex data structure manipulation in old C libraries tuned for performance resulted in many possible exploits against various widespread codebases. Image loading is a part of many popular applications such as web browsers and instant messengers, so this posed a significant problem.

In this case study, the threat is the possible exploit of a buffer overflow in the widely used "libpng" decoder library. A Portable Network Graphics (PNG) image with a corrupted tRNS segment can cause the CPU to jump into the image file itself, allowing the attacker to gain control of the program and acquire the privileges it was running with. On most desktops, this means they can now do anything the user can do.

This exploit is long since fixed, but older vulnerable versions of the library can still be obtained, and one is included along with this paper.

The code

To simulate such an attack, a simple C program was written that allows the user to pick an image file from their hard disk using a GUI file selector dialog box. The program then loads and displays the image. More complete equivalents to this program are shipped with every desktop OS.

This program can be found in the source code accompanying this paper, under the name `png-show.c`.

Another program, `exploit.c`, can be used to generate malicious image files. This will probably have to be run on the target system, as a successful exploit relies on knowing the location of the stack. As a result if the system provides address space layout randomization, it must be disabled before the attack can operate. The code inside the generated image files is known as shellcode: a small piece of hand-crafted machine code that creates a UNIX command line and connects it to a network socket. Although only a handful of bytes long this code is enough to hand control of the machine to the attacker. Once the shellcode has run anybody can connect to the port specified in the code using `telnet` until the computer is rebooted.

The RPC framework

The integration of FastRPC into this sample program did not take long, and resulted in an additional 34 lines of code. This code had a pleasant side-effect: if `libPNG` crashes for any reason during image decoding, the main application will survive intact and a little "no go" icon will be displayed to the end user. This is true even if the crash was simply due to a bug and not an attack.

Figure 6.1. The image viewer after a crash



Two problems were encountered during the case study:

1. The program does not use libpng directly, but rather uses the GDK Pixbuf convenience API. This simple image loader abstraction ships with every Linux system and provides an object oriented wrapper around image buffers.

The GDK Pixbuf image decoder library uses a type system called GObject. The type system initializes some internal data structures lazily, and as the initialization was restricted to the slave process the pixbuf object returned to the master process was then being used only partly initialized.

That resulted in internal assertion failures from the type system when the Pixbuf object was used. The solution was to force initialization of the GDK Pixbuf type system ahead of time so the registrations were available in both master and slave processes.

2. The Novell AppArmor framework was used in this exercise, mostly because it is simpler than SELinux. The framework allowed programs to open a file then send a usable file descriptor to a confined subprocess, and this was used to restrict the slave process to nearly no privileges at all. However when the program was run again two months later, a change either in AppArmor or the kernel had broken this mechanism. Discussion with the AppArmor developers revealed that file descriptor passing had never been intended to work in this way and the behaviour would not be changing back.

The workaround was to allow the slave process to access any file whose name ended in ".png": this is less secure, but works with the new kernel/AppArmor version.

The AppArmor developers did however state that they would be interested in allowing for file descriptor passing in a controlled, supported way in future versions.

Despite these issues the test was a success: when the AppArmor security system is confining the slave process correctly the shellcode does not run, as the slave does not have the privileges necessary to start a shell nor bind a network socket. So the attack was successfully repulsed, with only minor code changes and no user-visible difference in performance.

The AppArmor security profile for the application can be found in the code accompanying this paper.

Chapter 7. Conclusions

This chapter will summarise the achievements of the project and examine ideas for further development. To recap, the basic goals of the project were to write a remote procedure call (RPC) system that is:

- Fast enough to sit between internal components in C/C++ based software which exchange traffic regularly without causing user-visible slowdown.
- Easy enough for developers that separating their software into multiple processes is an attractive option.
- And by achieving the previous two goals, improve the security of desktop software and make it more resistant to attack from hackers, viruses and bots.

"Enough" is hard to judge, however the results chapter shows that relative to MS-RPC (which stands for all RPC systems like it) performance and ease of integration are much better. The case study shows that a real world application can be integrated with FastRPC to increase its security and prevent a real attack.

The basic deliverables, all achieved, were:

- Exchange pointer and primitive types as function arguments/return values with hard coded RPC wire-ids and dispatch tables.
- Allow for pointers to inside the stack.
- Allow for pointers to inside a shared heap.
- Demo of an application being hacked and once RPC protected, the hack failing.

The intermediate deliverables were:

- Address space management: ensure that the slave process only has the memory mappings it needs.
- Develop a re-usable framework for easy integration with other programs.

The first deliverable here was not achieved, and the second one was: FastRPC comes with documentation and is easily integrated into an existing C/C++ codebase.

The advanced deliverables were:

- Stack barriers for additional security
- Multi-threading support
- Allowing for multiple slaves under the control of a single master

The first deliverable here was achieved, but multi-threading and multi-slave support is not currently implemented. An outline of how thread support might work is provided below.

Problems encountered

Not everything went according to the plan. This section documents some of the difficulties encountered whilst developing the project.

Integration with third party libraries

An unexpected problem that took many hours to debug was the way the 3rd party "GDK Pixbuf" library failed to work correctly when split across processes. This turned out to be an issue with the type system code underlying the library, however, the failure was not obvious.

Given that the case study was quite a simple application, it's possible that as integration complexity grows even more diabolical problems could arise. Fundamentally, much software is not expecting to be "locked down" in this way. Despite this the benefits to be gained from doing so are still significant.

Limitations of Linux kernel security

One of the biggest hurdles was the unexpected change in kernel behaviour towards the end of the project (described in the case study chapter) in which passing opened file descriptors between processes of different privilege levels no longer worked correctly. This was being used to "lend" the slave the ability to read a particular file without it having the privileges to open that file directly.

This was a problem because fixing it so the case study program worked correctly again required significantly weakening the sandbox in which the image decoder ran. Discussion with the AppArmor and SELinux developers revealed that the old behaviour had never been intentional,

and that the current designs of these systems required the policy developer to know *in advance* which files a process would want to access. For the case of processing files selected by an end user in a confined process, this makes it significantly harder or less efficient to implement.

Therefore, whilst that is not actually a problem with FastRPC, safe deployment and integration into existing software really relies upon extensions to these MAC frameworks to allow files to acquire the ability to access a file they could not themselves open via passed file descriptors.

ASLR

Another problem involved the use of address space layout randomization on modern Linux kernels. ASLR is another technique used to improve security by making it harder to exploit buffer overflows. However:

1. ASLR is not universally available. It is not widely available on Windows nor the Mac, and it is sometimes disabled on Linux as it can break software that relies on a particular VM layout.
2. Not every type of attack can be stopped using ASLR, only buffer overflows can. A buffer overflow was used for the case study because it was easy to duplicate a real historical attack, but there are plenty of other approaches attackers can use.
3. ASLR can be defeated in some scenarios using entropy exhaustion and repeated attack attempts.[Shacham04]

For the purposes of this project, ASLR was temporarily disabled to make testing easier.

Ease of debugging

Because the nature of FastRPC is to split function calls over multiple processes debugging such a program can prove tricky. Whilst FastRPC will give coherent stack traces - a significant improvement over standard RPC frameworks - ensuring that a debugger follows the fork may require quite good knowledge of the debugger itself. Also, if there is a bug in FastRPC this can seriously confuse most debuggers as the problem may occur on another stack or inside assembly that doesn't correspond to any actual function.

Most debugging during the development of FastRPC was done the old fashioned way, with print statements and manual analysis of the code.

Areas for future work

There are several areas in which this work could be expanded upon. Most of them involve improving FastRPC or the surrounding software to get closer to the ideal of perfectly sandboxed software. Some of these ideas are discussed more below.

1. Improving framework security
2. Multi-threading support
3. Implementing file descriptor passing or some equivalent in SELinux or AppArmor
4. Handling callbacks and re-entrant code
5. Developing an algorithm that can automatically partition software into modules according to what privileges each part needs. This would be difficult but potentially possible through the use of advanced static code analysis algorithms, but lies more in the realm of software engineering research.
6. Adding support for exception handling and throwing exceptions across RPC boundaries
7. Windows support

Improving framework security

FastRPC currently allows slave processes to modify any part of the stack below the barrier including the return address entries. This could be used to force the master to jump back into the stack and begin executing malicious code with master-level privileges.

Preventing this is not technically difficult. It requires the master to copy the stack being sent back from the slave into a temporary buffer for pre-processing. The original stack should then be unwound and each return address copied from the original stack onto the temporary stack. By effectively "deleting" any changes the slave made to the return addresses this potential attack vector is sealed off.

Multi-threading support

Put simply, a multi-threaded program does more than one thing at once. Threading is tricky at the best of times as the programmer must manually ensure that no two threads attempt to access the same resources simultaneously, and that appropriate design infrastructure is in place to ensure that threads don't interfere with each other.

Currently FastRPC assumes that there is only one thread that will interact with it at any one time. This assumption will be safe for many programs but not all, and so programmers writing multi-threaded code will have to be extra careful in how they use FastRPC.

An extension to make it work with multiple threads would be straightforward: simply locking the internal structures so only one RPC can be processed at once would be suitable in the majority of situations. This would potentially have low performance in the case of many threads crossing the process boundary at once, but that is unlikely in a desktop scenario. In a server scenario it might be more of a problem: in this case effectively having multiple sockets, heaps and dispatch loops inside the slave would eliminate the bottleneck. The only remaining issue preventing perfect transparency then would be thread local storage (TLS) segments. Sufficient OS-specific low level magic could be enough to make this work transparently as well.

More advanced multi-threading support depends on being able to synchronize the master and slave address spaces after the initial fork. Currently FastRPC does not attempt to deal with this transparently, but as discussed in the design chapter tracking address space changes and "balancing the books" between master and slave is required to keep FastRPC as easy to use as possible.

Handling callbacks and re-entrant code

One issue that FastRPC doesn't try to handle at all is that of callbacks. Imagine that some long-running calculation has been separated into a slave process, but the main process wishes to display a progress bar indicating how far the computer has got and how much time is remaining.

The traditional way to implement this would be for the master to supply a *callback* function which the slave repeatedly calls every few seconds with an update on its progress.

Unfortunately FastRPC is currently one way: attempting to call back to the master from the slave will have unpredictable effects. This problem is solvable, by using pipes for instance, but the resulting code would be inconvenient and awkward.

Re-entrancy is a tricky problem which RPC system designers have wrestled with for many years. The approach DCOM and CORBA take is to re-enter the main loop inside the proxy ¹, so that a process can be both simultaneously a server and a client at once. This approach can result in *unexpected* re-entrancy because *any* incoming call can be handled by this loop, not just one from the same logical call stack.

This can cause total havoc when control jumps to a completely unexpected place from "inside" a function that theoretically can't do that. Very difficult to reproduce bugs are the usual result.

A better approach would be to eliminate the typical client/server master/slave terminology in favour of simply having two way "pipes" between processes which don't imply a particular role for each process. Each side would be able to both generate and receive RPCs. Re-entrancy would be handled by tracking the logical call stack identifiers across calls and blocking all re-entrancy attempts except those belonging to the same logical stack.

Developing an automatic partitioning algorithm

It would be nice if there was some automatic way to divide software into components and apply RPC such that a "make my software more secure" program could be written.

Such a program would have to be able to accurately quantize a program into discrete software components and analyse the communication between them. Automatic modularisation of software is an on-going topic of research within the academic software engineering community, and results from this effort could be put to practical use in such a tool. Even so, a distinction should be made between separating already existing modules into separate processes so they can be locked down and actually modularising software - it's possible to imagine a program attempting the former whereas the latter may still require an element of human ingenuity.

Windows support

Whilst FastRPC is designed for Linux-based hosts, the majority of attacks occur against the dominant Microsoft Windows operating system. As such it would be useful to port FastRPC to Windows and apply the techniques to Windows-based software. Even apparently simple programs like the WinAmp media player have been recently exploited to install spyware and advertising software behind the users backs, so there can be no doubt that it would be worth doing.

The main UNIX facility that Windows does not provide is the ability to fork. However, due to the more predictable address space on Windows this could be worked around by using a very small function that links the two address spaces together by hand, as discussed in the design chapter.

That leaves the requirement of a MAC security framework. Windows does not have one by default, but a limited form of this can be added by the CoreForce framework. CoreForce can be worked around by a sufficiently skilled attacker in its current form as it does not restrict the debugger APIs, however, it would still be better than nothing.

Overall conclusion

The project achieved almost all its basic and intermediate objectives. The implementation performed well against Microsoft RPC, the most widely deployed RPC framework in the world today.

Whilst there is still much to be done, the separation of software modules into confined processes using RPC is possible, practical and need not result in a worse user or developer experience. Through the use of new techniques an RPC engine significantly faster than the current state of the art can be built without custom hardware. If these techniques had been used several years ago the image decoder attack demonstrated in the case study would have been less of a threat.

There are many possibilities for further work. Most of them involve extending FastRPC to be more correct, usable with more advanced software, and available on more systems. Some of them require extending the underlying operating system itself to make confinement of processes easier and more flexible.

In the course of writing this paper, several exploits for common programs such as Internet Explorer² and WinAmp³ were found and fixed, but not before being widely used to install malicious software on end users computers. It is clear that our current protections for C and C++ based software are not good enough, and that more must be done in future. It is hoped this papers contribution will help stem the tide.

Endnotes

¹Chris Brumme, Apartmentms and Pumping in the CLR

Conclusions

²eWeek, April 11th 2006; "MS Patch Day: 10 Flaws Fixed in Monster IE Update"

³WinAmp skin file vulnerability: <http://secunia.com/advisories/12381/>

Bibliography

- [Darpa2000] *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*. C, P, C, S, and J. DARPA Information Survivability Conference and Expo (DISCEX), 2000.
- [Ritchie93] *The Development of the C Language*. Dennis Ritchie. ACM SIGPLAN NOTICES, 1993
- [Moffett94] *Specification of Management Policies and Discretionary Access Control*. JD Moffett. Network and Distributed Systems Management, Ed. Morris, 1994
- [NSA98] *The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments*. PA Loscocco, SD Smalley, PA Muckelbauer, and RC Taylor. Proceedings of the 21st National Information Systems
- [Saltzer75] *The protection of information in computer systems*. J. H Saltzer and M. D. Schroeder. Proceedings of the IEE, 63, 9 (Sept 1975), pp. 1278-1308
- [Xerox84] *Implementing remote procedure calls*. Andrew Birrell and Bruce Nelson. ACM Transactions on Computer Systems, 1984
- [Felten97] *Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface*. A Bilas and EW Felten. Journal of Parallel and Distributed Computing, 1997
- [WinNT] *"Undocumented Windows NT"*. Prasad Dabak, Sandeep Phadke, and Milind Borate. Hungry Minds; Bk&CD Rom edition (October 1999)
- [Muller98] *Fast optimised Sun RPC using automatic program specialization*. G Muller, R Marlet, EN Volanschi, C Consel, and C Pu. RAPPORT DE RECHERCHE-INSTITUT NATIONAL DE RECHERCHE EN 1998
- [Box98] *Essential COM*. Dom Box. Addison-Wesley Professional 0201634465.
- [Shacham04] *On the effectiveness of address space randomization*. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. ACM Computer and Communication Security Symposium, 2004

Appendix A. Example code

FastRPC initialization code

```
// declare the slave/server side dispatch table
rpc_call dispatch_table[] = { (rpc_call) &test1,
                               (rpc_call) &test2,
                               (rpc_call) &test3,
                               NULL };

// declare the master/client side proxies
RPC_PROXY("test1", 1);
RPC_PROXY("test2", 2);
RPC_PROXY("test3", 3);

// initialize and error check
static void initrpc()
{
    if (rpc_setup(dispatch_table) >= 0) return;

    fprintf(stderr, "perfctest: could not initialize fastrpc\n");
    exit(1);
}

.... runtest() ....
```

MS-RPC Client/Master code

```
// define hooks for the generated code
```

```
void * __RPC_USER MIDL_user_allocate(size_t size)
{
    return malloc(size);
}

void __RPC_USER MIDL_user_free(void *ptr)
{
    free(ptr);
}

int main(int argc, char **argv)
{
    handle_t rpcBinding;
    RPC_STATUS status;
    unsigned char *szStringBinding;
    // initialize connection
    status = RpcStringBindingCompose(NULL, "ncacn_ip_tcp",
                                     "localhost", "4747",
                                     NULL, &szStringBinding);

    if (status)
    {
        printf("comtest: failed to get string binding: 0x%lx\n", status);
        exit(1);
    }

    printf("comtest: string binding is %s\n", szStringBinding);

    status = RpcBindingFromStringBinding(szStringBinding,
                                         &hBindingHandle);

    if (status)
    {
        printf("comtest: failed to convert string binding: 0x%lx\n", status);
        exit(1);
    }
}
```

```
RpcTryExcept
{
    runtest();
}
RpcExcept(1)
{
    printf("comtest: rpc exception 0x%lx\n", RpcExceptionCode());
}
RpcEndExcept

RpcStringFree(&szStringBinding);
RpcBindingFree(hBindingHandle);
}
```

MS-RPC Server/Slave code

```
void stop()
{
    RpcMgmtStopServerListening(NULL);
}

void * __RPC_USER MIDL_user_allocate(size_t size)
{
    return malloc(size);
}

void __RPC_USER MIDL_user_free(void *ptr)
{
    free(ptr);
}

int main(int argc, char **argv)
{
```

```
RPC_STATUS status;

/* select the TCP/IP protocol sequence and endpoint */
status = RpcServerUseProtseqEp("ncacn_ip_tcp",
                               20,
                               "4747",
                               NULL);

if (status)
{
    printf("Failed to register RPC protocol sequence endpoint\n");
    exit(status);
}

status = RpcServerRegisterIf(PerfTest_v0_0_s_ifspec, NULL, NULL);

if (status)
{
    printf("Failed to register RPC interface: 0x%lx", status);
    exit(1);
}

printf("listening ... \n");
status = RpcServerListen(1, 20, FALSE);
if (status)
{
    printf("Failed to start server: 0x%lx\n", status);
    exit(1);
}
}
```

MS-RPC IDL

```
import "unknwn.idl";

[
    uuid(aef87bde-2251-4418-82f8-53415295f795),
    pointer_default(unique),
    implicit_handle(handle_t hBindingHandle)
]
interface PerfTest
{
    struct test
    {
        int a;
        int b;
        int c;
        char *x;
        char *y;
        char *z;
    };

    /* test basic rpc */
    int test1(int num);

    /* test pass by reference */
    int test2(struct test *arg);

    /* test pass by value */
    int test3(struct test arg);

    void stop();
}
```