Appears in the *2003 IEEE Symposium on Security and Privacy,*
Oakland, CA 2003

Vulnerabilities in Synchronous IPC Designs

Jonathan S. Shapiro
Department of Computer Science
Johns Hopkins University
shap@eros-os.org

Abstract

Recent advances in interprocess communication (IPC) performance have been exclusively based on thread-migrating IPC designs. Thread-migrating designs assume that IPC interactions are synchronous, and that user-level execution will usually resume with the invoked process (modulo preemption). This IPC design approach offers shorter instruction path lengths, requires fewer locks, has smaller instruction and data cache footprints, dramatically reduces TLB overheads, and consequently offers higher performance and lower timing variance than previous IPC designs. With care, it can be performed as an atomic unit of operation.

While the performance of thread-migrating IPC has been examined in detail, the vulnerabilities implicit in synchronous IPC designs have not been examined in depth in the archival literature, and their implications for IPC design have been actively misunderstood in at least one recent publication. In addition to performance, a sound IPC design must address concerns of asymmetric trust and reproducibility and provide support for dynamic payload lengths. Previous IPC designs, including those of EROS, Mach, L4, Flask, and Pebble, satisfy only two of these three requirements.

In this paper, we show how these three design objectives can be met simultaneously. We identify the conflict of requirements and illustrate how their collision arises in two well-documented IPC architectures: L4 and EROS. We then show how all three design objectives are simultaneously met in the next generation EROS IPC system.

Keywords: *operating systems, capability systems, interprocess communication, vulnerability.*

1 Introduction

Thread-migrating interprocess communication (IPC) designs are one of the driving forces behind the current resurgence of interest in microkernel-based systems. In particular, IPC implementations by Liedtke [20, 21], and Shapiro [30] have shown that protected IPC invocations can be reduced into the 135 cycle range on Pentium-family processors.[1] Experimental analysis by Ford [8] shows that much of this performance advantage cannot be achieved in asynchronous IPC mechanisms. Given these results, performance-motivated reports of the demise of microkernels [3] may have been premature. The question of security-motivated demise remains open.

The *potential* security benefit of high-performance IPC is straightforward. In contrast to asynchronous or buffering IPC designs [1, 27], the latency of thread-migrating IPC is low enough that applications can be factored into multiple protection domains, each encapsulated by a process boundary and selectively linked by protected, IPC-based communication. Domain-based isolation is an essential building block for high-assurance systems [19, 33, 29, 5, 12, 31]. At least one commercial system has been constructed using this approach to domain enforcement [11].

Selected denial of service attacks against the L4 microkernel and its servers (including several re-examined here) have been briefly examined in the literature [22]. This paper provides an in-depth exposure of application-level vulnerabilities that are implicit in *any* synchronous IPC system. When a synchronous communication mechanism is introduced, the security and trust implications of sender blocking must be considered. When the system design also incorporates user-level pagers [1], interactions between these pagers, string transfers, and blocking must also be considered. Relatively experienced IPC designers have failed to recognize the potential security issues inherent in this combination [7].

A satisfactory IPC design must simultaneously satisfy certain functional, testability, and security requirements. In particular:

[1] These results predate the introduction of the Pentium family fast system call instructions. Comparably careful implementations of those designs on current machines should therefore reduce the common case IPC time down into the 60-70 cycle designs.

- **Asymmetric Trust**: The IPC architecture must not embed any assumption that the sender trusts the recipient.

- **Reproducibility**: The behavior of the IPC primitives should not be influenced by exogenous factors such as system load. This is required for testability and operational predictability under variations in workload.

- **Dynamic Payload**: The IPC architecture must support transmission of messages whose length is not known in advance to the receiving party. Without dynamic payload support, procedures returning (e.g.) precise (unbounded) integers or dynamically sized vectors cannot be handled at the IPC interface.

This particular set of requirements is deceptively difficult to satisfy simultaneously. Due to the presence of timeouts, Mach and L4 fail the reproducibility requirement. In the absence of timeouts or truncation, Pebble [9] and Fluke [7] fail the asymmetric trust requirement. Prior to the work reported here, EROS failed the dynamic string requirement.

This paper examines in detail certain denial of service problems that arise in synchronous IPC designs. We identify the cases in which there is potential for such denial, and describe the available mechanisms in EROS and L4 by which applications may attempt to protect themselves. We show how the next generation EROS IPC subsystem simultaneously meets all three of the requirements identified above, and in the process show that functional requirements such as support for dynamic message payloads need not sacrifice reproducibility. The primary goals of this paper are to ensure that denial of service vulnerabilities are not neglected in future IPC designs, and to show how they are addressed in the next-generation EROS IPC system.

The balance of this paper proceeds as follows. In section 2, we selectively review the L4 and EROS IPC designs. Section 3 describes the problem of asymmetric trust in detail and the options for dealing with this problem. Section 4 performs a further case analysis on IPC safety in conditions of asymmetric trust, identifying many cases where existing, low cost mechanisms are sufficient to address the problem and introduces the ***trusted buffer object*** (TBO), a solution for the general case. Section 5 justifies why the TBO can be trusted where an ordinary client cannot, and establishes additional requirements on the IPC subsystem that are necessary to achieve an effective TBO design. The remaining sections describe related work, acknowledgments, and conclusions.

2 Review of the L4 and EROS IPC Designs

This paper explores two interprocess communication designs for purposes of illustration and discussion: those of L4 and EROS. From a feature perspective, the two designs have been converging for several years. Both have been carefully engineered and specified, but they embody somewhat different views of how IPC should be implemented. Before proceeding to the body of the paper proper, it is useful to briefly review these two IPC designs. The descriptions provided below are not comprehensive, focusing instead on those portions of the two IPC systems that are relevant to denial of service vulnerabilities.

2.1 L4 IPC

The L4 IPC system [20, 21] implements a single basic primitive ***SendAndReceive***. This operation sends a message to a named thread (or task) id and blocks waiting for a response. The receive phase can be omitted, allowing message sends that initiate a new thread of control within the receiver.

As part of the specification of the receive phase, the process performing a SendAndReceive can indicate whether it will accept a message from any sender (an open wait) or only from a particular thread (a closed wait). This enables the implementation of remote procedure calls: by receiving using a closed wait, the invoker of the RPC is assured that the next incoming message will be received from the invoked thread.

The payload of an L4 IPC operation consists of a bounded vector of "immediate words," some number of which are guaranteed to be transmitted in registers. This word vector is optionally followed by a bounded "dope vector," each entry of which can specify either a byte string of arbitrary length (limited by the address space size) or an address space mapping to be transferred. Ignoring some complications arising from scatter/gather support, the receive specification must include corresponding dope vectors specifying where each logical component of the message is to be received.

An L4 message whose payload is fully registerized is guaranteed not to induce an exception in either sender or recipient. Transmission of a data string can lead to receiver side page faults that are handled by user-mode fault handlers.

In general, there is no guarantee that the specified destination process is in the receive state at the time of the IPC operation. To avoid indefinite blocking, L4 invocations include two timeout values: the amount of time to wait before transmission is initiated, and the amount of time that can be spent servicing page faults. If either timeout is exceeded, the operation is aborted and an error is signaled to the sending process. The specified timeout can be infinite.

Vulnerabilities The two key weaknesses of the L4 design are the absence of protection in the specification of the recipient (any sender may invoke any process) and the presence of timeouts in the IPC interface. In addition, there are two minor issues: the IPC mechanism reveals the sender's task id to the recipient, and the widely reported high-performance Pentium implementation fails to save and restore segment register values.

The absence of IPC protection deprives a service of the ability to restrict its callers. A hostile process may perform denial of service attacks by repeatedly invoking operations on an arbitrary target process. This can be mitigated using the original "clans and chiefs" mechanism [23], but this solution imposes significant overhead on all invocations. A more efficient solution proposed in [22] is to allocate one server thread per client and use closed waits. This solution requires all service applications to accept the complexity of multithreading, has unfortunate implications for resource consumption, and does not address the problem of session establishment for such services. An experimental design modification by Jaeger has proposed a capability-like IPC redirection mechanism [12] that provides access control and renders the recipient identity opaque. This design has not yet been incorporated into the L4 specification, but is expected to add between 20% (typical) and 50% to the cost of an L4 IPC operation.

The timeout mechanism mitigates certain low-cost denial of service attacks, but not effectively: the receiving server thread is occupied by the IPC operation until the timeout occurs. The presence of timeouts in the primitive IPC specification impedes reproducible operation unless workloads and queueing behavior can be predicted. Timeout failures are difficult to test in real systems, and have the potential to cascade in unpredictable ways.

While the L4 IPC system has the ability to transmit multiple, independently labeled strings, the inclusion of multiple strings has little impact on vulnerability. The primary concern in string transmission is page fault handling rather than aggregate string length or number of strings.

It is debatable whether the minor issues identified above should be viewed as significant vulnerabilities. Revealing sender task identity is clearly a failure of encapsulation. In principle this is unfortunate, but it is not clear that the revelation has significant security consequences. One concern is that revealing the caller task identity demonstrates authority only indirectly, and makes interposition between client and server difficult [13]. Failure to save and restore segment registers reveals the underlying interrupt event flow to any process that cares to observe it. This is a high-bandwidth inward covert channel, but the hole could be straightforwardly closed with minor alterations to the implementation.

2.2 EROS IPC

The EROS IPC system was derived from that of KeyKOS [11], but has been influenced by various aspects of the L4 design. EROS implements three primitive invocations: SEND (which does not wait), CALL (sends and enters a closed wait), and RETURN (sends and enters an open wait). The fact that a process is in an open or a closed wait is explicitly recorded in the process state of the waiting process.

In contrast to the L4 design, the open/closed wait distinction in EROS expresses a restriction on the capability type that must be invoked rather than the process id of the invoker. A receiving process in the ***available*** state (the open wait) must be invoked using a ***start capability*** to the receiving process. If the receiving process is not in the available state, this invocation will block. A receiving process in the ***waiting*** state (the closed wait) must be invoked using a ***resume capability*** to the receiving process. Resume capabilities are produced by the CALL operation and are consumed when invoked. The process model guarantees that if a resume capability exists to a given process, that process is in the waiting state. This ensures that invoking a resume capability will never block waiting for interprocess rendezvous.

The payload of an EROS IPC operation contains four registers, four capabilities, and a bounded string. Forthcoming enhancements will remove the string bound and provide L4-like scatter/gather support. In contrast to L4, an EROS process must hold a capability to the process it wishes to invoke. Capabilities are named indirectly in the invocation by specifying their index in a kernel-managed per-process capability list. EROS provides only limited means for mapping transfers: the sender can explicitly construct a mapping structure and transmit a capability to this structure to the recipient. EROS also provides a translucent forwarding object known as a ***wrapper*** that allows capabilities to be selectively rescinded.

As with L4, an EROS invocation whose payload consists only of registers and capabilities is guaranteed not to induce an exception in either sender or recipient. Transmission of a data string can lead to page faults in the recipient if the receive region has not been properly prepared by the recipient. If such a page fault would require the invocation of a user-mode page fault handler, and the invocation is a resume capability invocation, the transmitted string is truncated. This behavior guards against hostile clients that seek to block a server indefinitely.

The interactions between EROS invocation types and capability types allow two mutually trusting processes to establish an "extended mutual exclusion" using co-routine style CALL operations. Process A performs a CALL on B. Instead of returning, B performs a CALL on the resume capability to A. The two processes continue in this fashion until both are done, and the final invocation performed is a RETURN operation. Until this final invocation is performed, whichever party is blocked is in a closed wait. This ensures that third-party invocations cannot interrupt the transaction in progress.

Vulnerabilities The key weakness of the EROS design is its requirement that the client know in advance an upper bound on the size of any message returned by a server.[2] This is a functional rather than a security failing, but it is rather a nuisance when implementing a language-neutral capability interface.

The EROS truncation rule has been mis-characterized as a correctness flaw by Ford *et al.* [7]. In fact, it reflects a conscious decision that servers are not obligated to return correct data to hostile clients. Liedtke also found the truncation decision surprising (personal communication). Discussion of its rationale resulted in the introduction of separate page fault timeouts into the L4 IPC system. In the absence of either truncation or timeouts, delivery is assured at the cost of exposure to denial of service. We note that given a choice between exposure, truncation, and timeout there are really no desirable outcomes. The solution that we are adopting in the next generation EROS system is feasible only in capability-based systems, and relies heavily on EROS's ability to factor resource and trust dependencies.

3 Asymmetric Trust

A key, under-recognized requirement in IPC systems is the ability to support asymmetric or qualified trust relationships between two communicating processes. This occurs whenever multiple client applications call a single server application, and is especially acute when the common (shared) application is a reference monitor. While the clients trust the server to process their requests, a correct server is presented with a curious set of conflicting objectives:

- It must respond to the client requests according to the requirements of its interface and operational specification.
- It cannot trust that a given client will faithfully execute its part in the application-level invocation protocol. In particular, no assumptions can be made by the server about the possibility of client-induced blocking unless blocking is precluded by the primitive IPC mechanism.

[2] While it has not been emphasized in prior publications on the EROS IPC system, the EROS system includes a translucent forwarding object. The original purpose of the translucent forwarder was to support selective rescinding of client access. Selective revocation of this kind can be used defensively to revoke hostile sessions in the event of request flooding attacks.

A hostile client might attempt denial of service on other clients simply by calling a server and blocking indefinitely in a non-receiving process state, causing the server in turn to become blocked when it attempts to reply. We will refer to this type of a client as a ***defecting client***.

Four basic design features have been attempted in various IPC designs to manage and mitigate this problem:

Buffering Mach [1], along with most early IPC systems, implements message buffering. Buffer blocks are kernel allocated, which effectively converts a well-localized attack on a single server into a global denial of resource attack on the entire operating system.

Multithreading L4 [20, 21] and Mach support multithreaded servers. In L4, threads are explicitly allocated resources, and a given server will eventually run out of them. Multithreading is therefore not effective against the denial of resource attack, but it *does* ensure that the attack also pressures the kernel scheduler, expanding the scope of the vulnerability.

Truncation EROS [30] ensures that all server replies are "prompt" by virtue of its IPC specification. A ***prompt*** invocation is one that cannot be blocked by the actions of an untrusted party. The EROS process model guarantees that the client is in the proper state when the server replies using a resume capability. However, this is only a partial solution. Unless care is taken, user-level pagers can be used by a hostile client to block a replying server in a non-returning user-level page fault handler. To preclude this attack, EROS truncates the message and reports the fact of truncation to the client whenever a user-mode handler would need to be invoked. Defined pages that have been removed by the single-level store implementation do not result in truncation, because the kernel is trusted to supply them promptly.

Timeouts L4 [20, 21] includes a timeout specification as part of each invocation, with reserved values to mean "do not wait" and "wait forever." While a timeout prevents total blockage of a server, a small number of hostile clients can exploit the use of a server-side timeout to implement severe denial of service against other clients.

A summary of which features are included in commonly referenced IPC systems is given in Table 1.

Feature	**Mach**	**L4**	**EROS**
Buffering	Yes	No	No
Multithreading	Yes	No	Via Retry
Truncation	No	No	Yes
Timeouts	No	Yes	No

Table 1. Summary of defensive features in commonly referenced IPC implementations.

3.1 Effect of Buffering

As we have already mentioned, kernel-implemented IPC buffering creates opportunity for global denial of service by performing a denial of resource attack on the available kernel buffers. Because of Mach's lazy copy feature, which performs large copies using transparently kernel-implemented copy on write mappings, attacking the Mach IPC system in this way requires fairly large copies: the pressure on pages is converted into pressure on page tables. When implemented on most hierarchical memory management architectures, a small number of cooperating hostile processes can exhaust Mach kernel memory on a 32-bit machine by performing misaligned copies of a 2 gigabyte region; each copy allocates enough page tables to consume 8 megabytes of kernel memory.

To perpetuate this attack, the attacking process need never acquire the full 2 gigabytes. By starting with a smaller region whose size just exceeds the kernel's policy threshold for copy on write transfer a hostile process can use multiple transfers to create the large region without actually holding a large number of data pages. The essence of the problem is that neither party in the transaction is "charged" for their mapping pages by the kernel.

While the additional copies imposed by buffering are undesirable, the *security* problem with buffering arises from a misattribution of burden (cost). By failing to attribute the cost of buffer storage to an appropriate process, the kernel becomes open to attack. The VAX/VMM security monitor mitigated this issue using memory quotas [18], but a quota-based approach is not practical in efficient IPC systems. Introduction of such a quota mechanism into a synchronous IPC system must eventually result in the delivery of an allocation fault to a user-mode exception handler, which is exactly the problem that EROS faces with user-supplied page fault handlers.

The alternative is to implement an IPC primitive that operates over buffered channels in pipe-like fashion. This approach abandons essentially all of the IPC performance advances of the last 20 years, and the resulting IPC primitive is too slow to support an adequate degree of compartmentalization. Neither L4 nor EROS implements buffering in the kernel, and it is now generally accepted that buffering should not be included in the design of a kernel-level IPC primitive.

3.2 Effect of Multithreading

The effect of multithreading on IPC denial of service takes two forms. If new threads are allocated transparently by the kernel, there is an implied denial of resource attack by exhausting kernel memory. If threads are allocated explicitly by the server process, then the attack requires N copies of the hostile client, where N is the number of available server threads.

Multithreading is required in certain servers, but from an assurance perspective it is best avoided on complexity grounds. Where a single-threaded server is feasible, the complexity of concurrency management is eliminated and assurance of the server is more easily achieved.

L4 provides multithreading-aware IPC in the microkernel. EROS will shortly implement a RETRY operation that allows a service to force a caller to re-perform its last invocation on a service-supplied capability. This mechanism allows an EROS service to selectively block and unblock callers in the kernel after examining their request. Most cases of multiplexing and out of order return can be handled using only a single service thread by leveraging this mechanism.

3.3 Effect of Truncation

Truncation imposes a subtle requirement on all object interfaces: an upper bound on the length of the reply must be known by the client at the time of the call so that it can pre-probe the receive area to ensure that needed pages are defined. This is an impediment for operations that return (e.g.) vectors or strings, as it is common for both to be dynamically sized. In the general case, neither of these types can safely be used as the return value from an interprocess call unless the underlying IPC system supports dynamically sized return values.

EROS inherited its truncation approach from KeyKOS [11] and it has seemed adequate for many years. We have lately come to feel that it is problematic. Several changes in the EROS IPC system over the years have combined to alter our view:

1. The expansion of the KeyKOS [11] one page IPC payload limit to 64 kilobytes in EROS reduced the length pressure on messages, and started us thinking about how to map CORBA-like object invocations directly onto the primitive invocation mechanism.
2. This in turn led to a paper design, to be implemented in the next round of EROS IPC modifications, that removes the string length bound entirely (subject to the limits of the address space size).
3. The subsequent design and implementation of CapIDL, which provides a language-neutral interface definition language for capabilities, drew our attention to the fact that unbounded dynamically sized vectors cannot be supported if the recipient must know the message size in advance.
4. Our desire to use E [25], a capability-based scripting language, as a scripting language for EROS objects led us to introduce a standard `GetSignature()` operation on all conforming capabilities. This operation returns a string whose length cannot be known to the client in advance.

> As luck would have it, the `GetSignature()` operation is the first time we have needed to define a commonly used operation with a response size that is not knowable at call time.

To our knowledge, EROS and KeyKOS are the only current IPC systems implementing truncation. L4 does not.

3.4 Effect of Timeouts

As previously discussed, timeouts impede predictable behavior and testability. Personal discussion with Yoon Ho Park regarding how timeout values were actually used in L4-based SawMill multiserver system [10] concluded that the use of L4 timeouts (and timeouts in general) can be divided into four categories:

1. Do not wait at all (corresponds to EROS truncation).
2. Wait indefinitely
3. Wait for some period motivated by an externally specified duration in the real world. For example, if a disk drive is specified to have a maximum seek time of 12ms, but no seek completion interrupt has been received within 20ms, something is probably wrong.
4. All other values. In principle any such value must be incorrect unless the dynamic range of possible workloads can be specified in advance.

In practice, L4 services do not use timeouts in the last category, typically resorting either to infinite timeouts or to non-blocking IPC operations. This suggests that in practice the addition of a timeout value is an ineffective guard against defecting clients.

Setting aside the question of effectiveness, timeouts embedded in IPC operations present difficulties in *friendly* use: they preclude testability, and they do not interact favorably with debugging (because the recipient may be stopped). In practice, when such a timeout is triggered it is usual for the IPC to fail, but the IPC sender is nonetheless blocked for the duration of the timeout when it could potentially be doing useful work.

When IPC timeouts are leveraged to support client-side page faults during IPC (as in L4), the timeout mechanism can be exploited by hostile clients to implement efficient denial of service attacks against other clients that share a common server (e.g. a window system) with the attacker. The attack proceeds by first implementing a client-side page fault handler that simply never waits for a page fault notification. With this page fault handler in place, the client sends a string containing an undefined page to the shared server. The receiving server thread (in L4: task) is rendered inaccessable until the timeout expires. In consequence, well-behaved clients cannot invoke the server. Multithreading does not circumvent this attack. It simply requires that several duplicates of the attacking client be used. All of these duplicates can share in common a single defecting page fault handler.

L4 provides timeouts, EROS does not.

4 Case Analysis for Suspicious IPC

In seeking a solution to the problem of asymmetric trust, we would like to satisfy three objectives:

1. We would like to avoid introducing complexity whenever possible. Complex strategies used by a server manifest as latency experienced by its clients.
2. When a client defects, we would like to impose the cost of misbehavior on the client.
3. We must avoid solutions that convert a localized denial of service into a systemwide denial of service.

4.1 Sources of Vulnerability

Before looking for a general solution to the defecting client, it is useful to first understand the cases in which the problem is actually a threat. This is useful because of an "escape hatch" in the requirements on the IPC subsystem: we need not guarantee service to attackers. If it can be determined that a receiver is defecting, the IPC subsystem can be absolved of its responsibility to deliver the message.

The potential sources of blockage in the IPC process lie in the invocation type (divisible or indivisible), the invokee state, the invoker's transmitted string (page faults), and the invokee's received string (page faults).

4.1.1 Invoker Thread, String Vulnerabilities

As long as IPC invocations are specified as indivisible, we can rely on the fact that the invoker cannot cause the invocation to block without completing. To prevent invoker-side page faults, the IPC implementation can behave as though a "dry run" is performed on the sender side before the real invocation, effectively causing all sender-side page faults to occur prior to the invocation and ensuring that the sender-side string is ready prior to invocation. Following a successful dry run, we may assume that exceptions in the send string region constitute defection and abort the invocation. This is how the EROS IPC mechanism is specified.

4.1.2 Invokee in Wrong State

The next concern is to consider whether the recipient may be in the wrong state to accept the IPC invocation at all. We must here consider two kinds of invokee states: open waits and closed waits. The blocking behavior expected by an invoker depends on the expected state of the invokee.

When an invoker process P_0 performs an IPC to a server process S that is in an open wait, there is an implicit race with all other processes that might be invoking S. It is possible that some other process P_1 beats P_0 to the invocation and P_0 may consequently block for a period of time controlled by S. Therefore, in any such invocation the invoker is implicitly declaring that they permit the invokee to indefinitely block the invoker.

In a closed wait matters are quite different. The invoker expects that the invokee is waiting for a response. If for some reason the invokee is *not* waiting in a closed wait, then the invokee has defected. In practice, the closed wait arises when a server is responding to a previous client call. This is the case in which prompt completion is required to prevent denial of service attacks against the server.

In the L4 IPC system, there is no direct coupling between the invokee state and the invoker invocation. Where the processes involved know that procedure-call semantics is expected the client can use a "wait forever" timeout and waits in a closed wait. The server responds using a "do not wait" timeout.

In the EROS IPC system, a ***start capability*** is a capability whose invocation will block unless the recipient is in the available (open wait) state. A ***resume capability*** must be used to invoke a process in the waiting (closed wait) state. Any operation that causes an EROS process to leave the waiting state causes all outstanding resume capabilities to be efficiently rescinded, guaranteeing that client-side debuggers cannot be exploited as a means to attack servers. Resume capabilities are consumed as they are used. This prevents a server that is later compromised from performing denial of service attacks on past clients.

4.1.3 Receive String Page Faults, Large Strings

When the invocation protocol declares that the invokee may legally block the invoker, as in "wait forever" (L4) or start capability (EROS) invocations, we need not consider *why* blocking occurs. By agreeing to block, the invoker is implicitly saying that it trusts the receiver to make decisions about the disposition of the invoker's thread of control. As long as it can tell that this is the case, the IPC subsystem is free to run an invokee-specified page fault handler. If the invoker has specified that the invocation must be prompt, as in the do not wait (L4) or resume key (EROS) invocations, invokee page faults cannot be serviced unless a guarantee of prompt completion can be achieved.[3]

Unless care is taken in the design, a parallel problem can arise if long strings are to be transferred. Real-time schedulers rely on the fact that clock-driven and interrupt-driven preemption are recognized quickly by the processor. As a result, the internal implementation of an IPC that transmits long strings must break the operation into multiple internal units of operation, each having a maximum duration chosen to avoid interfering with the scheduler.

Given the need for internal units of operation, the IPC designer may be tempted to make the external (i.e. application-visible) *specification* of the IPC primitive divisible in some form. There are two security design hazards here:

1. The server must not be blocked by an untrusted client when engaging in an extended transfer, and

2. The server must not be required to indefinitely hold state in hopes that an IPC might later be resumed, which precludes servicing *other* requests while there is an incomplete divisible IPC outstanding.

To avoid these hazards, the scatter/gather and long string enhancements to the EROS IPC specification allow an IPC to be cleanly aborted, but do not permit it to be "paused" prior to completion.

4.2 Application Layer Implications

With the sources of vulnerability characterized, we can finally examine the impact on the application-level remote procedure call protocol in cases where invokee page fault handlers cannot be safely executed.

Provided that IPC operations are specified as indivisible, the difficult case arises only when *all* of the following conditions apply:

- The invocation type precludes blocking,
- The payload size cannot be known in advance by the receiver, *and*
- The side-effects or computational cost of the operation preclude recovery by delivering a truncated response to the client, adjusting the client receive area, and replaying the invocation.

Our experience with EROS has been that the vast majority of interprocess invocations have a statically specified upper bound on their payload in both directions, even when data motion is involved. Read and write requests, for example, typically specify an upper bound on the buffer length. Whenever such a bound can be established, we can reasonably require that well-behaved recipients will pre-validate their receive buffer areas. In all such cases, the sender can robustly use a non-blocking, truncating IPC operation. Most replies carry no string at all. For those that do, the string is generally smaller than one page, so the receive area validation requirement imposes no significant performance cost.

[3] I have chosen to gloss over the distinction in L4 between page fault servicing timeouts and invokee readiness timeouts; the essential point is that the invoker has said that they will not agree to block for invokee pagers.

If the invokee of a non-blocking IPC operation knows the length of the expected message and fails to provide adequately validated buffer space then it has not correctly executed the higher-level RPC protocol. In this case we can presume that it has defected. The non-blocking requirement is therefore a concern only when a well-behaved receiver is *unable* to bound the length of the reply.

Many of these unboundable cases – in particular those that involve the return of strings – are replayable. One example is the previously described "get signature" operation:

```
replayable
vector<char, *> GetSignature(IF);
```

In this invocation, the server can respond using a non-blocking IPC. If the response string exceeds the valid buffer length of the recipient, the reply is truncated. As long as the IPC primitive reports to the stub layer the number of bytes that were sent (as opposed to received) the stub can transparently increase the receive string buffer size and retry the invocation.

4.3 The Difficult Case

We have now reduced the difficult case – non-replayable invocations with client-unknown payload – to a relatively rare set of situations: those in which a side effect occurs or the cost of performing a "dry run" invocation (replay) is prohibitive. In L4, the solution at this point is a page fault IPC timeout. Because EROS is a capability-based system, it offers the possibility of a more flexible solution.

If a large message is to be transferred in a single logical operation, the client cannot reliably accept it, and the server must not block, then *somebody* must provide memory to hold it. In a typical IPC design, either the sender holds it in sender memory and uses some timeout mechanism to decide when to abort the invocation, or the kernel must provide buffers. Based on the IPC specification, it initially appears that EROS must either truncate these invocations or incorporate some form of timeout into the capability invocation mechanism. This was the source of the mischaracterization in Ford *et al.* [7]. The next generation EROS system provides the underpinnings for a third solution.

First, we must admit that the problem of server blocking has been slightly mischaracterized. We have said that the server must not block, but this is overly constraining. A more precise statement is that the server cannot be blocked *by an untrusted party*. In particular, if a trusted buffering agent were available, the server could safely encapsulate the message in a buffer for later consumption by the client.

The problem with buffering in general is that (a) it is expensive, and (b) the wrong party pays for the storage – usually either the kernel or the sender. The first is merely a nuisance. The second is a a potential cause of denial of service. In EROS, it is possible to create a buffer that can be *trusted* by a server but *paid for* by the client. We refer to this object as the ***trusted buffer object*** (TBO). Ideally, the use of a trusted buffer object would proceed as follows:

1. When CapIDL (the EROS IDL compiler) is asked to generate an invocation stub for a non-replayable operation that has an unboundable reply string, it modifies the invocation signature to use a trusted intermediary: the trusted buffer object. The client invokes the TBO, which invokes the server on behalf of the client.
2. On receipt, the server verifies using means described below that the trusted buffer object is authentic. If this verification succeeds, the server knows that this object really executes the trusted buffer object program. As a consequence, it knows that the TBO can be counted on not to defect. Further, it knows that any buffer storage space used by the TBO is ultimately paid for by the client program.
3. The server transmits its response string to the TBO using the "extended mutual exclusion" provided by the EROS IPC primitives. This has one of two outcomes:
   (a) The TBO accepts the entire string, or
   (b) The TBO runs out of space and reports this.
4. On completion, the TBO returns to the client, passing the string supplied by the server.

Note that a client can reuse its TBO object repeatedly. Once created, the per-invocation overhead of the TBO is minimal.

A protocol optimization is possible to avoid recopying the string from the TBO to the client. Instead of accepting the string that the TBO attempts to return to the client, the client may elect to accept zero bytes worth of data. Following completion of the TBO's reply, the client can directly map the TBO's address space into client memory. If desired, the TBO space can be *permanently* mapped into client memory at TBO creation time, eliminating the overhead of call-time mapping manipulations.

During an early review of this design, Mark Miller observed that the same memory mapping optimization might be feasible on the *server* side of the transaction, yielding a zero-copy implementation for long string transfers. The problem with this lies in the fact that the server must be prepared to recover from unsatisfied page faults within this mapping region when client space is exhausted. Asking the TBO how much space is available does not help, because the client has the authority to revoke this space in mid-transfer. Our sense at this point is that the efficiency of normal TBO invocation is high enough, and recovering from these faults is complex enough, that this approach will not usually prove to be worthwhile.

By introducing the trusted buffer object into the protocol, the entire burden of resource allocation is placed on the client in such a way that the server is safe from denial of resource and denial of service. In particular, the TBO is transmitted using capability arguments to the IPC primitive, and these are guaranteed to be transmitted promptly. The overhead of using the TBO is largely lost in the noise when very large strings are transferred.

5 Trusting the TBO

Given that a server cannot in general trust its clients, we need to account for why injecting the TBO as an intermediary is helpful, and what constraints must be satisfied if the TBO is to be safely trusted. The constraints are straightforward:

- The returning server must be able to determine whether it is returning to a TBO – that is, to a process executing code that can be trusted to respond promptly.
- The server must be in a position to know that the TBO will actually *execute* that code – that is, that the client cannot disable the scheduling authority under which the TBO is executing.

Prior to the work described here, EROS provided means for client authentication and recovery from destruction, but did not provide adequate control over scheduling authority to support the TBO.

5.1 TBO Authentication

The identification of a calling application (as distinct from a calling principal) is directly supported by the EROS ***constructor*** [32] mechanism. TBO objects are created by the TBO constructor. Every EROS constructor "brands" each process that it creates by inserting a unique capability into a reserved capability field of the created process (the brand slot). The brand capability can be any convenient capability type. To ensure that its brand is universally unique and private, the constructor uses a distinguished start capability to itself as the brand for its products. This distinguished start capability is guaranteed to be accessible only to that constructor.

To support application authentication, the EROS kernel implements an operation known as the ***Identify*** operation. Given a process, resume, or start capability to a process P and an alleged brand capability, the `Identify` compares the alleged brand capability with the actual brand previously recorded in the process P. The operation returns `true` (`false`) according to whether the two brand capabilities are (are not) equal.

Using this kernel primitive, any constructor is able to identify its products. Given access to the TBO constructor, any service can identify whether it is returning to a TBO.

5.2 Assurance of Execution

There are four ways in which the client might prevent the TBO from executing:

1. The client might reclaim its storage by destroying the space bank (the source of storage) from which it was created. In this event the capability to the TBO will become detectably invalid, so the server is able to detect defection.
2. The client might reclaim TBO storage in mid-transaction, while the server is waiting for the TBO. The EROS kernel already makes provision for defense against this: destruction of a process causes its outstanding resume capabilities to be invoked, waking the server.
3. The client might contrive to starve the TBO of storage by providing an inadequately populated space bank. The TBO code guards against this and returns an appropriate error if this occurs, allowing the server to detect defection.
4. The client can disable the TBO's schedule during execution.

To address the last problem, we are adding a new feature into the next generation EROS IPC mechanism: ***schedule donation***. If both invoker and invokee agree to do so, the invoker's schedule capability will be copied to the invokee as a side effect of the invocation. Agreement is required so that a service can decline to execute under caller control. Similarly, it should not be possible for an arbitrary program to acquire a thread of control from its caller by trickery. The L4 IPC design provides a similar mechanism.

The specification of schedule donation is that the recipient continues to execute under the donated schedule until such time as it explicitly alters its own scheduling authority, either explicitly or by accepting another donation. This implies that the donating process trusts the recipient to give up the donated schedule when appropriate. As the TBO is trusted, schedule donation is appropriate here. Having established the identity and safe scheduling control of the TBO, the server returns an extended string to the TBO using a ***non-prompt return*** operation (one that permits page fault servicing).

Once a string has been loaded into the TBO by a server process, the only operations the client can do are

- Fetch the string. Doing so forces the client to donate scheduling authority to the TBO object, relieving the server of any remaining scheduling exposure that might arise from an error in the TBO implementation.

- Ignore the TBO. In this case the TBO will retain the server schedule until destroyed, but as it will not initiate instructions without an invocation from the client, and any such invocation will force a schedule donation, we do not really care what schedule the TBO acts under.

Note that neither of these actions imposes additional client-controlled costs on the server-supplied TBO scheduling authority. Scheduling isolation is preserved.

6 Related Work

While many of the pieces described here rely on several generations of improvement in the EROS system, EROS owes a tremendous debt to its predecessor KeyKOS [11].

A great deal of work on thread-migrating IPC has been done in the last decade, most notably by Liedtke [20, 21], Ford [8], and Shapiro [31]. Though the KeyKOS capability invocation mechanism [11] predates it, most of the current work on thread-migrating IPC derives in some measure from the "lightweight remote procedure call" work by Bershad [2].

Work on packet filtering [26, 24, 6] is similar in flavor to the more restricted filtering performed by filtering indirectors. We are not aware of such filters being used to *defer* packet processing, nor do they appear to have been used to filter dynamically tagged messages.

Use of registers to speed interprocess communication was heavily used in the V Distributed System [4], and appears to have been independently proposed by Karger [16] in connection with the SCAP system [17, 15]. The SCAP design gains particular advantage if the trust relationship between caller and callee is known to both parties.

Liedtke *et al.* [22] have considered selected denial of service attacks against the L4 microkernel and its servers, including several of the problems identified here. Their work does not examine application-level vulnerabilities in depth, nor does it propose design refinements to the L4 IPC architecture that might mitigate its vulnerabilities.

The dynamic return payload problem addressed in Section 4.3 is identical to the one way network transfer problem in multilevel secure systems, noted by Karger and independently by Rushby [14, 28]. In a one way network transfer, the recipient is incapable of signaling error to the sender and therefore must provide sufficient storage to accept a message of unknown length.

7 Acknowledgements

The impetus for the work described here was the implementation of the CapIDL capability IDL system, which is derived closely from earlier IDL work in CORBA. CapIDL was jointly designed by Mark Miller and the author. A recent discussion with Dave Presotto helped spark the last connections that led to the invention of the trusted buffer object.

EROS builds heavily on the KeyKOS system designed by Norm Hardy, Charles Landau, Alan Bomberger, and Bill Frantz while at Tymshare, Inc. All of these people have been helpful and patient in describing the workings of KeyKOS and encouraging the development and evolution of the EROS system. Bryan Ford has been kind enough at various points to explain a number of details in the Fluke implementation and the Mach version 4 IPC implementation.

Jochen Liedtke took the time several years ago while we were both at IBM to discuss the merits of timeouts and the problem of hostile pagers. Jochen's continuous advances in the performance and design of microkernel operating systems led to improvements in the EROS implementation and drove us to a deeper and more careful understanding of operating system design. Yoon-Ho Park similarly took time to discuss the experiences of the L4 team in building the SawMill system [10].

This paper originated from a discussion with Jay Lepreau at the Symposium on Operating System Design and Implementation in 1999, which first prompted us to recognize that vulnerabilities in synchronous IPC systems arise from the complex and subtle interaction of many factors in their designs, and that these interactions had not been adequately explored in the archival literature.

8 Conclusions

In addition to fast performance, an effective interprocess communication system must provide reproducible behavior, deal with assymetric trust among communicating processes, and enable support for messages that contain dynamically sized payloads. This paper describes why simultaneous satisfaction of these requirements is challenging, and identifies a set of enhancements to the current EROS system that let us meet all three design objectives. Two key enablers of the solution proposed here are the ability in EROS to authenticate the code executed by an application independent of its user, and the ability (via confinement) to protect a trusted program from tampering by its user. No generally satisfactory means to satisfy all of these requirements simultaneously has previously been proposed.

To our knowledge, no previous papers have been published that expose in depth or adequately address the interprocess denial of service vulnerabilities that are implicit in synchronous IPC designs. The analysis presented here illustrates in detail an unusual case of authority factoring: the provider of the trusted buffer's storage authority and the provider of its execution authority need to be distinct in order to prevent certain classes of denial of service attacks. Based on other ongoing design activities within the EROS community, this appears to be an example of a general pattern that emerges in many places where multiplexing crosses a trust boundary.

The solution proposed here leverages application-based authentication heavily. We rely on the ability to identify and safely execute components that are instantiated by untrusted providers, and whose storage originates from an untrusted source (the client). It is difficult to see how this particular way of straddling the trust boundary can be achieved without some form of protected naming primitive. In our TBO approach, the server relies on receiving such a name (the resume capability) as a basis for authenticating the TBO. With care, a cryptographic hash of the invoker executable image injected by the kernel into every invocation might serve as a substitute for capabilities. Note that this hash can be cached in the kernel's per-process data structure; its use need not be expensive.

Given the number of IPC systems (and more broadly, operating systems) that do not adequately support communication across assymetric trust relationships, it appears that the issues involved are not widely understood. One goal of this paper is to ensure that the problem of asymmetric trust is not neglected in future IPC designs, and to describe one strategy for how to support it.

References

[1] M. Acceta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian Jr., and M. W. Young. Mach: A new kernel foundation for UNIX development. In *Proc. 1986 USENIX Summer Technical Conference*, pages 93–112, June 1986.

[2] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *Proc. 12th Symposium on Operating Systems Principles*, pages 102–113, Dec. 1989.

[3] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proc. 14th Symposium on Operating Systems Principles*, Dec. 1993.

[4] D. Cheriton. The v distributed system. (3), Mar. 1988.

[5] *U.S. Department of Defense Trusted Computer System Evaluation Criteria*, 1985.

[6] D. Engler and M. F. Kaashoek. Dpf: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. SIGCOMM '96 Conference*, pages 53–59, Stanford, CA, USA, Aug. 1992.

[7] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the fluke kernel. In *Proc. 3rd Symposium on Operating System Design and Implementation*, pages 101–115, Feb. 1999.

[8] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating threads model. In *Proc. Winter USENIX Conference*, pages 97–114, Jan. 1994.

[9] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The pebble component-based operating system. In *Proc. 1999 USENIX Annual Technical Conference*, pages 267–282, Monterey, CA, USA, June 1999.

[10] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *Proc. ACM SIGOPS European Workshop*, Sept. 2000.

[11] N. Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, Oct. 1985.

[12] T. Jaeger, K. Elphinstone, J. Liedtke, V. Panteleenko, and Y. Park. Flexible access control using IPC redirection. In *Proc. 7th Workshop on Hot Topics in Operating Systems*, pages 191–196. IEEE, Mar. 1999.

[13] T. Jaeger, J. E. Tidswell, A. Gefflaut, Y. Park, K. J. Elphinstone, and J. Liedtke. Synchronous IPC over transparent monitors. In *Proc. Ninth ACM/SIGOPS European Workshop "Beyond the PC: New Challenges for the Operating System"*, Sept. 2000.

[14] P. Karger. *Non-Discretionary Access Control for Decentralized Computing Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, May 1977. MIT/LCS/TR-179.

[15] P. Karger. *Improving Security and Performance for Capability Systems*. PhD thesis, University of Cambridge, Oct. 1988. Technical Report No. 149.

[16] P. A. Karger. Using registers to optimize cross-domain call performance. *ACM SIGARCH Computer Architecture News*, (2):194–204, Apr. 1989.

[17] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proc. of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, Oakland, CA, Apr. 1984. IEEE.

[18] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, (11):1147–1165, Nov. 1991.

[19] B. W. Lampson and H. E. Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19(4):251–265, May 1976.

[20] J. Liedtke. Improving IPC by kernel design. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 175–188. ACM, 1993.

[21] J. Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical Report GMD TR 933, GMD, Nov. 1995.

[22] J. Liedtke, N. Islam, and T. Jaeger. Preventing denial-of-service attacks on a μ-kernel for weboses. In *Proc. HotOS-VI*, May 1997.

[23] J. Löser and M. Hohmuth. Omega0: A portable interface to interrupt hardware for l4 systems. In *Proc. First Workshop on Common Microkernel System Platforms*, Dec. 1999. Revised: Jewel Edition, January 5, 2000.

[24] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proc. USENIX Technical Conference*, pages 259–269, Jan. 1993.

[25] M. S. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments. In *Proc. Financial Cryptography 2000*, Anguila, BWI, 2000. Springer-Verlag.

[26] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, USA, Nov. 1987.

[27] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the Chorus distributed system. Technical Report CS-TR-90-25, Chorus Systemes, F-78182 St. Quentin-en-Yvelines Cedex, France, 1991.

[28] J. Rushby and B. Randell. A distributed secure system. *IEEE Computer*, 16(7):55–67, 1983.

[29] M. D. Schroeder, D. D. Clark, and J. H. Saltzer. The MULTICS kernel design project. In *Proc. 6th ACM Symposium on Operating Systems Principles*, pages 43–56. ACM, Nov. 1977.

[30] J. S. Shapiro, D. J. Farber, and J. M. Smith. The measured performance of a fast local IPC. In *Proc. 5th International Workshop on Object Orientation in Operating Systems*, pages 89–94, Seattle, WA, USA, Nov. 1996. IEEE.

[31] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, Dec. 1999. ACM.

[32] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proc. 2000 IEEE Symposium on Security and Privacy*, pages 166–176, Oakland, CA, USA, 2000.

[33] W. A. Wulf, R. Levin, and S. P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*. McGraw Hill, 1981.