Design of the EROS Trusted Window System

Jonathan S. Shapiro shap@cs.jhu.edu
John Vanderburgh vandy@cs.jhu.edu
Eric Northup digitaleric@digitale.net
Systems Research Laboratory
Johns Hopkins University

David Chizmadia dchizmadia@promia.com
Promia, Inc.

Abstract

Window systems are the primary mediator of user input and output in modern computing systems. They are also a commonly used interprocess communication mechanism. As a result, they play a key role in the enforcement of security policies and the protection of sensitive information. A user typing a password or passphrase must be assured that it is disclosed exclusively to the intended program. In highly secure systems, global policies concerning information flow restrictions must be honored. Most window systems today, including X11 and Microsoft Windows, have carried forward the presumptive trust assumptions of the Xerox Alto from which they were conceptually derived. These assumptions are inappropriate for modern computing environments.

In this paper, we present the design of a new trusted window system for the EROS capability-based operating system. The EROS Window System (EWS) provides robust traceability of user volition and is capable (with extension) of enforcing mandatory access controls. The entire implementation of EWS is less than 4,500 lines, which is a factor of ten smaller than previous trusted window systems such as Trusted X, and well within the range of what can feasibly be evaluated for high assurance.

1 Introduction

Window systems play a key role in modern computing systems. They serve as the primary mediator of user input and output, and provide an interprocess communication mechanism (cut and paste) that is widely used and universally expected. Most modern window systems trace their conceptual ancestry to the Xerox Alto [31]. In the Alto design, applications are presumptively friendly and the computer display is a single-user device. A basic goal of the Alto design was to encourage cooperation among applications in an environment of trust. These assumptions and goals are inherited by both the X Window System [27] and Microsoft Windows. Unfortunately, these assumptions of trust are incompatible with even minimal standards of security.

Window systems have direct access to sensitive information, both in the form of sensitive input (e.g. passphrases) and timing data. They implement critical paths in support of selected trusted applications (e.g. the login service). They are necessarily party to the enforcement of global information flow restrictions when systemwide mandatory access controls are in effect. Current designs include shared mutable resources, which are an obvious no-no, and provide a remarkable amount of server-side resource used to hold client data, creating a rich field of opportunity for storage denial of service. They perform operations that have high variance and observable latency, the combination of which facilitates both timing denial of service and covert channel construction. As a result, window systems provide a wealth of vulnerabilities that attackers can exploit – even in otherwise compartmentalized systems. Attention to security in their design is vital.

In the late 1980's there was a flurry of work on compartmented mode workstation (CMW) implementations. Proceeding from requirements put forward by Mitre [33], TRW developed Trusted X, an implementation of the X Window System suitable for multilevel secure environments based on the CMW requirements [10]. This effort identified both major and minor design flaws in X, many of which could not be fixed compatibly and remain issues today. Related work at Mitre was conducted as part of the compartmented mode workstation effort [4, 20].

The CMW effort gave essentially no attention to potentially hostile actions occurring within an MLS compartment. As noted by Abrams [1], this is also true of the *Trusted Computer System Evaluation Criteria* [7] and (in our opinion) the *Common Criteria* [14]. Viewed in light of current-day commercial threats, this ommission seems problematic. Defense against a scripting virus that simulates a password request prompt or creates an excessive number of windows falls outside of the scope of most compartmentation strategies; these are problems of trusted path and discretionary control. In today's environment, it seems optimistic to assume that two applications in a single mandatory access control domain are mutually friendly. The mandatory policy does not object to information moving between these two processes, but it would be useful to ensure positive confirmation of user intent: some identifiably authorizing action performed by the user, such as a keystroke corresponding to a "paste"

It was an explicit goal of X Version 11 to specify mechanism, not policy.

David Rosenthal, *Inter-Client Communications Conventions Manual* [26]

Figure 1: X11 Design Philosophy

operation.

This paper presents the objectives, design, and analysis of the EROS Window System (EWS). EWS is a new trusted window system for the EROS capability-based operating system. It is a "fresh start" design that provides robust traceability of user volition and is capable (with minor extension) of enforcing mandatory access controls. Building on the primitive mechanisms of the EROS operating system, we have created a window system that is a factor of ten smaller than previous trusted window systems such as Trusted X. The implementation provides an efficient, double-buffered display system that significantly reduces the number of resources that must be managed by the display system, and ensures that all resources used in support of a client session are allocated from client resources. The implementation is under 4,500 lines of code. Future enhancements will include a high-performance 3D graphics rendering pipeline comparable in performance to the direct rendering [15] of X11 or Microsoft's DirectX mechanism. This enhancement is *not* expected to significantly increase the size of the security-enforcing code.

2 Objectives and Overview

In their extensive security review of X11, Epstein and Pisciotto [9] state that "Authentication is the most obvious security problem with X." In today's commercial threat environment, this characterization seems generous. The most obvious security problem in X is the absence of policy of any sort (Figure 1). The goal of the X11 designers was to maximize the ability of applications to interoperate, partly to promote a new vision of computer interaction. In the quest for a policy-free design, even the user is disintermediated from control. Given a request for the content of the paste buffer from an application, there is no way that an X server can determine whether the user has performed any action authorizing that paste. X assumes not only that applications are cooperative, but that their actions reflect the volition of the user. In a world of increasingly hostile applications, this trust assumption has become an unsupportable luxury.

EWS proceeds from the diametrically opposing position: our goal is to ensure that the user is actively in the decision loop, and complete isolation between applications is our default. Having adopted confinement as a fundamental organizing principle within the EROS system, we are unwilling to permit unrestricted information flow at the window system. Our goal is to ensure that any communication between window system clients is authorized by *both* the user and the applicable mandatory control policy, and that this communication proceeds only in the direction that the user and policy indicated. Capability systems provide natural underpinnings for direct manipulation, which simplifies secure user interface design [35].

That said, there is an enormous user investment in the idioms of current window systems. In particular, the "cut and paste" and "drag and drop" idioms are now universally adopted and expected. Users have become accustomed to overlapping window systems, and to applications with closely coupled, render-intensive interaction loops. Given this, we wished to create a window system in which the "look and feel" of current usage idioms could be largely preserved.

2.1 Principles and Goals

After reviewing the conclusions of Epstein and Piccioto, we arrived at a list of design principles and goals for EWS:

R1. **Isolation** No operation performed on one client session should be able to affect or observe state associated with other sessions – in most cases, not even the state of subsessions.

R2. **No Mutable Sharing** The display server should provide no shared mutable state to clients.

R3. **Minmize Server Resource Types** The total number of resource types managed and/or allocated by the server should be minimized. This is one aspect of overall complexity reduction.

R4. **Minimize Algorithmic Complexity** Many graphics operations are complex and have high variance. The number and complexity of algorithms and data structures in the server should be minimized.

R5. **Restricted Communication** The display server should provide strictly limited interprocess communication facilities. Provide what is necessary to support current usability idioms; nothing more.

R6. **Volitional Traceability** No communication may occur between applications through the display server unless we can demonstrate an authorizing user action.

R7. **Resource Conservatism** The display server should not enqueue either input requests or events. Both promote resource denial of service and covert channels. Output events may be queued, but total output queue length should be bounded. More generally, the display server should operate using only bounded resources. Dynamically allocated resources, if any, should come from the client.

R8. **Small Size** The display server should be small enough to be evaluable. Our initial goal was to achieve the 30,000 LOCC target of Trusted X.

R9. **Low Variance** Each input event should be delivered to exactly one recipient application, and each operation should complete in fixed, small time. The display server should not multiply messages. Similarly, each incoming request should in general have one response. When more than one response is necessary, the total number should be a small constant integer.

With three exceptions, we were able to achieve these objectives:

1. Clipboard interaction establishes a temporary unidirectional communication channel. It necessarily involves notification of both sides by the display server, which is a small multiplication of messages (and therefore violates R7).

2. Our design supports hierarchical client subsessions. This hierarchy expresses visual containment only; subsessions are fully isolated from their parent sessions for communication purposes with one exception: destruction of a session implies destruction of all descendant subsessions (violates R1).

3. Window structures are dynamically allocated using display server memory (violates R5). A quota system is needed to limit communication achieved by exhausing the total number of available server window structures. A quota of this sort will also limit attacks that operate by creating large numbers of windows. This has not yet been implemented.

2.2 Design Overview

The functions of a display server can be divided into five main categories:

1. Input processing, including events and client requests

2. Rendering and display update.

3. Interprocess communication (cut and paste).

4. Trusted user interaction and feedback, which includes window decorations, labeling, and trusted path management.

5. Isolation support.

We will discuss how each of these is approached in EWS in the sections that follow, and then examine how a variety of security concerns are addressed by the design.

As a capability system, EROS is object-based. In consequence, EWS is an “object server,” and requests are performed by synchronously invoking operating system protected capabilities. It has become conventional to speak of a server that responds to interprocess procedure calls in this fashion as an “RPC Server.” We emphasize that all interprocess communications in EWS are *local* remote procedure calls [5]. The EROS capability invocation mechanism [29] provides a high-performance transport for such invocations. For reasons that will become clear below, the synchrony of these invocations is not a bottleneck to display performance.

The EWS display server does not directly implement remote connection or cryptographic transport layer protection. Both are cleanly separable functions that have generic utility for many applications. There is no reason that the display system should duplicate this function when it can be satisfactorily implemented in a separably assurable component. Cox *et al.* [6] propose a compelling architecture for separating transport security and key management from applications.

The display server also omits authentication functionality. In a capability system posession of a capability is a necessary and sufficient proof of authority to perform the operations invokable through that capability. In the context of EWS, a client either possesses a `Session` capability or they do not, and distribution of capabilities is a separable problem. User accounts are created with an initial desktop session that can be detached and reattached by the login subsystem. Responsibility for subsession

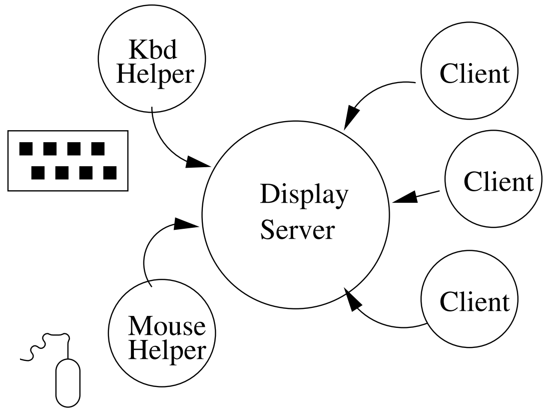

Figure 2: EWS Components

creation initially lies with the user's primary "shell." In a multilevel secure system, this shell would be a trusted application have responsibility for creating compartments and associating security labels with the subordinate Session capabilities that it grants to applications within these compartments.

3 Input Processing

In the EWS design, the hardware frame buffer and hardware input devices are "owned" by the display server process (Figure 2). Each input device has an associated process that blocks on that device waiting for a hardware-level input event to occur. This event is reprocessed into canonical form by the helper process, and the helper process then invokes the display server to "post" the event using a synchronous RPC operation. From the display server perspective, all interactions arrive as remote procedure calls from some process. Requests from device helpers and requests from generic clients arrive on distinguishable interfaces by virtue of the fact that the associated RPC invocations are performed on distinct capabilities.

In contrast to many other display servers, incoming requests are generally not queued by the display server. Each is processed immediately and enqueued on the outbound event queue of the receiving client. In the case of mouse events, the events are delivered to the client session owning the window in which the event occurred. A *MouseDown* event causes all subsequent mouse events until the corresponding *MouseUp* to be delivered to the window in which the *MouseDown* occurred (but see the discussion of "drag and drop" in Section 6). In-order processing imposes three constraints on the display server:

1. Requests must be "prompt," by which we mean that their completion must not involve any operation that might be blocked pending the completion of some other request. An acceptable exception to this rule is requests that explicitly *request* to block.

   As discussed by Mercer [18], blocking or queueing requests results in priority inversion, which in turn creates a covert signalling opportunity.

2. Requests must be low latency.

   Requests executed by the server run under a different schedule than that of their client. Their latency therefore can be seen as a source of variance in real-time context switch latency. Given the design of the EWS display server, it is more effective to establish a small upper bound on request latency than to attempt priority queueing solutions that might require a operating system support for multilevel scheduling.

3. Requests should not incur large variance in processing latency. Variance of this form can be exploited for both resource denial of service and covert signalling.

While the display server does not enqueue requests in general, it *does* perform queueing in connection with client-requested rendevous and client event delivery. When a client issues a *WaitMouseEvent* request, the server checks the per-client-session list of undelivered events (which is bounded). If one exists, it is returned, otherwise the client request is queued. EROS provides an operation, RETRY, that allows the server to redirect the client to a kernel stall queue whose wakeup is controlled by the display server. At a later time, the desired input event will cause this client to be awakened, whereupon it will reissue its request.

The difference between RETRY-based queueing and application-level queueing is subtle but significant. Because clients queued using RETRY are blocked on a kernel queue, their reactivation honors the scheduling policy of the operating system. Application level queue implementations, including the request dispatch queue of X11, generally do not have access to OS-level priority information. Even if they did, dynamic adjustments to priority cannot safely be revealed to such applications.[1] EROS directly exposes the operating system queueing mechanism

[1] Revealing dynamic reprioritization supports efficient covert channel construction.

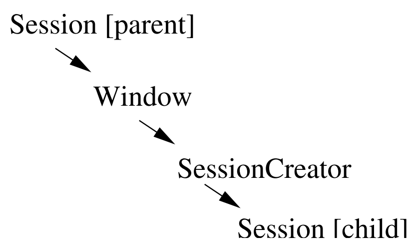

Figure 3: Nested Sessions

via a capability-protected interface and provides operations that allow the display server to exploit it for queueing purposes.

4 Sessions

Sessions serve both as the means by which windows are created and as the unit of containment for mandatory access controls. Sessions are hierarchical. A top-level application running within one session can implement its own mandatory control policy within that particular session if desired. From a security standpoint, this is primarily useful for debugging, but it also supports separation of concerns. While the manager can ensure that the communication activity of a subordinate application across session boundaries is restricted, the manager is *not* able to observe the internal events and actions of that application.

4.1 The Session/Window Hierarchy

EWS associates each window with a unique client session. Every EWS window is created by performing a *CreateWindow* operation on some `Session` capability. Every session has an associated containing window, and the windows created using that `Session` capability are created as child windows of the session's parent window. Client sessions are hierarchical: having created a window W, the holder of a `Session` capability can create a new `SessionCreator` capability whose parent window is W. This new `SessionCreator` can be provided to newly instantiated applications, and effectively defines the context of the root window with respect to that application. The `SessionCreator` can be used by the subortinate application to create new `Session` capabilities (Figure 3).

Operations in one session are not observable by other sessions – not even by parent sessions. Input events are delivered to the owning session of the window in which they occur. The hierarchical session mechanism allows us to construct graphical shells that appear to contain their applications. Interactions with these client applications are not be observable by the shell.

The intermediate `SessionCreator` provides a mechanism for validating isolation. The receiving client is able to perform a test on the `SessionCreator` capability verifying that it is really a capability to the display server. The client is then assured that sessions created using this `SessionCreator` are exclusively held by the client, and cannot be spied on by the owner of the parent window. Similarly, since the parent window owner never possesses the Session capability, the parent cannot create windows that might attempt to deceive input processing directed to the client. As seen by the user, the resulting window structure appears entirely normal (Figure 4).[2]

4.2 Mandatory Controls

While EWS does not currently implement mandatory access control, the session system has been designed with mandatory controls in mind. The unit of mandatory control labeling is the session. Communication operations between two windows are permitted only if a label-checking predicate indicates that the communication is permitted. At present, we have implemente only the trivial predicate that returns *true* in response to all requests. This amounts to an entirely discretionary control policy.

However, the access predicate function need not be implemented in the window server. If provided by a trusted source – either at startup or statically at system design time – an independent process can implement the mandatory control predicate. This allows the same mandatory control agent to be used by many subsystems, and isolates the two implementations for assurance purposes. It also permits different subsystems to implement different mandatory access control policies within their respective compartments. Our intended usage model is that the top-level "shell" is a trusted agent (it must be, since it is part of the trusted path), and this shell is therefore permitted to specify a mandatory control agent and a label when creating application sessions. The EROS IPC mechanism is fast enough to make such an external access control agent practical.

Because EROS is persistent, there is no need to reestablish these session relationships each time the user logs in. Instead, each user account executes an independent copy of

[2] We have not yet determined how best to display multilevel labeling. Any labels of this sort would certainly depart from current user expectations.

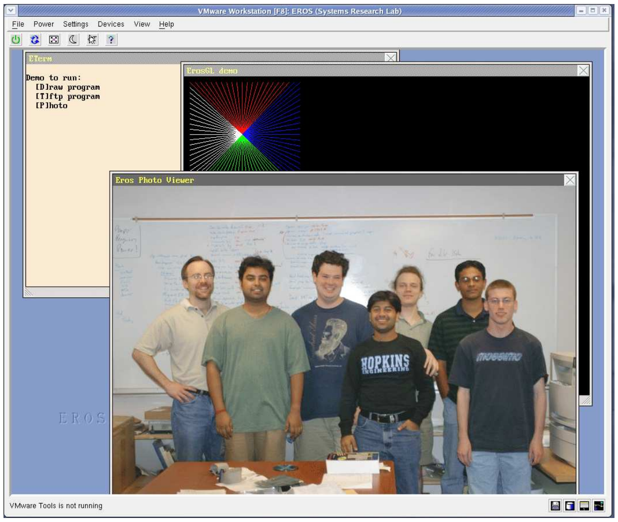

Figure 4: Windows in Multiple Sessions.

the entire window system. At login time, the login agent connects the per-user window system to the frame buffer and hardware input devices. On logout, these connections are severed.

5 Rendering

In order to reduce the complexity of the server, EWS implements drawing operations in the client rather than the server. This is also motivated by issues of variance. In the presence of clipping, operations such as *DrawPolygon* may involve several orders of magnitude more processing than line drawing. Further, in the absence of "backing store," server-side drawing necessitates that update notices be sent from the display to the client when portions of its windows are revealed. It is usually more space-efficient to have the application redraw than to save the bits, and memory was precious on early workstations. One hazard of this design is that a client can manipulate the visibility of its windows so as to exploit expose events as a signalling channel. A second hazard is that delivering these expose events imposes significant execution costs on the display server.

To simplify the server, we initially considered restricting the client to constructing complex structures using successive *DrawTriangle* commands following the logic of OpenGL [28], but abandoned this approach when a faster and simpler method became apparent: shared memory.

EWS uses shared memory mappings between client and server to represent window state. When a client wishes to create a window, it supplies to the display server a read-only shared memory region containing a bitmap. The display server maps this shared region, and subsequently performs *bitblt* operations to transfer portions of the bitmap to the physical display. The client renders directly into the bitmap, and advises the server when changes have occurred by issueing an *UpdateRectangle* request to the server. Note that this reverses the flow of traditional update notification, and eliminates the channel associated with X11 update notices. It also permits the server to redraw the frame buffer at login time without communicating to the clients. The resulting design is conceptually similar to the Apple Quartz architecture [3], and can be extended to encompass the high performance 3D pipeline features of Quartz Extreme. A side benefit of client-side rendering is that the server is no longer responsible for font handling. The EWS display server contains a single, compiled-in font that is used for title bars. This allows us to use *bitblt* for window titles without incorporating the complexity of a complete font rendering system into the server.

Given the underlying EROS primitives, it is possible for the client to rescind the shared memory region without notice to the display server. This may occur out of malice or because the client's storage is revoked for reasons beyond its immediate control. The display server *bitblt* routine is the only routine in the display server that reads the client memory region. It is wrapped by an exception handling catch block. In the event of an invalid memory reference, the display server assumes that the client has reneged on its entire interaction contract and rescinds that client session.

Note that the shared mapping contract ensures that a given client can detect at most one *bitblt* operation performed by the display server, and only by using a mechanism that causes the client session to be severed.

5.1 Invisible Windows

To support isolation in nested client sessions, EWS provides restricted support for invisible windows. An invisible window has no backing bitmap, and is not a candidate for input events. *In order to receive input, a window must be visible.* The role of an invisible window is to provide a coordinate space that contains a subordinate client session.

We note that restricting invisible windows does not entirely resolve problems of event hijacking. Pragmatically, there is no difference in appearance between an invisible window and a borderless visible window all of whose pixels have fully or nearly transparent alpha values. At present, EWS does not support borderless top-level windows. This mitigates, but does not eliminate, the problem of alpha transparency.

6 Interprocess Communication

Conventional window systems handle cut and paste interactions using a broadcast communication mechanism. When a user performs a "cut" operation, the application performing the cut claims ownership of the cut buffer by sending a message to the display server. The server retransmits this notification to all clients who have registered an interest in clipboard notifications. This approach, and the security issues that arise from it, are well documented [20]. In EWS, no events are transmitted to the destination until the user performs a "paste" action, and then only if the communication is permitted by applicable mandatory access controls.

Tying the server paste logic back to a clearly identifiable user action is necessary to limit certain types of covert communication. In the absence of a traceable user action, any client could claim that a paste had occurred at any time. Cut actions are similarly hazardous, because a hostile client could interfere with permitted communication by falsifying cut events. Collectively, these concerns motivate a desire to make "paste" into a traceable atomic action. For keyboard-initiated cut or paste (e.g. Control-V), traceability is not a significant challenge. The server maintains a key table indicating which well-known keystrokes authorize cut and paste actions. The drag and drop protocol similarly has clearly identifiable interaction events.

Mouse-initiated cut and paste operations are trickier, because there is no simple way to relate mouse actions to application claims that a cut or paste has been performed. We considered moving menu management into the display server, but felt that this would both complicate the display server and unnecessarily restrict application designers. Further, it would require significant changes in existing graphical toolkits and would therefore present an impediment to portability.

6.1 Traceable Cut and Paste

EWS resolves these problems by introducing a new type of invisible window. We require that the visible regions conveying cut and paste authority be identified by the application. The cut and paste windows accept no events, but clicks "passing through" these windows result in clipboard authorization. For each of these special windows there is a distinguished standard (server defined) cursor used to indicate when the mouse is above these regions. When a MouseUp event occurs within one of these special windows, the display server knows that an authorizing user action has occurred. To ensure positive user feedback, the server will not perform a cut or paste operation unless the distinguished cursor has been visible for a minimum amount of time. This prevents unintended cut or paste actions that might result from randomized modifications of the window positions, but allows the application to simulate multiple active regions by relocating the active region to fall under the mouse as the mouse moves.

Note that both the "cut" and the "paste" operation require tracing. The user must know both where the data is coming *from* and where the data is going *to*. The EWS display server keeps a record of which windows own the cut and paste contexts at any given time.

6.2 Drag and Drop

In general, the EWS design avoids situations where one client gets notified of interaction sequences associated with another. This was the motivation for directing mouse sequences beginning with a MouseDown and ending with a MouseUp to the MouseDown window.

There is one widely accepted user idiom that conflicts with this handling of mouse events: drag and drop. We have resolved this by providing direct support in the display server. At any time after receiving a MouseDown followed by a MouseMove, the origin client window can optionally inform the display server that this mouse sequence is a drag action. In that case, subsequent MouseMove events may be delivered to other windows in the form of DragOver events, and the final MouseUp event (which completes the drag and drop idiom) is delivered to *both* the originating and the destination window.

Two points should be noted here with regard to covert channels and multilevel security:

- The display server is aware that the drag and drop idiom is a precursor step to an act of communication. DragOver and MouseUp events are delivered to the window under the mouse only if that window would be permitted to receive the data transfer implied by the drag and drop idiom.

- DragOver events are not a significant covert channel, because they are limited by the rate of user input.

6.3 MLS Format Negotiation

In MLS systems, a problem with both "cut and paste" and "drag and drop" can arise from format negotiation. The client is prepared to provide some number of different formats, but does not wish to render all of them because most of them will not be used. The recipient has a (hopefully intersecting) set of formats that it wishes to receive. At a minimum, this set includes the native format (e.g. so that the drawing can be transferred back to the original application for subsequent editing) and at least one common format that the recipient can render. The usual approach to negotiating formats is that the sender sends a list of transmissable formats and the recipient replies with the subset that it wants. This is acceptable in a single-compartment environment, but in an MLS environment, this downward communication is not permitted.

An elegant way of eliminating the downward communication problem is feasible in systems that, like EROS, provide a confinement mechanism [17, 30]. The EROS operating system provides a utility service called a ***constructor*** that instantiates new programs. Among the services provided by the constructor are the ability to verify that newly instantiated programs created by that constructor have no outward communication channels. Building on this this utility, we can divide the problem into two parts: (1) transmitting the singleton "native" format of the sender and (2) transmitting a set of confined converters that know how to translate from this native format to other formats that the client knows how to produce.

The main problem with transmitting the memory region containing the singleton native format is ***durability***. The memory region containing the native format material will be needed for an unbounded amount of time, and a recipient in a higher-level compartment is not permitted to inform the sender in a lower-level compartment that it is done with the data. Our solution is to require every sender to supply a constructor for initially empty, confined memory regions that are built from sender storage. The native format is serialized to this region, the region is frozen (to prevent further modification by either party), and a capability to it is transferred to the recipient. The recipient is hazarded by the fact that the sender can reclaim the storage at any time. A recipient wishing to retain the memory region for any length of time is therefore well-motivated to copy its content into a recipient-supplied memory region. In the current implementation, the sender can detect the deallocation of the memory region by the receiver and can use observation of deallocation latency for signalling purposes.[3]

Unfortunately, we cannot simply transfer a vector of constructor capabilities for the converter programs. While the display server could verify that each member capability is a leak-free constructor capability, the sender could subsequently alter some vector element to be a capability to be something else. Instead, we have the sender transmit a constructor to a single, confined conversion agent. The conversion agent can be asked for the set of formats it knows how to produce and can then be asked to produce each desired format in turn. This is most easily implemented by having each converter be a separately constructable utility application. A hidden advantage in this design is that the storage needed to perform the conversion is provided by the recipient rather than the sender. Note that all the constructors involved are created at the time the application is installed. No paste-time instantiation of converters is required.

The final cut and paste transfer protocol, including format negotiation, goes as follows:

1. The display server instantiates a new memory region using the region constructor supplied by the sender. It provides the resulting region capability to the sender.

2. The sender writes its native paste format to the new memory region and informs the display server when it has completed doing so. During this step, it also provides a capability to the converter constructor.

3. The display server now "freezes" the resulting memory region, preventing either sender or receiver from performing further modifications.

4. The display server now provides both the native memory region capability and the capability to the converter constructor to the recipient.

The resulting cut and paste interaction supports full format negotiation with no downward channel.

[3] To limit this hazard, we will shortly introduce a secure storage exchange operation by which ownership of the storage is transferred to the recipient at the time of the paste operation and the sender immediately sees their free resource pool restored. Secure resource interchange of this form is generically useful in many other circumstances.

7 User Interaction

Because the window system is the primary mediator of user input, there are certain operations users perform that it must assure. Most of these can be viewed as trusted path issues, and we will consider three here: title bars, window labeling, and pass phrase entry.

7.1 The Title Bar

The title bar problem is a problem of control: does "minimize" mean "inform the application that we would like to minimize", or does it mean "*tell* the application that we have minimized it?" Indeed, should we tell the application of such actions at all? The decision matters primarily because it determines who is responsible for rendering and interpreting the title bar. Our policy in EWS is that these functions are directives rather than requests, and in consequence that the display server must handle these functions. In the work reported here, title bar and border rendering are performed by the display server.

A second concern with the title bar is the problem of font forgery. If applications are permitted to set the title bar font, they are in a position to alter the information displayed. In EWS, title display is managed by the display server using a fixed, compiled-in font. In a production implementation, we would probably allow the user to select from a number of predefined fonts using a privileged application, but eliminating the need to render fonts within the display server provided a significant reduction of code.

7.2 Window Labeling

In a multilevel secure environment, window security labels are required, and the requirements specified for Compartmented Mode Workstations [33] are generally taken to be definitive. Unfortunately, these requirements are incomplete. There is no label that the display server can apply on a window border that cannot be visually forged by a client. Using alpha blending to "dim down" non-focus windows or identify trusted windows is insufficient: an application can implement a visibly indistinguishable child window and dim it's own primary window using the same algorithm.

The EWS display server defeats this attack by prominently featuring the border of the focus window using a bright color while dimming non-focus windows. A bright border color is chosen because dimming of darker colors using alpha blending is less easily noticed by the eye. Separately, the EWS display server reserves a band at the bottom of the display that is used to provide labeling feedback.

7.3 Pass Phrase Entry

Pass phrases present a particular challenge in a windowed environment. Because the input is inherently sensitive, it is important for the user to know that they are providing it to the intended application.

Because EROS is a capability system, many operations that initially appear to require trusted path interaction do not. For example, there is no need for a trusted path to support a trusted SaveAs agent. The protection in the SaveAs case devolves from the fact that only the SaveAs agent holds a capability to the user's file system. An application might forge the appearance of a SaveAs dialog, but cannot forge possession of the necessary file system capability.

When the "protection by guardianship" design pattern is widely applied, the only remaining requirements for trusted path interactions arise in three cases:

- Password prompts
- Cryptographic key pass phrases
- Login authentication

This list is small enough and specialized enough that it is reasonable to declare that these components must be trusted subsystems. A client application may independently instantiate many copies of the trusted password validator, but the interaction between client and validator is restricted: the client supplies a user name and the validator returns true or false depending on whether the user typed the correct password. Similarly, there may be many instantiations of our equivalent to Factotum [6], but none of these reveal decrypted cryptographic key bits to their client applications.

In the context of a capability-based system, it appears possible to impose the restriction that all trusted paths are connections between the display and a small number of trusted applications. If these applications are trusted, then in particular they can be trusted to identify themselves honestly. We have therefore resolved the trusted path problem in EWS by providing a distinguished "trusted client session" interface. A trusted client session is one whose client is a trusted application. It otherwise implements the same operations as a normal client session.

When a window associated with a trusted client session is active, all other windows are overlayed with a red alpha-blended overlay, and the reserved labeling region at the bottom of the display is distinctively marked.

8 Vulnerability Analysis

The vulnerability of the EROS Window System is drastically smaller than that of X11 or Trusted X as the result of four architectural decisions:

- The removal of general rendering responsibility from the display server. Our server implements only `bitblt` and `rectfill` operations, both of which have mature, well-tested implementations.
- The simplification of the event handling logic.
- The elimination of authentication and network communication responsibilities from the server.
- Our abandonment of the X11 communication model in favor of accountable, confined information transfer.

We suspect, but have not endeavoured to prove, that the covert channel bandwidth available through EWS is less than that of X11. There are clearly fewer points of implicit rendezvous, and generally reduced variance across EWS operations that might be exploited for timing measurement. The absence of server-side queueing also helps.

While these changes clearly reduce the vulnerability of the server, it is important to ask what new responsibilities have been imposed on clients that might have security implications. Clients now carry two blocks of content that were not required in the X11 design:

- A code library implementing rendering, which may be compromised.
- A font library, which is probably shared across multiple applications.

Our feeling is that the rendering library does not introduce a substantial new threat. Applications already depend extensively on widget libraries; the introduction of the rendering library into the build does not introduce any new problems that were not already present.

The font library is a greater concern, though fonts were not really protected under the X11 design either. We do not know of any technique capable of preventing font forgery by the font distributor. The EROS capability system provides sufficient protections that fonts cannot practically be modified after installation, and there are no display operations that allow one client to modify the fonts used by another.

The current EWS prototype *is* vulnerable to resource exhaustion. A hostile client could create enough windows to exhaust the virtual memory of the display server. Our plan for this is to restrict the total number of simultaneous windows (say, to 65,536), and reserve a subset of this for allocation by trusted applications. We can then construct a trusted usage reporting agent that would alert the user to this abuse and allow the user to destroy the offending application.

9 Usability

While a full usability test is beyond the scope of this paper, we did perform a very informal usability test using a paint program that we constructed as an early testing tool. Wesley Vanderburgh, age 4, created the drawing in Figure 5. The resulting figure was enhanced by his father for publication. Wesley is in many respects representative of potential end users for EWS. He is completely comfortable using the Microsoft system, largely impervious to training, and eager to get on to useful work without interruption or distraction – play time is valuable! While the image did take a while for Wesley to generate, our unbiased observer (his father) reported that this appears to be due to the immaturity of the test subject's fine motor functions rather than any deficiency in the window system. We note that this test is inconclusive, as four year olds exhibit considerably greater adaptability and flexibility than mature computer users.

On a more serious note, the window system described here has been used in presentations to DARPA without difficulty or noticeable interactive performance deficiencies. Our limiting factor in testing is the immaturity of the EROS runtime environment and the consequent difficulty of bringing up commonly used applications. A port of the Gtk graphics toolkit is currently in progress, which we hope will resolve this.

10 Related Work

Considering the importance of window systems in modern computing, there has been surprisingly little work on security in window systems. We have discussed throughout

Figure 5: Usability demonstration by young potential Picasso.

this paper the impact of Trusted X [9, 8] and the Compartmented Mode Workstation [4, 20, 21, 33] efforts.

A key decision in the design of EWS was the adoption of local shared memory to support our basic rendering model. This was encouraged by our experiences as early users of the Blit [22, 24] bitmapped terminal, and later by the architectural success of the Gnot (the original display for Plan 9 [23]). The success of these two systems convinced us that the argument for generic remoting advanced by Gettys and others is not compelling. Even in the absence of a cooperative display update protocol, bitmap propagation strategies such as those used by tools like VNC [25] do an excellent job of providing efficient display update while reducing the displays server's trusted computing base by an order of magnitude. The display update protocol used in EWS is actively VNC-friendly. For applications such as movie display or gaming where low latency is required, remote connections are unsatisfactory from a usability standpoint. In the movie case, there is also a substantial bandwidth (and therefore power) cost imposed by performing decompression before the bits arrive at the destination display. In short, generic remoting appears to be viable only in the cases where interactive performance does not matter.

While many other well-known window systems exist, most notably those of the Macintosh [2, 3], Microsoft Windows, and the Alto [31], none have given particular attention to the possibility of hostile applications.

An increasing number of applications today incorporate scripting languages or full programming languages. Among many others, Tygar and Whitten have identified several categories of vulnerabilities that can arise from such mechanisms [32]. Effective use of the EWS mechanisms in concert with the capability underpinnings of EROS eliminate many of these vulnerabilities.

Ka Ping Yee has considered various concerns in secure usability design [35]. Yee's work in this area has been strongly influenced by years of exposure to the EROS community and the E capability-based scripting language of Mark Miller. The reverse is also true; EWS contains elements that are included specifically to support some of the idioms proposed by Yee.

The PERSEUS project is attempting to provide security guarantees in the context of mobile devices that are comparable to those of the EROS project. Their architectural overview paper [19] provides an overview of both the design issues and some of the possible techniques that might serve as solutions. A challenge facing the PERSEUS project today is that they have implemented their prototype on top of the FIASCO kernel [13], which is an implementation of the experimental L4*x*2 architecture [16]. While acceptable for research purposes, this decision was problematic in a system that was created with the goal of ultimate commercial deployment: the L4 architecture did not (and does not) provide sufficient security at the microkernel level to be adequate for use in a secure system. This critique was raised by one of the authors at the time the PERSEUS project was first proposed, and has yet to be addressed. Recently, collaboration has started between the L4 community and the EROS community to identify and specify a next-generation secure L4 architecture. Among other systems, the PERSEUS project will clearly benefit from these revisions.

Following up on the web spoofing work by Felten *et al.* [11], Ye and Smith have examined the problem of trusted paths in browsers [34]. They examine various methods for displaying trusted path information to the user, and explore the pitfalls of each. This is an area that needs further exploration in EWS. Our work on EWS to date is largely complementary with the work of Ye and Smith. Where Ye and Smith focus on issues of *presentation*, we have focused on issues of *separation*. Our goal is to ensure that ordinary applications lack the necessary authority to disrupt the trusted path successfully, and to ensure that any hostility encountered in an application remains confined to that application.

10.1 DoPE

DoPE [12] is a window system created by the L4 team at T.U. Dresden for use in real-time systems. The real-time design environment presents many of the same constraints that arise in trusted window systems. While motivated by real-time predictability rather than security, the DoPE system must minimize variance, and in doing so, must apply resource minimization techniques that are comparable to those described here. Discussions between the two project teams revealed that the two systems are comparable in size and complexity when DoPE's rendering operations and higher-level widgets are excluded from the comparison (Table 1).

Feature	**EWS**	**DoPE**
Core function, drivers	4,500	7,000
Higher Widgets	N/A	3,000

Table 1: Comparative sizes of EWS and DoPE, in lines of code.

The total lines of code attributable to drivers are approximately equal in the two systems. EWS provides two hardware-dependent display drivers (VMWare and Rage-128). DoPE implements only one display driver (VESA), but the driver exports a richer set of rendering operations than the EWS drivers. Given this, we believe that the 2,500 line gap in the respective core functionality is primarily due to the inclusion of lower-level widgets in the DoPE display server design.

As originally conceived, EWS incorporated a similar widget system, but we are aware of no motivation from either a security or a performance perspective that requires this functionality to be implemented by the trusted computing base (TCB). In consequence, TCB minimization requirements therefore mandated moving this function to the client, and we never attempted a server-side widget implementation. If the above breakdown of function and complexity is correct, the security argument for *removing* widgets was clearly compelling: the complexity of the DoPE widget set (including lower and higher widgets) is approximately equal to the complexity of the entire EWS trusted computing base.

Both DoPE and EWS plan to incorporate support for 3D acceleration in future work. Hardware interactions of this kind are necessarily trusted, but from a security perspective, defensive engineering practice suggests that such function should be implemented by a separate protection domain. Software rendering routines intended to replace missing hardware functions should be implemented by the client rendering library, which is entirely outside of the trusted computing base; there is no reason to incorporate such function into the display server.

11 Acknowledgements

While it has diverged in recent years, the original EROS architecture was closely derived from that of KeyKOS. No work derived from KeyKOS could be complete without acknowledging the principal architects and implementors of that system: Norman Hardy, Charlie Landau, and William Frantz. Each of these individuals has participated in and encouraged work on the EROS system.

While this paper does not present X11 as a positive example for purposes of security, it is a system that contains many brilliant ideas that have had a strong influence on the work of the authors for many years. In large measure, X11 and its predecessors are responsible for the acceptance of bitmapped computer graphics today. Many years ago, Phil Karlton spent a fair bit of time explaining the design of X11 to Jonathan S. Shapiro while we worked on the Silicon Graphics ProDev tool set. More recently, Jim Gettys offered some of his rationale for the desirability of remote display access. Ultimately, though it was not his intention, he reaffirmed our view that remoting didn't belong in the window system.

Jeremy Epstein was gracious enough to review a draft of this paper, and offered a number of helpful comments.

We are extremely greatful to Norman Feske and Hermann Härtig of the Dresden L4 group for their time and courtesy in explaining the implementation and design concept of DoPE.

Wesley Vanderburgh graciously permitted us to reproduce Figure 5. Framed and signed lithographs may be obtained by request from the artist, who is struggling to pay for his first grade home-schooled education and would appreciate your support. Twenty years from now, they will surely be very rare and possibly more valuable than SCO stock.

12 Conclusion

We have presented the design of the EROS Trusted Window System, which provides robust traceability of user volition and is capable (with extension) of enforcing mandatory access controls. The EWS implementation, including the two current display drivers, is less than 4,500 lines, which is a factor of ten smaller than previous trusted window systems such as Trusted X, and well within the range of what can easily be evaluated for high assurance.

Based on our experience with both the implementation and the result, the EROS Window System is practical, usable and assurable. As is so often the case in asking how to secure subsystems, the key lay in deciding what to remove. What is staggering in this instance is that the trusted component of EWS is between 2% and 5% of the lines of code of X11 with no user-apparent reduction in functionality or utility. It can readily be extended to new input devices, and extension of this form would *not* entail a complex re-evaluation effort because input drivers are strongly isolated. Most of the work would lie in the associated device helper, which is isolated from the display server by a protection boundary.

While we have not attempted to tune the EWS implementation for performance, the evidence of the widely-used Apple Quartz 2D implementation suggests that final performance should be acceptable.

The small size of EWS provides a partial validation of the EROS design. A key idea in EROS is that breaking applications into small, protected components yields more secure applications and often allows smaller programs to perform very powerful functions by leveraging existing components.

Both EROS and the EROS window system implementation will be accessable via the EROS web site at the time of publication.

References

[1] M. D. Abrams. Renewed understanding of access control policies. In *Proc. 16th National Computer Security Conference*, pages 87–96, Oct. 1993.

[2] Apple Computer. *Inside Macintosh*. Reading, Massachusetts, 1985.

[3] Apple Computer. *Quartz 2D Reference*. Apple Computer, Inc., 2003.

[4] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings. Compartmented mode workstation: Prototype highlights. *IEEE Transactions on Software Engineering*, 16(6):608–618, June 1990.

[5] 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.

[6] R. Cox, E. Grosse, R. Pike, D. Presotto, and S. Quinlan. Security in plan 9. In *Proceedings of the 11th USENIX Security Symposium*, pages 3–16, San Francisco, 2002.

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

[8] J. Epstein, J. McHugh, H. Orman, R. Pascale, A. Marmor-Squires, B. Dancer, C. R. Martin, M. Branstad, G. Benson, and D. Rothnie. A high-assurance window system prototype. *Journal of Computer Security*, 2(2):159–190, 1993.

[9] J. Epstein and J. Picciotto. Trusting X: Issues in building Trusted X window systems -or- what's not trusted about X? In *Proceedings of the 14th Annual National Computer Security Conference*, Washington, DC, USA, Oct. 1991. A survey of the issues involved in building trusted X systems, especially of the multi-level secure variety.

[10] J. Epstein, et. al. A prototype B3 Trusted X Window System. In *Proceedings of the Seventh Annual Computer Security Applications Conference*, San Antonio, TX, USA, Dec. 1991. The architecture for TRW's high assurance multi-level secure X prototype.

[11] E. W. Felten, D. Balfanz, D. Dean, and D. S. Wallach. Web spoofing: An internet con game. In *20th National Information Systems Security Conference*, Baltimore, Maryland, Oct. 1997.

[12] N. Feske and H. Haertig. DOpE – a window server for real-time and embedded systems. In *Proc. 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, Dec. 2003.

[13] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proc. 2001 USENIX Annual Technical Conference*, pages 217–230, Boston, MA., 2001.

[14] *Common Criteria for Information Technology Security*. International Standards Organization, 1998. International Standard ISO/IS 15408, Final Committee Draft, version 2.0.

[15] M. J. Kilgard, D. Blythe, and D. Hohn. System support for openGL direct rendering. In W. A. Davis and P. Prusinkiewicz, editors, *Graphics Interface '95*, pages 116–127. Canadian Human-Computer Communications Society, 1995.

[16] L4 eXperimental reference manual, version X.2. Technical report, L4KA Team, University of Karlsruhe, 2001.

[17] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[18] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proc. IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[19] B. Pfitzmann, J. Riordan, C. Stüble, M. Waidner, and A. Weber. The PERSEUS system architecture. In D. Fox, M. Köhntopp, and A. Pfitzmann, editors, *VIS 2001, Sicherheight in komplexen IT-Infrastrukturen*, DuD Fachbeitrge, pages 1–18. Vieweg Verlag, 2001. Also available as IBM Research Report RZ 3335 (#93381).

[20] J. Picciotto. Towards trusted cut and paste in the X Window System. In *Proceedings of the Seventh Annual Computer Security Applications Conference*, San Antonio, TX, USA, Dec. 1991. A discussion of the security problems associated with cut and paste in multi-level secure versions of X.

[21] J. Picciotto and J. Epstein. A comparison of Trusted X security policies, architectures, and interoperability. In *Proceedings of the Eighth Annual Computer Security Applications Conference*, San Antonio, TX, USA, Dec. 1992. A survey of interoperability issues among CMWs and the TRW prototype.

[22] R. Pike. The blit: A multiplexed graphics terminal. *Bell Labs Tech. J.*, 63(8, part 2):1607–1631, Oct. 1984.

[23] R. Pike. $8\frac{1}{2}$, the plan 9 window system. In *Proceedings of the Summer 1991 USENIX Conference*, pages 257–265, Nashville, 1991.

[24] R. Pike, B. Locanthi, and J. Reiser. Hardware/software tradeoffs for bitmap graphics on the blit. *Software - Practice and Experience*, Jan. 1985.

[25] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[26] D. Rosenthal. *Inter-client Communications Conventions Manual, version 2.0*. X Consortium and Sun Microsystems, 1994.

[27] R. W. Scheiffler and J. Gettys. *X Window System*. Digital Press, 3rd edition, 1992.

[28] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification, version 1.0*. Silicon Graphics, Inc., 1993.

[29] 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.

[30] 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.

[31] C. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. Boggs. Alto: A personal computer. *ACM Transactions on Computer Systems*, 2(1), Feb. 1984.

[32] J. Tygar and A. Whitten. WWW electronic commerce and Java Trojan horses. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 243–250, Oakland, CA, 1996.

[33] J. P. L. Woodward. Security requirements for system high and compartmented mode workstations. Technical Report MTR 9992, Revision 1 (also published by the Defense Intelligence Agency as document DDS-2600-5502-87), The MITRE Corporation, Bedford, MA, USA, Nov. 1987. The original requirements for the CMW, including a description of what they expect for Trusted X.

[34] Z. E. Ye and S. Smith. Trusted paths for browsers. In *Proc. 11th USENIX Security Symposium*, pages 263–279, 2002.

[35] K.-P. Yee. User interaction design for secure systems. In *Proc. 4th International Conference on Information and Communications Security*, pages 278–290, Dec. 2002.