BAA-00-06-SNK

FOCUSED RESEARCH TOPIC 5

BY MARC STIEGLER AND MARK MILLER

Report Name:

A Capability Based Client: The DarpaBrowser

COMBEX INC.

CONTACTS

TECHNICAL MARC D. STIEGLER
PO Box 711
Kingman AZ 86402
Voice: (928) 718-0758
Cell Ph.: (928) 279-6869
Email: marcs@skyhunter.com Fax: 413-480-0352

ADMINISTRATIVE HENRY I. BOREEN
P.O. Box 4070
Rydal PA19046
Voice: 215-886-6459
Email: hboreen@comcast .net
Fax: 215-886-2020

DATE OF REPORT

26 June 2002

TABLE OF CONTENTS

A Capability Based Client: The DarpaBrowser

Executive Summary

The broad goal of this research was to assess whether capability-based security [Levy84] could achieve security goals that are unachievable with current traditional security technologies such as access control lists and firewalls. The specific goal of this research was to create an HTML browser that could use capability confinement on a collection of plug-replaceable, possibly malicious, rendering engines. In confining the renderer, the browser would ensure that a malicious renderer could do no harm either to the browser or to the underlying system.

Keeping an active software component such as an HTML renderer harmless while still allowing it to do its job is beyond the scope of what can be achieved by any other commercially available technology: Unix access control lists, firewalls, certificates, and even the Java Security Manager are all helpless in the face of this attack from deep inside the coarse perimeters that they guard. And though the confinement of a web browser's renderer might seem artificial, it is indeed an outstanding representative of several large classes of crucial security problems. The most direct example is the compound document as so well known in the Microsoft Office Suite: one can have a single Microsoft Word document that has embedded spreadsheets, pictures, and graphics, all driven by different computer programs from different vendors. Identical situations (from a security standpoint) arise when installing a plug-in (like RealVideo) into a web browser, or an Active-X control into a web page.

In such a compound document none of the programs need more than a handful of limited and specific authorities. None of them need the authority to manipulate the window elements outside their own contained areas. They absolutely do not need the authority to launch trojan horses, or read and sell the user's most confidential data to the highest bidder on EBay. Yet today we grant such authority as a matter of course, because we have no choice. Who can be surprised, with this as the most fundamental truth of computer security today, that thirteen year old children break into our most secure systems for an afternoon's entertainment?

To tackle the problem, Combex used the **E** programming language, an open source language specifically designed to support capability security in both local and distributed computing systems. We used **E** to build ***CapDesk***, a capability secure desktop, that would confine our new browser (the ***DarpaBrowser***) which would in turn use the same techniques to even more restrictively confine the renderer. Once we had completed draft versions of the CapDesk, DarpaBrowser, and Malicious Renderer, we

brought in an outside review team of high-profile/high-power security experts to review the source code and conduct live experiments and attacks on the system. The DarpaBrowser development team actively and enthusiastically assisted the review team in every way possible to maximize the number of security vulnerabilities that were identified.

The results can only be described as a significant success. We had anticipated that, during the security review, some number of vulnerabilities would be identified. We had anticipated that the bulk of these would be easy to fix. We had anticipated that a few of those vulnerabilities might be too difficult to fix in the time allotted for this single research effort, since the technology being used is still in a pre-production state. But more important than any of this, we had also predicted that no vulnerabilities would be found that could not be fixed straightforwardly inside the capability security paradigm. All these expectations, including the last one, were met. As stated by the external security review team in their concluding remarks on the DarpaBrowser:

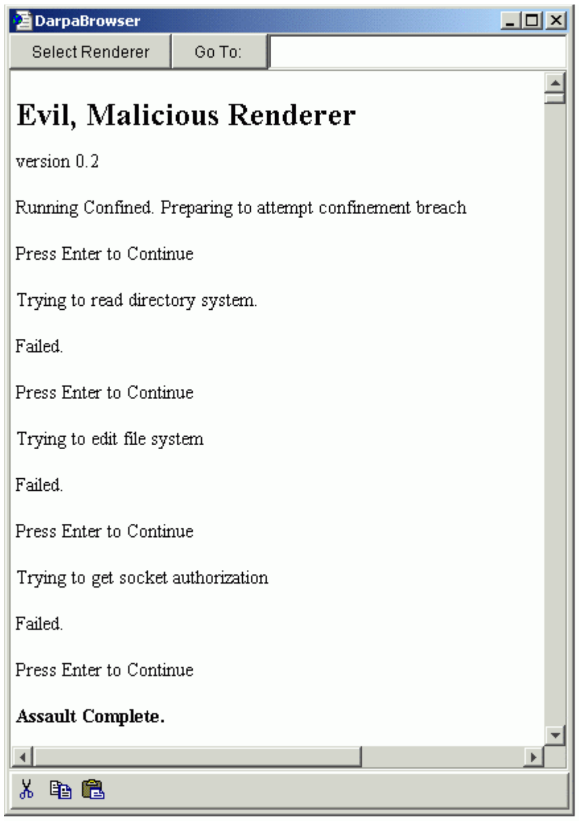
> *We wish to emphasize that the web browser exercise was a very difficult problem. It is at or beyond the state of the art in security, and solving it seems to require invention of new technology. If anything, the exercise seems to have been designed to answer the question: Where are the borders of what is achievable? The* **E** *capability architecture seems to be a promising way to stretch those borders beyond what was previously achievable, by making it easier to build security boundaries between mutually distrusting software components. In this sense, the experiment seems to be a real success. Many open questions remain, but we feel that the* **E** *capability architecture is a promising direction in computer security research and we hope it receives further attention.*

One of the by-products of this research, as a consequence of building the infrastructure needed to support the experiment, was the construction of a rudimentary prototype of a capability secure desktop, CapDesk. CapDesk and the DarpaBrowser with its malign renderer provide a vivid demonstration that the desktop computer can be made invulnerable to conventional computer viruses and trojan horses without sacrificing either usability or functionality.

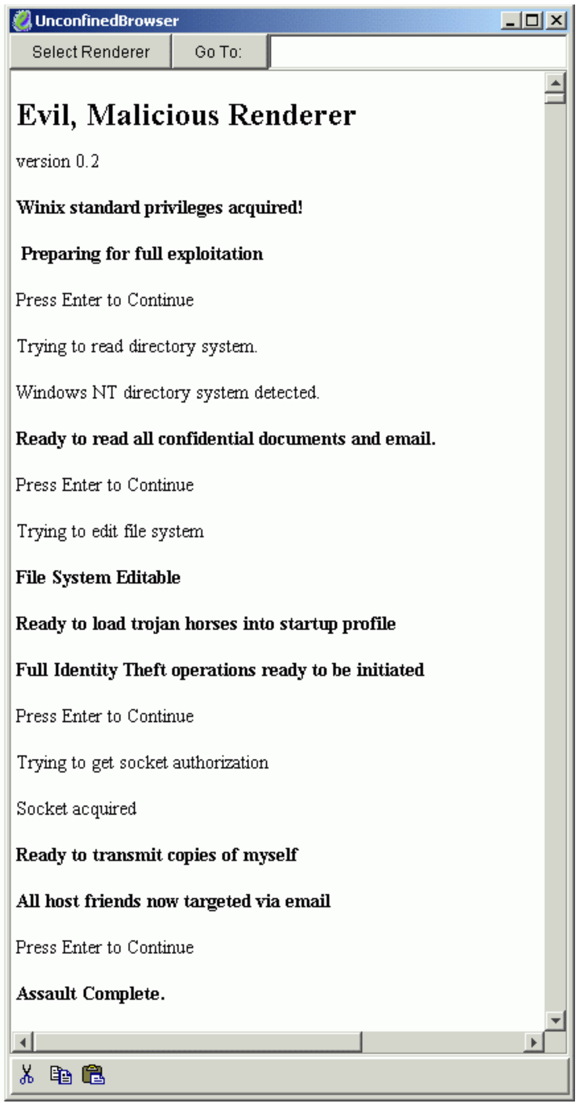
These results could have tremendous consequences. They give us at last a real hope that our nation--our industrial base, our military, and even our grandmothers and children--can reach a level of technology that allows them to use computers with minimal danger from either the thirteen year old script kiddie or the professional cracker. The Capability Secure Client points the way to a still-distant but now-possible future in which cyberterror appears only in Tom Clancy novels.

For a full decade, year after year, computer attackers have raced ever farther ahead of computer defenders. It is only human for us to conclude, if a problem has grown consistently worse for such a long period of time, that the problem is insoluble, and that the problem will plague our distant descendants a thousand years from now. But it is not true. We already know--and this project has begun to demonstrate--that capability security can turn this tide decisively in favor of the defender. The largest question remaining is, do we care enough to try.

We end this executive summary with a picture we believe to be more eloquent than any words. On the right is the world of computer security as it exists today. On the left is the world of computer security as it can be. You, the reader of this document, are now on the front line in the making of the choice between these two worlds.



```
Evil, Malicious Renderer
version 0.2
Running Confined. Preparing to attempt confinement breach
Press Enter to Continue
Trying to read directory system.
Failed.
Press Enter to Continue
Trying to edit file system
Failed.
Press Enter to Continue
Trying to get socket authorization
Failed.
Press Enter to Continue
Assault Complete.
```



```
Evil, Malicious Renderer
version 0.2
Winix standard privileges acquired!
 Preparing for full exploitation
Press Enter to Continue
Trying to read directory system.
Windows NT directory system detected.
Ready to read all confidential documents and email.
Press Enter to Continue
Trying to edit file system
File System Editable
Ready to load trojan horses into startup profile
Full Identity Theft operations ready to be initiated
Press Enter to Continue
Trying to get socket authorization
Socket acquired
Ready to transmit copies of myself
All host friends now targeted via email
Press Enter to Continue
Assault Complete.
```

The Malicious Renderer running in two different environments. On the left, the renderer is running under capability confinement. It is unable to achieve any compromise of the security either of the browser that uses the renderer, or the underlying operating system. On the right, the exact same renderer is running unconfined, with all the authorities any executing module receives by default under either Unix or WinNT (referred to as "Winix" here). With Winix authorities, the renderer takes full control of the user's computing resources and data.

DarpaBrowser Capability Security Experiment

Introduction

Goals

A precise description of the goals can be found in the Project Plan in the Appendix. The driving motivation was to show that capability based security can achieve significant security goals beyond the reach of conventional security paradigms.

What is wrong with conventional security paradigms? They all impose security at too coarse a granularity to engage the problem. This failure is acutely visible when considering the issues explored in this research, in which we must protect a web browser and its underlying operating environment from the browser's possibly-malicious renderer. Even though the browser must grant the renderer enough authority to write on a part of the browser's own window, the browser absolutely must not allow the renderer to write beyond those bounds. This is a classic situation in which ***POLA*** is required. Let us discuss the POLA concept, and then look at a number of conventional security technologies to see how they fail to implement POLA at all, much less assist in this sophisticated context.

Principle of Least Authority (POLA)

The shield at the heart of capability confinement is the Principle of Least Authority, or POLA (introduced in [Saltzer75] as "The Principle of Least Privilege"). The POLA principle is thousands of years old, and quite intuitive: never give a person (or a computing object) more authority than it needs to do its job. When you walk into a QuickMart for a gallon of milk, do you hand the clerk your wallet and say, "take what you need, give me the rest back"? Of course not. When you hand the clerk exact change, you are following the POLA principle. When you hand someone the valet key for your car, rather than the normal key, you are again following POLA. Our computers are ludicrously unable to enforce POLA. When you launch any application--be it a $5000 version of AutoCAD fresh from the box or the Christmas Elf Bowling game downloaded from an unknown site on the Web--that application is immediately and automatically endowed with all the authority you yourself hold. Such applications can plant trojans as part of your startup profile, read all your email, transmit themselves to everyone in your address book using your name, and can connect via TCP/IP to their remote masters for further instruction. This is, candidly, madness. It is no wonder that, with this as an operating principle so fundamental no one even dares think of alternatives, cyberattack seems intractably difficult to prevent.

Failures Of Traditional Security Technologies

- **Firewalls:** Firewalls are inherently unable to implement POLA. They are perimeter security systems only (though the perimeter may be applied to a single computer). Once an infection has breached the perimeter, all the materials and machines within the perimeter are immediately open to convenient assault. A firewall cannot discriminate trust boundaries between separate applications; indeed, it doesn't even know there are different applications, much less different modules inside those applications. To the firewall, all the applications are one big happy family.

- **Access Control Lists:** Access Control Lists (ACLs) as provided by Unix and WinNT are also inherently unable to implement POLA. It is possible, with sufficient system admin convolutions, to cast certain applications into their own private user spaces (web servers are frequently treated as separate users), but this is far too complicated for the user of a word processor. It fundamentally can't discriminate between applications within a user's space, much less discriminate components inside an application (such as document-embedded viruses written with an embedded programming language like Visual Basic for Applications, or an Active-X control or Netscape plug-in for a Web Browser).

- **Certificates:** Of all the currently popular proposals for securing computers, certificates that authenticate the authors of the software are most pernicious and dangerous. Certificates do not protect you from cyberviruses embedded in the software. Rather, they lull you into a false sense of security that encourages you to go ahead and grant inappropriate authority to software that is still not trustworthy. In the year 2000, an employee at Microsoft embedded a Trojan horse into one of the DLLs in Microsoft FrontPage. Microsoft asserted that they had had nothing to do with it, and started a search for the employee who had engaged in this unauthorized attack. What difference does it make whether Microsoft had anything to do with it or not? Microsoft authenticated it. The user's computer was just as subverted, regardless of who put it there. The real problem is, once again, the absence of POLA. FrontPage didn't need the authority to rewrite the operating system and install Trojan horses and should not have had such authority. The presence of a Trojan horse in FrontPage would, in any sensible POLA-based system, be irrelevant, because the attack would be impotent. There are uses for certificates in a POLA-based world, but this is not one of them.

- **Java Security Manager:** Of the collection of current security technologies, the Java Security Manager comes closest to being useful. The Java Security Manager wraps a single application, not a user account (like ACLs) or a system of computers (like firewalls). By using sufficiently cunning acts of software sleight-of-hand, one can even place different software modules inside different trust realms (though ubiquitously moving individual software objects into individual trust domains and handing out POLA authorities at that fine-grain level of detail would be too unwieldy, and too complex, to have understandable security properties). However, even the Java Security Manager does not begin to implement POLA. There are a handful of extremely powerful authorities that the Manager is able to selectively deny to the application. But authorities that are not self-evidently sledge-hammers, such as the authority to navigate around an entire window once you've gotten access to any individual panel, are far too subtle for the Java Security Manager to grasp. A malicious renderer could, with only a couple simple lines of code, take control of the browser's entire user interface, and gleefully spoof the user again and again, faster than the eye can blink.

As we see, none of these technologies is able to seriously come to grips with the crucial needs of POLA either separately or in combination. This is why computer security is in such a ghastly state of disrepair today. The answer is capabilities.

Capability Security

Capabilities are a natural and intuitive technology for implementing POLA. Capabilities can be thought of as keys in the physical sense; if you hand someone a key, you are in a single act designating both the object for which authority is being conveyed (the object that has the lock), and the authority itself (the ability to open the lock). Capabilities, like idealized physical keys, can only be gotten from someone who already has the key (i.e., the shape of the key cannot be successfully guessed, and a lockpick cannot figure it out either). Authority is delegated by handing someone a copy of the key. Revocation of a capability is comparable to changing the lock (though changing software locks is much easier than changing hardware). One might think that making individual keys for every object that could possibly convey authority in a computer program would be unwieldy. But in practice, since the authority is being conveyed by the reference (designation) to the object itself, and since you have to hand out the reference anyway for someone to use the object, it turns out to be quite simple. In most circumstances the conveyance of authority with the reference makes the authority transfer invisible, and "free". Finally, because capabilities naturally ubiquitously implement POLA, an emergent property of capability-confined software is ***defense-in-depth***: acquisition of one capability does not in general open up a set of exploits through which additional capabilities can be acquired. The implementation of capabilities that is today closest to production deployment is the **E** Programming Platform.

E **Programming Platform**

The **E** Programming Platform, at the heart of which lies the **E** programming language, is specifically designed to support capability security in both local and distributed contexts. **E** is an open source system [Raymond99], and the draft book ***E*** *in a Walnut* [Stiegler00] is the principal guide for practical **E** programming. **E** is still a work in progress. Indeed, the bulk of the work done to complete the

DarpaBrowser project was fleshing out parts of the **E** architecture needed by the browser environment. But operational software has been written and deployed with **E**. **E** is not yet feature complete, but the features it does have are robust.

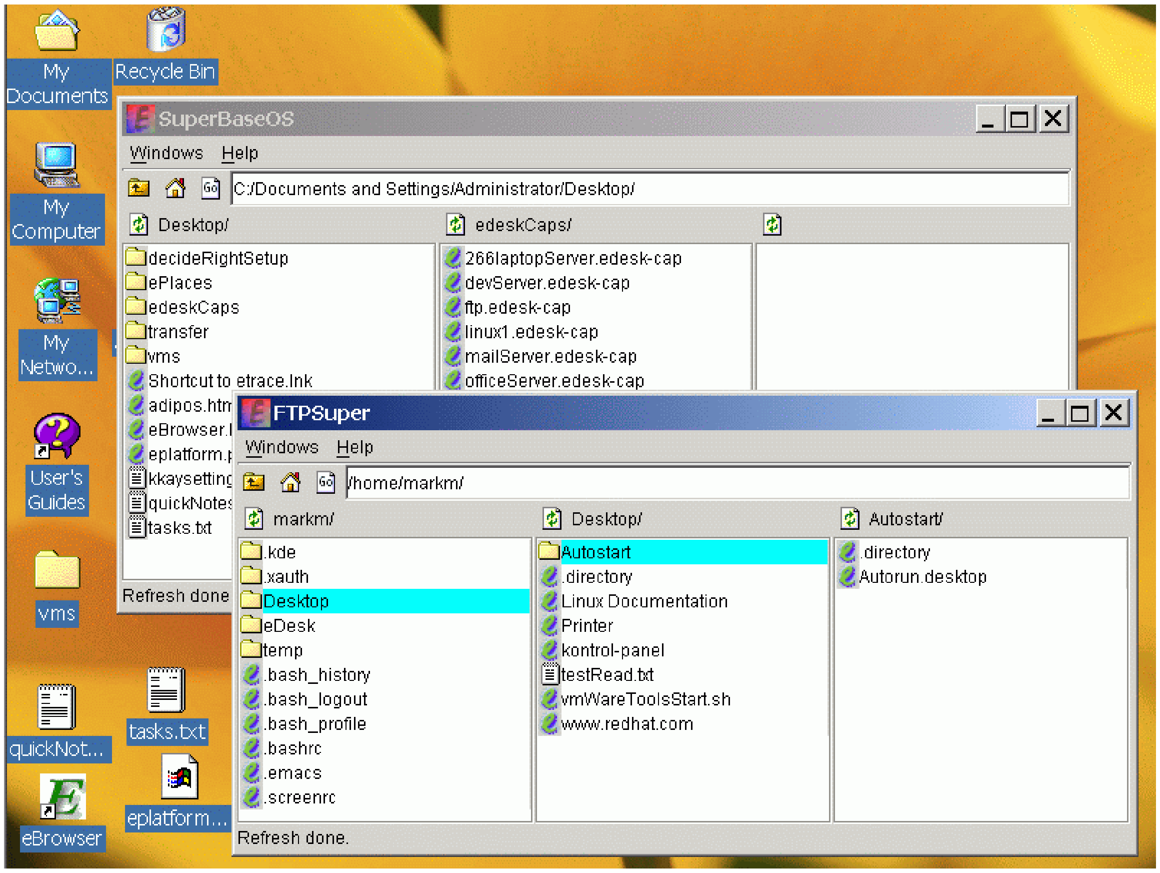
Preparation for the Experiment

Building CapDesk

We started with the eDesk point-and-click, capability secure distributed file manager that was the first serious **E** program used for production operations. We augmented eDesk with a rudimentary capability-confining launching system adequate for launching basic document-processing applications and a basic Web Browser. Adding the launching system turned eDesk into a rudimentary capability secure desktop, i.e., CapDesk.

Key components of the launching system include

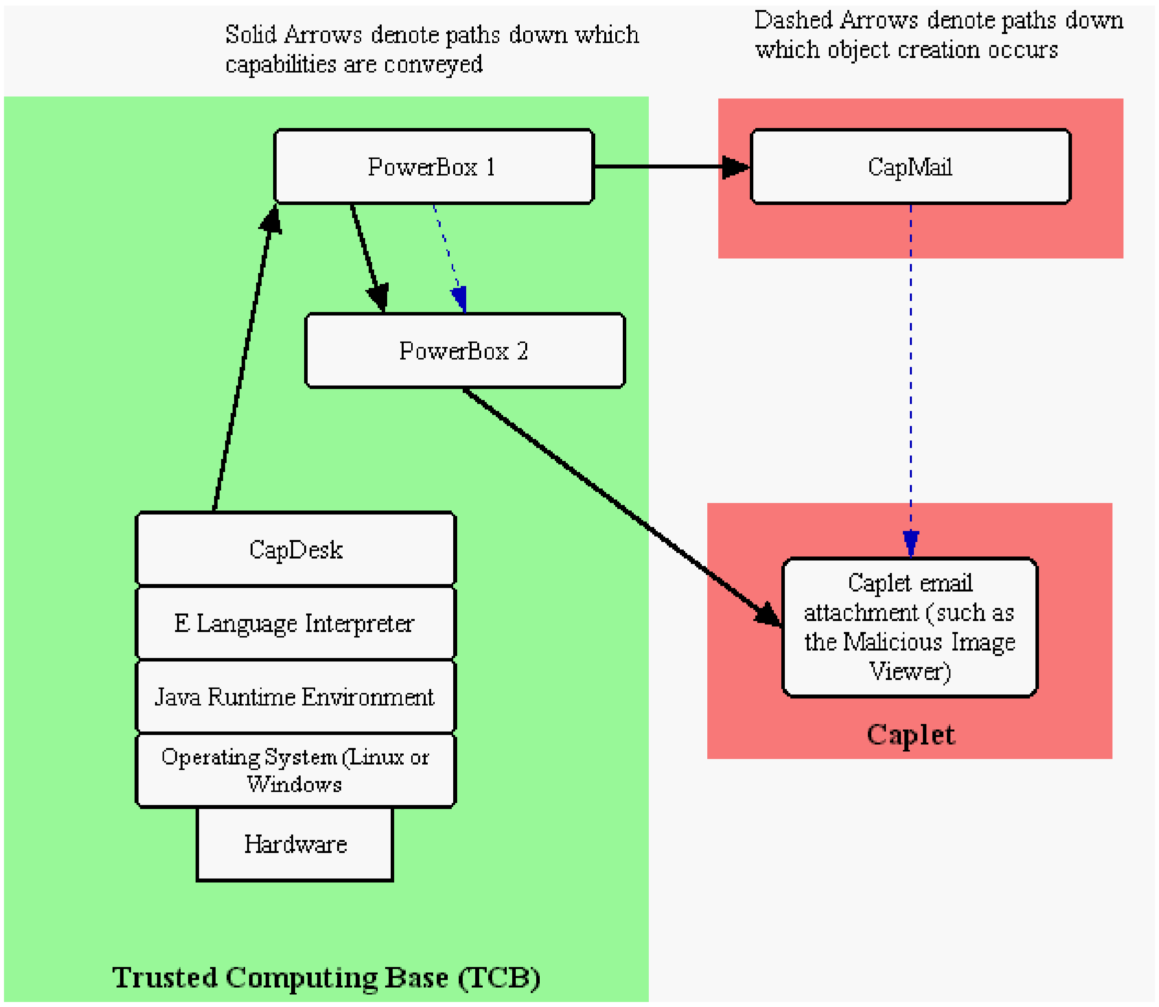
- Point-and-click installation module that negotiates *endowments* on the behalf of the confined application. An endowment is an authority automatically granted to the application at launch time. Basic document processors only need user-agreed upon name and icon for their windows, used to prevent window forgery. Web browsers work most simply if endowed with network protocols, specifically, in this case, HTTP protocol.
- Powerbox module that manages authority grant and revocation on behalf of a confined application. The powerbox launches the app, conveys the authorities endowed at installation, and negotiates with the user on the application's behalf for additional authorities during execution.



CapDesk operating as a capability secure, point-and-click, distributed file manager. In this example, CapDesk has one window open on a Windows computer, and another window open on a Linux computer. The user can copy/paste files and folders back and forth between the file systems, and directly run capability confined applications (such as CapEdit) on files on any part of a file system to which CapDesk has been granted a capability.

Building the E Language Machine

With CapDesk expanded to be able to launch capability confined applications, it became possible to build an **E** Language Machine (ELM). An ELM is the world's first general-purpose point-and-click computing platform that is capability-secure and invulnerable to traditional computer viruses and trojan horses. It is built by running a CapDesk on top of a Linux core; the CapDesk effectively seals off the underlying layers of software (Linux, Java, and **E** Language, which together with CapDesk comprise the Trusted Computing Base (TCB) which has full authority over the system) from the applications that are running (only capability confined applications can be launched from CapDesk, so only capability confined apps can run).


Solid Arrows denote paths down which capabilities are conveyed
Dashed Arrows denote paths down which object creation occurs
PowerBox 1
CapMail
PowerBox 2
CapDesk
E Language Interpreter
Java Runtime Environment
Operating System (Linux or Windows
Hardware
Caplet email attachment (such as the Malicious Image Viewer)
Caplet
Trusted Computing Base (TCB)

Architecture of the *E* Language Machine. A Linux kernel launches a Java virtual machine, which launches an *E* Language Interpreter, which launches a CapDesk. CapDesk seals off the underlying non-capability elements of the Trusted Computing Base from capability-confined applications launched from CapDesk. In this sample scenario, a hypothetical capability-confined mail tool is running, and the user has double-clicked on an email attachment. The attachment in run in its own unique trust/authority realm, making it unable to engage in the usual practices of computer viruses, i.e., reading the mail address file and connecting to the Internet to send copies of itself to the user's friends.

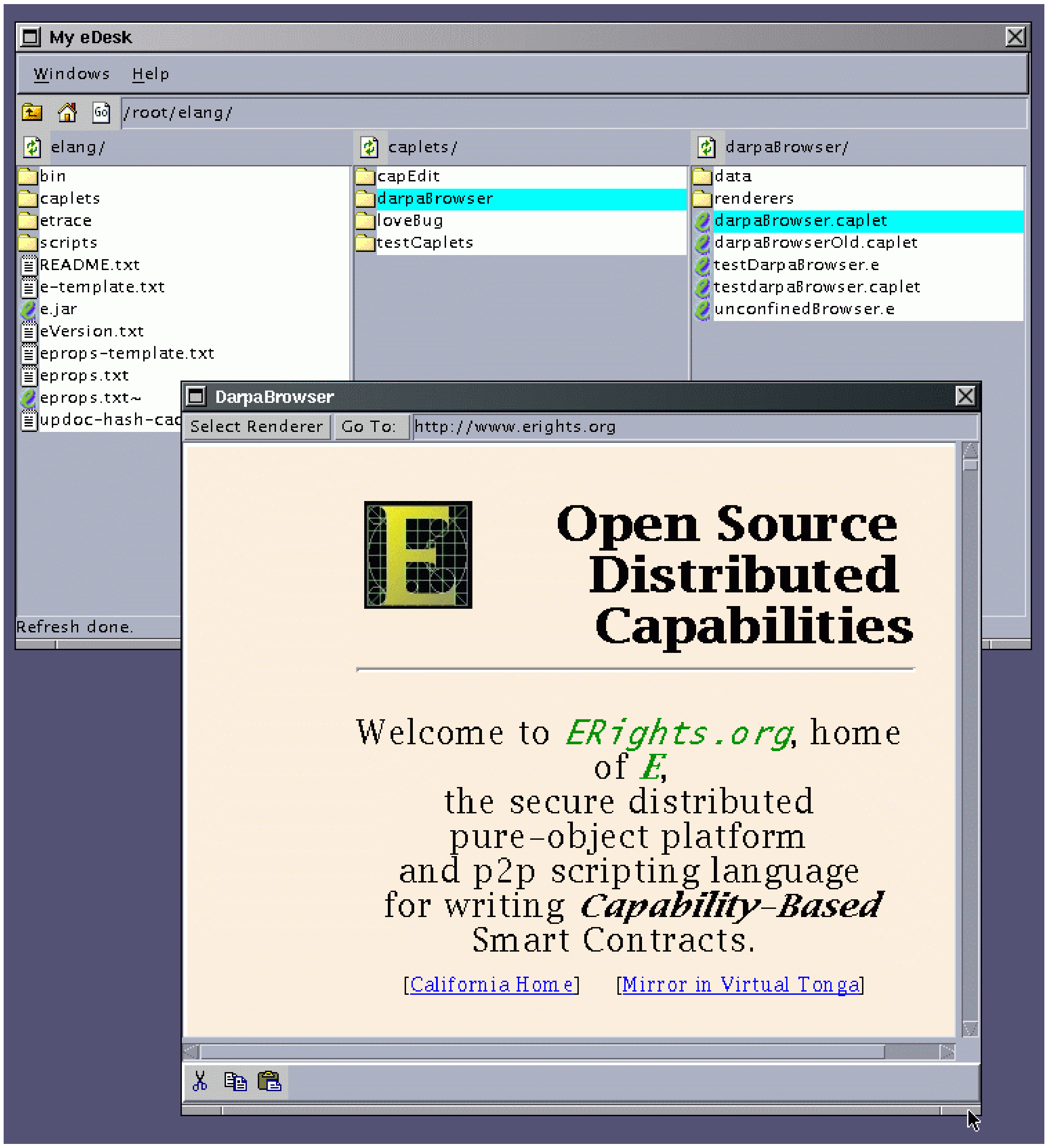
Building the DarpaBrowser

A web browser needs surprisingly few authorities considering the amount of value it supplies to its users. A simple browser needs little more than the authority to talk HTTP protocol. Since the DarpaBrowser was itself designed as a capability confined application, this meant that the browser never had very much authority available for the renderer to steal if the renderer somehow managed to totally subvert the browser. As later observed by the security review team in their report,

> *Withholding capabilities from the CapBrower is doing it a favor: reducing the powers of the CapBrower means that the CapBrowser cannot accidentally pass on those powers to the renderer, even if there are bugs in the implementation of the CapBrowser.*

One of the security goals was to prevent the renderer from displaying any page other than the current page specified by the browser. While capability confinement was trivially able to prevent the browser from going out and getting URLs of its own choice, there was one avenue of page acquisition that required slightly more sophistication to turn off: if the renderer were allowed to have a memory, it could show information from a previously-displayed page instead of the new one. Therefore, the renderer had to be made "memoryless".

With capabilities as embodied in E, one straightforward way of achieving memorylessness is to throw the renderer away and create a new one each time the user moves to a new page. This was the strategy used in the DarpaBrowser.



/root/elang/

elang/
bin
caplets
etrace
scripts
README.txt
e-template.txt
e.jar
eVersion.txt
eprops-template.txt
eprops.txt
eprops.txt~
updoc-hash-cac

caplets/
capEdit
darpaBrowser
loveBug
testCaplets

darpaBrowser/
data
renderers
darpaBrowser.caplet
darpaBrowserOld.caplet
testDarpaBrowser.e
testdarpaBrowser.caplet
unconfinedBrowser.e

Refresh done.

http://www.erights.org

Open Source Distributed Capabilities

Welcome to ERights.org, home of E, the secure distributed pure-object platform and p2p scripting language for writing *Capability-Based* Smart Contracts.

[California Home] [Mirror in Virtual Tonga]

An *E*LM with CapDesk and DarpaBrowser. The DarpaBrowser in this image is running the Benign Renderer, based on the JEditorPane widget of the Java Swing Library; this widget directly renders HTML.

Building the Renderers

According to the proposal originally presented for the DarpaBrowser, only two renderers would be built: a benign renderer, and a malicious renderer. As the project proceeded, it became clear that these were inadequate to test all the principles we wished to assess. A "text renderer" was built in time for the experiment. This renderer simply presents the text of the page on the screen, creating a "source view". This renderer was able to present any HTML document, no matter how badly the HTML has been written by the page author.

In the aftermath of the review, an additional renderer was built, the CapTreeMemless renderer, as described later.

Taming Swing/AWT

A key part of the effort of preparing for the experiment was ***taming*** the Java API, in particular, taming the AWT and Swing user interface toolkits. The act of taming involves applying a thin surface to a non-capability API that drives interactions into a capability-disciplined model. The Java API is not designed for capability security, yet contains an enormous amount of valuable functionality that cannot be easily rewritten from scratch. It turned out that the taming approach was in general adequate to make this API useable. Frequently, taming involves nothing more than suppressing "convenience" methods, i.e., methods that convey authority that programmers already have. Let us give a simple and a complex example:

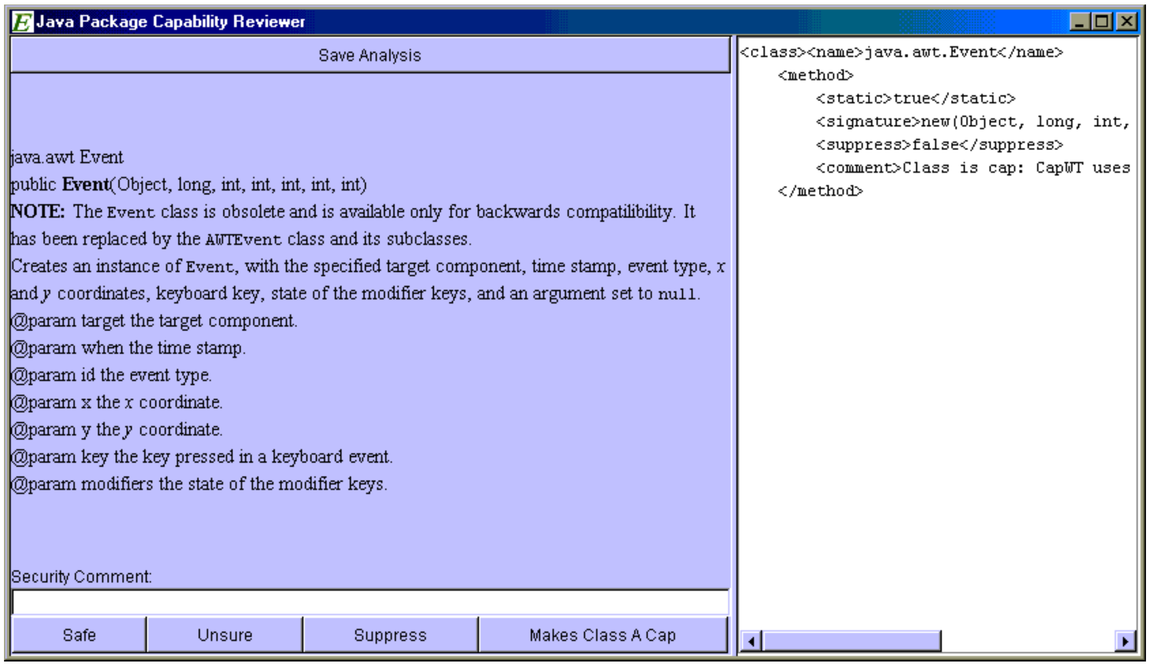
As a simple example, given any Swing user interface widget, one can recursively invoke "getParent" on the widget and its ancestors until a handle on the entire window frame is acquired. A malicious renderer could defeat, in a half dozen lines of code, the explicit goal of ensuring that the URL field and the page being displayed were in sync. Therefore the getParent method on the Component class must be suppressed to follow capability discipline.

As a complex example, the most ridiculous anti-capability subsystem we encountered in Swing was the Keymap architecture for JTextComponents. All JTextComponents in a java virtual machine share a single global root Keymap. Programmers can create their own local keymaps and add them as descendents from the global root Keymap, creating a tree of keymaps; the children receive and process keystrokes first, and can discard the keystroke before it reaches its parents. The part of this that is exquisitely awful is that it is possible to edit the global root keymap. Malicious software can trivially vandalize keystroke interpretation for the entire system. Even more maliciously, objects can eavesdrop on every keystroke, including every password and every confidential sentence that is typed by the user. This is not merely a violation of security confinement. This is a violation of the simplest precepts of object-oriented modularity. Not only is it trivially easy for malicious code to attack the system, it is trivially easy for the conscientious programmer to destroy the system by accident. Indeed, the way the DarpaBrowser team first identified this particular security vulnerability in Swing (long before initiating the actual taming process) was by accidentally destroying the keymap for the eBrowser software development environment.

To solve this problem, the methods that allowed the root keymap to be accessed had to be suppressed, and since child keymaps could not be integrated into the system without designating the parent, new constructors had to be created for keymaps that would, behind-the-scenes, attach the keymap to the root keymap if no other parent were specified. While the actual amount of code needed to tame this subsystem was small, more effort was needed to design the taming mechanism than would have been required by the JavaSoft Swing developers to create a minimally sensible object-oriented design in the first place.

Fortunately, the bulk of Swing is well-designed from an object-oriented perspective, which is what made it possible for the taming strategy to work well. Had the keymap subsystem not been an aberration, rewriting the user interface toolkit from lower-level primitives would have been more cost-effective despite the enormous costs such an undertaking would have entailed.

The AWT/Swing API is an enormous bundle of classes and methods. A substantial portion of the entire research effort went into this taming effort, including writing two versions of the CapAnalyzer (see picture) tool to support the human tamer in his efforts. The approach taken in this first attempt at taming was to be conservative, i.e., to shut off everything that might have a security risk associated with it, and enable only things that were well understood.



java.awt Event
public **Event**(Object, long, int, int, int, int, int)
NOTE: The `Event` class is obsolete and is available only for backwards compatilibility. It has been replaced by the `AWTEvent` class and its subclasses.
Creates an instance of `Event`, with the specified target component, time stamp, event type, *x* and *y* coordinates, keyboard key, state of the modifier keys, and an argument set to `null`.
@param target the target component.
@param when the time stamp.
@param id the event type.
@param x the *x* coordinate.
@param y the *y* coordinate.
@param key the key pressed in a keyboard event.
@param modifiers the state of the modifier keys.

```
<class><name>java.awt.Event</name>
    <method>
        <static>true</static>
        <signature>new(Object, long, int,
        <suppress>false</suppress>
        <comment>Class is cap: CapWT uses
    </method>
```

Version 1 of the *E* Capability Analyzer. The human analyst is walked through all the classes in a Java package, given the Javadoc from the API for that class, and allowed to individually suppress methods, and mark the class "safe" (i.e., it confers no authority), or "unsafe" (and must be explicitly granted to a confined module from another module that has the authority).

Limitations On the Approach

Before plunging into the experiment and its results, we should carefully note the limitations on the approach taken in the DarpaBrowser effort.

- **Goal Limitations:** As explicitly noted in the original proposal, Denial of Service attacks were out of scope, as were information leaks. The goal was to prevent the renderer from gaining authorities, such as the power to reset the clock or delete files. While as a practical matter the most dangerous kinds of information leaks were also prevented by capability confinement, some types of leaks, such as covert channels, were not even challenged, much less prevented. For more information about the details about the goals, see the Project Plan in the Appendix.

- **ELM Limitations:** The ELM is the world's first capability secure computing platform with a point-and-click user interface. It is a remarkable by-product of building the DarpaBrowser. We would be remiss not to note, however, that it has significant disadvantages compared to a true capability-secure operating system. Notably, the TCB is extremely large. The size of the TCB attracts risk of embodying security vulnerabilities. Furthermore, the architectural complexity of this TCB probably makes it too ungainly ever to pass a full security audit. Despite these limitations, however, the ELM still represents a substantial leap forward in the integration of security, flexibility, and usability.

- **Benign Renderer Limitations:** During the last days leading up to the experiment, it became clear that the benign renderer in particular was a weak experimental platform. This renderer was built, as specified in the proposal, using the Swing HTML widget. This gave us a professional-looking rendering at extremely little cost. One disappointment was that this HTML widget was extremely fickle about the HTML it would accept; as a consequence, very few pages on the Web will actually render through it.

  However, for our experimental purposes, this was not the major failing. More serious was that this widget used authority conveyed to it as a part of the Trusted Computing Base for much of its interaction with web pages. Consequently, the benign renderer exercised almost none of the authority confinement elements of the Browser: being a TCB HTML widget, it would just go out to the Web and get its own images based on the textual string of the URL, for example, without having to negotiate with the browser for actual authority. This was the reason we built the text renderer, as a proof that "real" rendering could be done without using TCB powers.

The Experiment

A snapshot was taken of the E/CapDesk system to use as a baseline for the security review. This baseline version of the system can be found at **http://www.erights.org/download/0-8-12/index.html**

The reviewers were given documentation on the system, goals, and infrastructure well in advance of the actual review, so that they could arrive with a reasonable familiarity and start fast.

For five consecutive days, the review team and the development team ate, slept, and drank DarpaBrowser security. On the first day the overall architecture was reviewed, and the schedule was made out for all the successive days of the review. The schedule ensured that no pieces of the system with signficant security sensitivities were excluded. Actual selection of what to review, how to review it, and how long to take reviewing it, were strictly driven by the reviewers; the development team assisted in every way possible, but they were only there to assist. Everyone took notes, and those notes were merged on the final day.

After the week of in-depth scrutiny, the review team wrote the security report that can be found in the Appendix.

Results

The written security review can be read in the Appendix. Anyone interested in the details of the results is encouraged to read the full report. To be extremely brief, the results were in line with our expectations: For the security goals specified in the project plan (which included goals from the original DARPA solicitation and our original proposal), we did find a number of security vulnerabilities (twenty-one in total) in our first implementation of the DarpaBrowser. Most of these vulnerabilities were simple programming errors that were easy to correct. Two of them have proven to be too hard/too unimportant/too irrelevant to the future development path of the E platform to fix within the limits of this contract; these two are described in detail the Post Experiment Development section below. However, not even the two unfixed vulnerabilities expose flaws in the fundamental capability security architecture. All can be straightforwardly repaired once the capability paradigm is embraced.

The crucial outcome of the experiment was of course the lessons learned, which are detailed in the next section. But for completeness' sake, we mention here an issue found in the memoryless version of the DarpaBrowser. By toggling the "Allow Memory" box on and off while browsing the Web, it becomes clear that the browser suffers a significant performance penalty when the renderer is made memoryless by creating a new renderer instance each time a new page is loaded. The performance analysis tools available with E at this time are too crude for us to state a conclusive explanation for this. We hypothesize that the problem lies in the way Java Swing discards and replaces subpanels. Regardless of whether the problem lies in Swing or in the current implementation of E, however, there is no reason that this should be an expensive operation, i.e., there is nothing about the capability paradigm that imposes a significant performance penalty for discarding objects, either within or across a trust boundary. We therefore do not say anything more about this discovery in this report, though it is another reason why we are eager to move from Swing to SWT, as described later.

Lessons Learned

General Truths

- **Acquisition of Dangerous/Inappropriate Authority Can Be Prevented by Capability Based Security.** All the vulnerabilities found in the experiment can be easily remedied within the capability architecture. Indeed, it can be argued that these vulnerabilities can *only* be remedied in the capability architecture, as suggested by the second general truth:

- **Leaving the capability paradigm invites grave security risks.** One of the two significant sources of vulnerabilities in the DarpaBrowser itself was the code for analyzing HTML. HTML is a simple text format that embodies implicit authority demands (for more detail, see the section below about HTML and the Confused Deputy problem). As a consequence, to deal with HTML--or to deal with any of the many other non-capability protocols now in use on the Web--you must depart from the capability paradigm long enough to process the protocol into a capability form. Through the DarpaBrowser investigation, we have learned just how important it is to enforce the following rules:

    - When you must leave the capability paradigm, get back as quickly (with as few lines of code) as possible.
    - Use the most rigorous techniques available for managing non-capability representations of authority.
    - Model the other forms of authority as capabilities whenever possible.

  In the DarpaBrowser, the strings embedded in the HTML that represent URLs were initially identified by using string matching. As discussed at length in the security review, this proved far too vulnerable. In the end an HTML parser was substituted for the string matcher, essentially eliminating this issue.

- **Taming a large API takes substantial resources.** The amount of time and effort required to tame Java AWT/Swing was significantly underestimated. Even taking the conservative approach described earlier (to shut off parts of the API that were not well understood), the speed with which taming was performed was too great, introducing vulnerabilities. The security review team concluded that taming was the most significant overall risk to the E capability implementation strategy, and we concur. A whole new tool should be written to support taming, and considerably greater resources must be committed to complete the taming process in such a fashion that the security community can feel confident in the result. A draft specification for the taming support tool can be found in the Appendix**.**

Capability confinement can significantly improve the cost/benefit ratio of security reviews. By following the flow of authority down capability references, even without any tool except the human eye, one can quickly identify large sections of code that cannot possibly have dangerous authority and do not need security review. The ability to cut off the review at the point where capabilities ceased flowing appeared repeatedly in the course of the experiment as a substantial time savings. This strength of capabilities was highlighted by the use of the powerbox pattern discussed below, though this was far from the only place in which the technique played a powerful role.

Significant opportunities for research in capability-based security patterns still exist. Capability based security has been known to the computer security field for decades as chronicled in [Levy84]; an update of Levy's chronology of capability milestones can be found in the Appendices. However, a relatively small percentage of the resources spent in computer security have been invested in the capability paradigm. As a consequence, within the capability field lie rich veins of security innovations still waiting to be mined.

Capability-Based Secure Programming is, with a few key exceptions, little different from Object-Oriented Programming. As suggested by the JTextComponent Keymap example described in the Taming section earlier, capability secure designs have a great deal in common with clean, modular object-oriented designs. Often a clean modular design is all one needs to secure a subsystem; the same minimization of object reference that reduces risk of accident and simplifies maintenance also implements much of POLA. Ironically, one of the differences between objects and capabilities is that, for capabilities, one must be more rigorous about applying object-oriented modularity: while the non-security-aware programmer can trade off modularity against other goals (even if the other goals are bad, as exemplified by the keymap), the capability-secure programmer needs to enforce modularity discipline pervasively. This has a number of specific consequences, described in detail in the Specific Insights section below. With those exceptions, however, the object oriented programmer will find little difference between object-oriented programming and capability-based programming when using a capability-based language such as E.

Within a capability-confined realm, even horrifically poor, security-oblivious programming can do little or no harm. This is one of the lessons of the confinement of the malicious renderer. Even if every bit of the architecture and implementation of the malicious renderer abandoned both capability and object-oriented design, little harm could come of it. The worst it could do is render the HTML poorly, i.e., it could be broken. It could not, however, harm the DarpaBrowser or the underlying system. The designer of the interface from the browser to the renderer needs to have skill as a capability-oriented programmer, but programmers without any special training can write the bulk of the code in a typical (secure) system.

Specific Insights

- **The Powerbox Pattern is a significant new invention**. The powerbox mediates authority grants for the confined module on behalf of the powerbox owner. If the module requests an authority with which the module has been endowed at creation, the authority is simply granted. If the module needs a new authority during operations, the powerbox negotiates with its owner for such authority. If the owner decides to change authorities during operations (either grants or revocations), the powerbox fulfills these changes.

  The Powerbox security pattern proved to be a powerful ingredient in leveraging the security review resources for maximum productivity. It effectively collects the security issues at the boundary between a pair of trust realms into a small body of code. Consequently, the bulk of the code inside a single trust realm does not have to be reviewed for security issues. The vast majority of the CapDesk code went un-reviewed, yet we have reasonable confidence that no vulnerabilities were missed because of this decision.

  This pattern was invented in the course of the DarpaBrowser research. We first developed the pattern for the CapDesk Powerbox, from which the pattern gets its name. The CapDesk Powerbox is the software module that mediates the granting of authorities to a capability confined application from the user. This pattern was reused (though incorrectly, as discussed in the General Truths earlier), at the interface between the DarpaBrowser and the renderer (embodied as the renderPGranter component). By following the now-well-understood pattern henceforth, future developers will be able to build more secure systems at less cost and with greater reliability. The Powerbox pattern is elaborated in the Appendix.

- **HTML embodies the classic Confused Deputy security dilemma.** HTML uses text to designate a page to be accessed without actually conveying the authority to access that page. Both the HREF attribute and the IMG tag are examples of places where the HTML text assumes the browser will use its own authority to fulfill the intent, not of the browser owner, but of the HTML author. This is the classic characterization of the Confused Deputy problem, which occurs when designation is separated from authority [Hardy88]. As a simple example, suppose a page from outside your firewall specifies a URL that is interpreted, inside your firewall, as connoting a particular page available inside the firewall. The page author beyond the wall almost certainly does not have authority to reach this page, but the browser does. In this circumstance, the HTML from that outside page, possibly written by an adversary and in collaboration with the malicious renderer, can use the browser's authority on its own behalf. This did not, as it turned out, violate any of the security goals of the project, but it is a disappointment.

Encountering the Confused Deputy problem was the proximate cause for Norm Hardy, the founder of modern capability thinking, to turn to capabilities in the first place. There is no solution except to totally embrace capabilities, by ubiquitously using capabilities not only in the browser, but in the HTML language itself.

It is possible to build a capability-oriented protocol derived from HTML that has many of the desireable properties of HTML, but which enables proper security enforcement. However, such a protocol would not be backwards-compatible with HTML (though note that one can layer a capability protocol on top of HTML [Close99]). Since this project explicitly called for working with HTML as it exists now, this line of research stopped when the problem had been identified.

- **Event-loop models of concurrency have a synergistic relationship with capabilities for ensuring security.** One of the more common causes of vulnerability is the Time Of Check To Time Of Use (TOCTOU) hole. In a TOCTOU vulnerability, a value is checked to confirm that it is valid, and then before it is used the malicious client changes it. TOCTOU vulnerabilities are exquisitely difficult to detect and fix in systems that use threads as part of their concurrency model, since the value change can happen in between the execution of individual lines of code. The E promise-based architecture, which puts a programmer-friendly face on event loops, guarantees that this path to sneaking in a change cannot occur. All the actions in a single event execute as an atomic operation. This greatly simplifies TOCTOU analysis, detection, and correction.

- **Drag/drop authority must be explicitly granted.** Authority for drag/drop must be explicitly conferred as a launch-time grant, not as a safe non-authority conveying operation. At first glance, it would appear that being a drag source or a drop target is not authority conveying: after all, the user, in performing a drag and a drop, is engaging in the kind of explicit action that can be used in the capability paradigm to identify and convey appropriate authority. However, there is a subtle problem with automatically granting the authority to be a drag source or a drop target to all comers. If components from different trust realms are granted authority inside a single window frame (as in the DarpaBrowser), the differently trusted component can engage in spoofing: it can trick the user into believing it is a part of the main application, not a separate application that must be treated differently. For example, in the DarpaBrowser, if the renderer could designate itself as a drop target, a user drop of a file on the renderer's panel would enable the renderer to present the data in that file without informing the browser. At that point the browser's URL field would be out of sync with the page being displayed.

This would explicitly breach the security goals of the project. It is fortunate for the overall E effort that the DarpaBrowser exercise was the context for making these first authority-distinguishing decisions, otherwise the E platform might well have had to make an upwards-compatibility break to close this vulnerability after it had been more extensively deployed.

- **Explicit Differences between Capability-Oriented Programming and Object-Oriented Programming include:**
  - **No static (global) public mutable state is allowed.** The E programming language enforces this, since there are no static public mutables. This has a minor but real impact on system design: an object such as the java.lang.System.out object cannot be created.
  - **Object instantiation sometimes requires more steps.** In a capability system enforcing POLA, there tend to be more levels of instantiation for an object: the work normally done by a single powerful constructor will, as a part of POLA, often be broken into a series of partial constructors as the final user of the constructed object gets just enough power to perform local customizations of the object. An explicit recognition of this multiple-level instantiation is the Author pattern followed by many emakers (E library packages). An E constructor will often run in the scope of an Authorizer that is first created and granted the needed authorities; then the constructor itself can be handed to less-trusted objects without having to give the less-trusted object the authorities needed to make the constructor. The most complex current example of this pattern is the FrameMakerMakerAuthor in the CapDesk Powerbox. An individual Caplet is granted an individual frameMaker for making windows. To create the frameMaker, there is first an authorization step in which authority to create JFrames (the underlying Swing windows) is granted. Then an intermediate Maker step customizes the frameMaker with an unalterable caplet pet icon and pet name so that the caplet cannot use its power to make frames to spoof the user.

    The Authorization step, and other intermediate levels of instantiation, can be disconcerting for the first-time capability programmer with an object-oriented background. It is, however, a simple extension of standard object-oriented practice. Indeed, between the start of this project and its completion we observed that the use of inner classes in Java has become increasingly ubiquitous in the example code from major vendors such as Sun and IBM. For the Java programmer who has become comfortable with these nested classes, the leap to multiple levels of instantiation is not even a speed bump, but more a pebble on the road. Meanwhile, the benefits of multi-layer POLA-oriented instantiation make it possible to be extremely confident, during debugging, that the majority of the library packages in a system could not have caused a surprising authority-requiring problem: one can see at a glance that the package did not have the authority, and move on to the next candidate for inspection.

- **Facets and Forwarders are common patterns and must be easily supported.** In capability programming, on the boundaries between trust realms, facets and revokable transparent forwarders are often used to grant limited access to powerful objects. The frameMaker above is an example, it is a facet on the Swing JFrame that does not allow the icon or the title prefix to be changed. Fortunately, the E programming language makes the construction of facets and forwarders painless. This implements the following user interface rule that is older than programming, and indeed, older than the printed word: "If you want someone to do something the right way, make the right way the easy way." Forwarders and facets have been made very easy. In E, the code for a general-purpose constructor for revokable forwarders can be written in as little as six lines of code:

```
def makeRevokableForwarderPair(obj) :any {
  var innerObj := obj
  def forwarder {match [verb, args] {E call(innerObj, verb, args)}}
  def revoker {to revoke() {innerObj := null}}
  [revoker, forwarder]
}
```

- **Encapsulation must be strictly enforced.** As noted earlier, modularity discipline must be followed pervasively. It cannot be broken for convenience.

- **In capability style, there can be no unchecked preconditions on the client in a lesser trust realm.** If a client does not fulfill the preconditions in a contract with an interface, and if the implementation of the interface does not check and detect this failure, the results are unpredictable. Such unpredictability is the enemy of security, and cannot be tolerated. In the context of E, a large part of this principle can be implemented through the following E-specific rule:

- **Rigorous guards should be imposed on arguments received across a trust boundary.** Due to the nature of E semantics, a malicious component can send an object across the trust boundary which changes its nature as it is used, essentially spoofing the recipient. E has the most flexible and powerful dynamic type checking system yet devised for a programming language (using *guards*). However, to support rapid prototyping, these guards are optional in E. Therefore a best practice for E objects on the trust boundary is to impose the most restrictive guards possible on every argument received. Because of the DarpaBrowser experience, an experimental feature has been implemented for E that would, on a module-by-module basis, allow the developer to require guards on all variables in the module. A powerbox module, for example, should probably operate with this extra requirement imposed upon it.

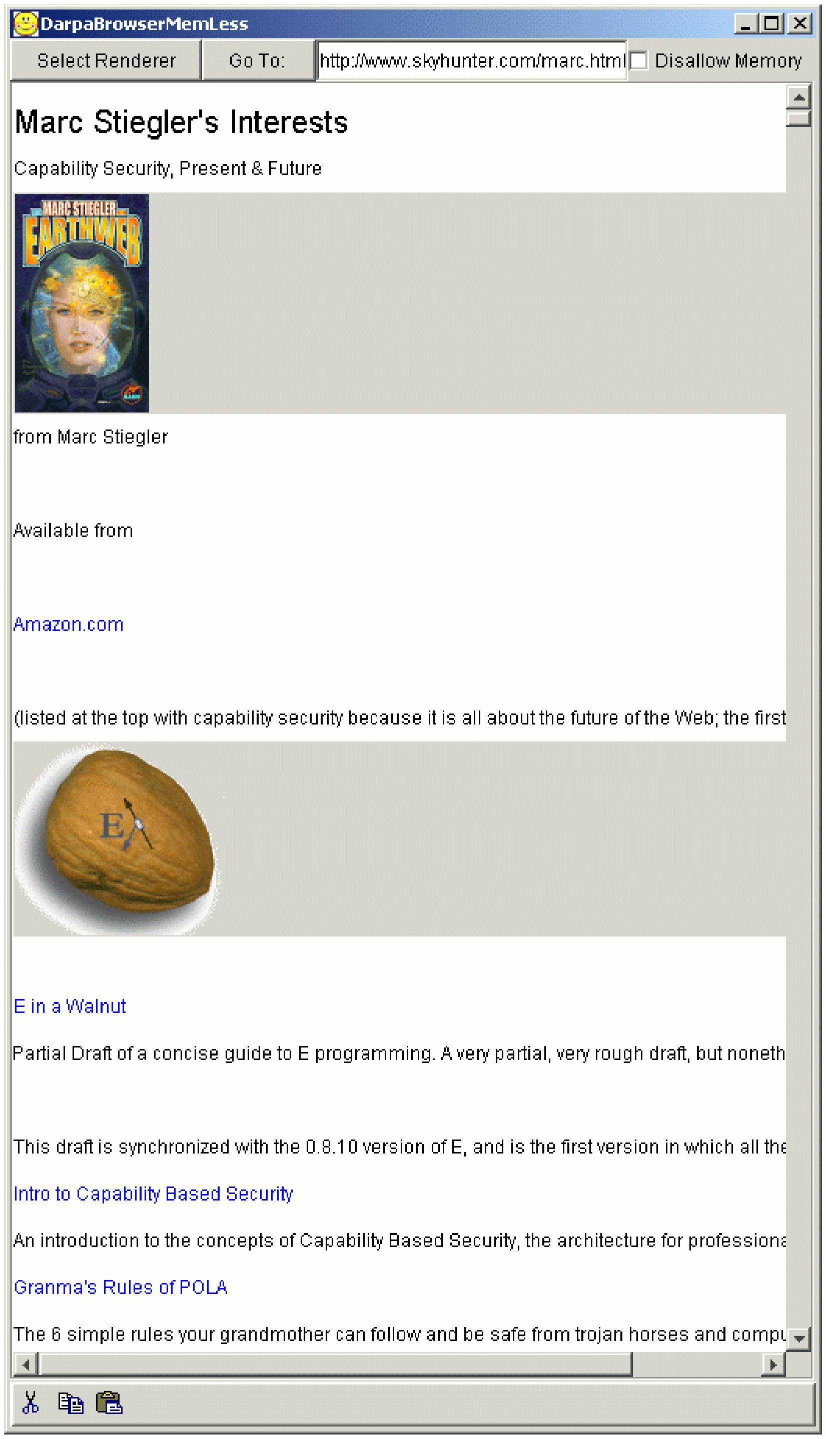
Post-Experiment Developments

Closing Vulnerabilities

The major effort after the security review was to clean up as many of the security vulnerabilities identified in the review as possible for the final deliverable. The single most time consuming part of that effort was to alter the interface between the DarpaBrowser and its renderer to use a parse tree with embedded capabilities for describing the page to the renderer, rather than using text strings and raw URLs.

The DarpaBrowserMemless was the result of that effort. It uses parse trees rather than string matching, and embeds the correct capabilities in the parse tree handed to the renderer rather than playing a "guessing game" by trying to validate the url strings sent back to it from the renderer.

In addition to upgrading the benign and evil renderers to work with the new browser, a new renderer was built, the "capTreeMemless" renderer. This renderer addressed a desire expressed by the security review team, which was that at least one render that did not have special TCB authority be created that demonstrated that authority conveyance for items other than links, in particular images, operated correctly inside the capability paradigm.



Marc Stiegler's Interests

Capability Security, Present & Future

from Marc Stiegler

Available from

Amazon.com

(listed at the top with capability security because it is all about the future of the Web; the first

E in a Walnut

Partial Draft of a concise guide to E programming. A very partial, very rough draft, but noneth

This draft is synchronized with the 0.8.10 version of E, and is the first version in which all the

Intro to Capability Based Security

An introduction to the concepts of Capability Based Security, the architecture for professiona

Granma's Rules of POLA

The 6 simple rules your grandmother can follow and be safe from trojan horses and compu

CapTreeMemLess renderer at work. While not visually attractive, it does demonstrate the use of an HTML parse tree with embedded capabilities for rendering images and enabling links.

The status of all the vulnerabilities identified in the security review can be tracked through links embedded in the HTML version of the review found at **http://www.combex.com/papers/darpa-review/index.html**. These links tie directly into the bug tracking system for the E platform. As can be seen there, nineteen of the twenty-one vulnerabilities have been closed. The two vulnerabilities left unaddressed, and the reasons for leaving them unrepaired, are:

- **125505 Suppress show while allowing frame display:** An application with minimal window-creation authority can, by creating a window, steal the focus from the currently active application, and possibly receive sensitive data being typed by the user before the user realizes that he is no longer working with the intended window. This vulnerability does not impact the DarpaBrowser's ability to confine the renderer (the render does not have window creation authority), but it is a true vulnerability that should be repaired. However, due to the way in which Swing bundles the window opening and window activation operations, it is not trivial to fix: the JFrame uses a single operation, show() to open the window, bring it to the front, and steal the focus. Simply suppressing the show() method is not one of the choices, since it is a required operation. We would still proceed to fix this problem (by building an experimental subclass of JFrame and re-implementing as much as it takes to unbundle the opening of the window from the stealing of the focus) if it were not for an additional development that occurred late in the course of the project: IBM brought out an alternative windowing kit, SWT, that can replace Swing and appears to be superior in many ways to Swing. SWT is described later in this report. Since we now tentatively plan to replace Swing with SWT for E programming, a major undertaking to build a better JFrame would be a waste of effort.

- **125503 Prevent backtrace revealing private data:** A thrown exception could in principle carry sensitive information across a trust boundary. Once again, this does not effect the DarpaBrowser and its goals: the renderer is never in a position to receive sensitive data that it does not have more straightforward access to anyway (i.e., the browser may be used to read a sensitive page, in which case the renderer gets it directly anyway, for rendering). Furthermore, the description of this vulnerability as written in the security review has been found to be erroneous: the bug is both much less dangerous (it cannot leak authority) and much more difficult to fix than had been understood at the time of the review. As a consequence, we have allocated resources to more urgent requirements at this time.

Development of Granma's Rules of POLA

As a consequence of the demonstrability of the CapDesk/DarpaBrowser system, we have been able to present capability security principles to large audiences of people who previously would have found the topic too academic to appreciate. As people grasped that security really was possible, a few retreated into complaints that, except in very simple examples (like the current CapDesk/DarpaBrowser example), the management of the security features of the system would be too complicated for "normal" people to use. Fortunately, developing the CapDesk system gave us sufficient insight into the "normal security needs" for the "normal user", that we were able to develop a simple set of guidelines with the following features:

- Can be quickly taught to people of only modest computer skill,
- Allows people to get their work done easily and without barriers,
- Nevertheless guarantees that effective security (for "normal" needs) is maintained.

Having been given a serious review by the E language community in the E language discussion group, these guidelines are now known as Granma's Rules of POLA. Their description can be found in the Appendix. Much research remains to be done in the area of making capability security user friendly, but this document points in a promising direction.

Introduction of SWT

In the closing months of this contract, a new technology came to our attention. IBM has released an alternative to JavaSoft's AWT/Swing user interface toolkit, the Standard Widget Kit (SWT). This widget kit has already been used for a very sophisticated user-interface application, the Eclipse software development environment. SWT has the following advantages over AWT/Swing:

- SWT is much smaller than AWT/Swing. This has tremendous ramifications for the taming process, which was highlighted in the security review as the single greatest risk in the E platform. Reducing the size of the toolkit has a more-than-linear and critical impact on both feasibility and risk.
- SWT engages the native widget kit at a higher level of abstraction than does Swing. As a consequence, applications written in Java/SWT really do look and feel exactly like any other application written specifically for the platform, since it is usually using the native widgets. The difference in attractiveness and comfort for the user is, all by itself, a compelling reason to switch. Java with SWT is an enormous step forward in the land of "Write once, run anywhere".
- SWT uses a true open source license, unlike the Swing SCSL license. This has two important advantages: if it turns out during taming that simple taming is complicated and problem-laden (as with the JFrame grab-focus-on-opening behavior in Swing, described earlier), we have the option of simply modifying the problematic class rather than attempting a poor taming effort. The second advantage is that the license allows us to compile SWT into native and dot-net forms, allowing us to use a single standard toolkit across all three of the E implementations requested by the E user base: Java, native binary, and dot-net. This too would be, all by itself, a compelling reason to switch.

- It uses a very different garbage collection strategy than Swing. We uncovered a significant garbage collection hole in Java for Windows and Linux (Java on the Mac seems fine) during the course of this effort. While everything worked well enough for the prototyping work done in this project, for a production environment this memory leak would be unacceptable. We believe the memory leak lies in Swing and its interface to the native graphics system. The SWT garbage collection strategy, while more primitive, is also more likely to work correctly. Once again, this all by itself would be a compelling reason to switch.

  Our exploration of SWT has not yet progressed to the point where we are certain that it meets our needs for functionality and capability-compatible modularity, but it looks very promising. The Eclipse development environment is *de facto* evidence that the toolkit has extensive functionality, so it is unlikely that a problem will be identified in this realm. The capability-compatibility remains the greatest risk, though preliminary investigation has not identified any fatal problems (though we have identified one place, in the drag and drop formats supported, where more work will be required than is required in Swing).

Assessment of Capabilities for Intelligent Agents and User Interface Agents

One of the enhancements made to the DarpaBrowser that went above and beyond the minimum requirements of the contract was the embedding of a caplet launching framework inside the browser itself. If the DarpaBrowser encounters a page whose url ends with ".caplet", the browser not only renders the text from that page, but also launches a capability confined application from it. This application lives in a separate trust realm from the browser itself; it is guaranteed that the new caplet cannot use the browser's authorities, and it is also guaranteed that the browser cannot use any authorities later granted to the new caplet.

The use of the DarpaBrowser to launch caplets from across the web in safety is a simple demonstration of the potential of capability security for enabling the development of a new generation of harmless yet powerful mobile software agents. Cooperating members of a community could grant sensible authorities (such as, the authority to read a document folder) to mobile agents confidently and painlessly. These software agents could delegate their authority to more evolved agents without either user intervention or user concern. There is no security or usability reason such agents could not evolve to the point where they, not the human beings, were doing the most work on the Web, informing us about results when they reached a threshold of value relevant to us.

Let us look at a simple example, the SETI screensaver project, in a capability context. The SETI screensaver is effectively an agent seeking computing resources. The authorities it needs are reasonably simple:

Authority to communicate with a single Web site (the SETI central coordination site)

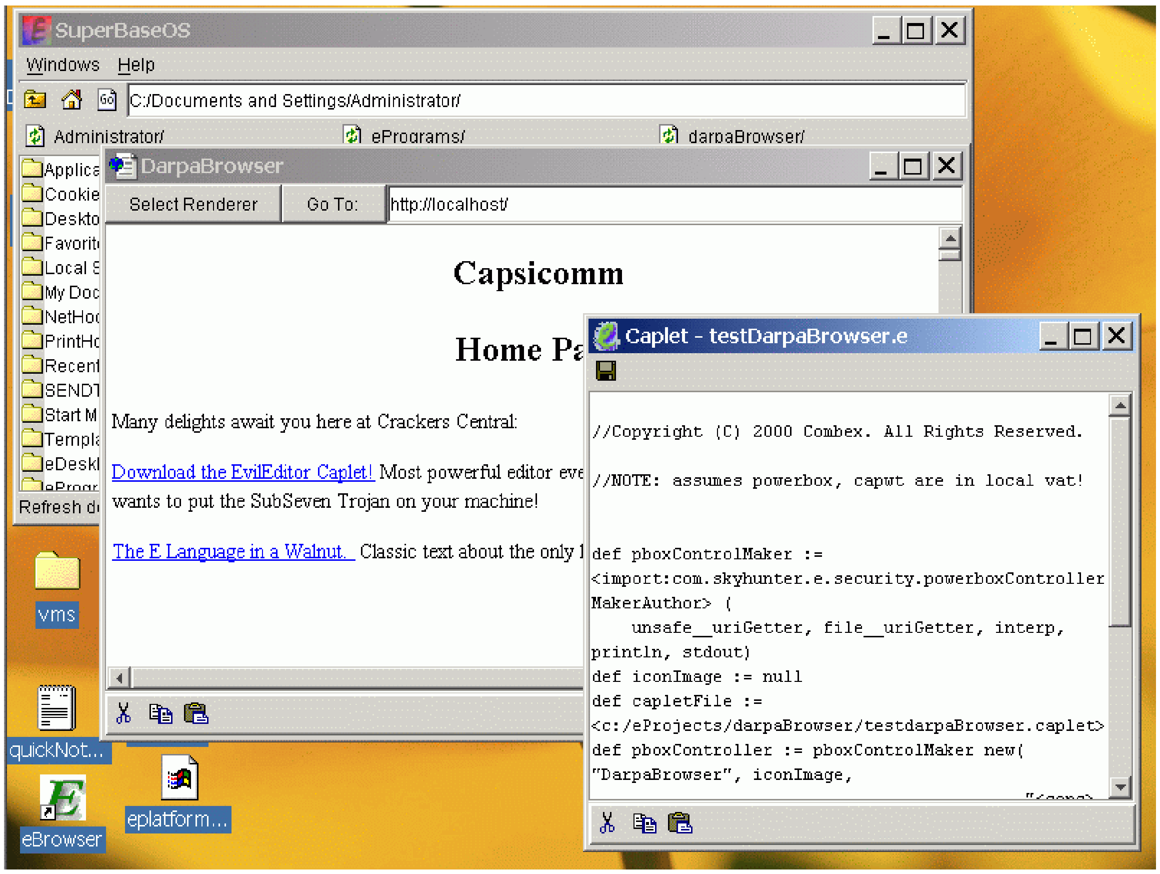
Authority to write to the screen until a keystroke or mouse click occurs (i.e., an authority on a revokable forwarder to the display, with the forwarder automatically revoking itself on KeyPress or MouseDown)

These authorities run little risk of compromising the computer. Indeed, these authorities fall easily inside Granma's Rules of POLA: running the SETI screensaver requires no breaking of the basic security principles. Yet the current SETI software, because it must run with standard Windows ambient authority, is as dangerous as the possibly-malicious renderer in the DarpaBrowser. Large numbers of people who would otherwise run SETI will not do so because of the security implications. And the caution about allowing software agents to run on the individual home computer is necessarily multiplied by orders of magnitude if the desire is to run agents on large databases. As a consequence, we believe that capability security is not merely a good idea for software agents. We believe capability security is a requirement if this kind of computing is ever to achieve its destiny.

To support software agents properly, the capability management system embodied in the CapDesk Powerbox must be fleshed out to support all the different kinds of authority grants that make sense. A particular area where some research is required is in the general-purpose designation of authority to speak to other objects, as distinct from the authority to access system resources (the authority to read a directory is access to a system resource; the authority to talk to a third party spell checker requires a general-purpose object-to-object granting framework). But the principles have already been demonstrated. The road to a flexible capability framework is generally smooth, with only a few twists and turns remaining before software agents can be properly and fully supported.

Conclusions

Capability based security enables software developers to achieve computer security goals that cannot be reached with conventional security systems. The capability paradigm also enables more cost-effective security reviews that can provide better confidence that these security goals have been achieved. Furthermore, early indications are that these security goals can be achieved with neither undue hindrance of the user, nor with noticeable constriction of the functionality of the computing platform. The user-friendly power of capability security is demonstrated in the picture below.



Capsicomm

Home Pa

Many delights await you here at Crackers Central:

Download the EvilEditor Caplet! Most powerful editor eve
wants to put the SubSeven Trojan on your machine!

The E Language in a Walnut. Classic text about the only l

```
//Copyright (C) 2000 Combex. All Rights Reserved.

//NOTE: assumes powerbox, capwt are in local vat!

def pboxControlMaker :=
<import:com.skyhunter.e.security.powerboxController
MakerAuthor> (
    unsafe__uriGetter, file__uriGetter, interp,
println, stdout)
def iconImage := null
def capletFile :=
<c:/eProjects/darpaBrowser/testdarpaBrowser.caplet>
def pboxController := pboxControlMaker new(
"DarpaBrowser", iconImage,
```

In this picture four different trust realms interact flexibly, securely, and in a user-friendly fashion. The CapDesk in the background is running with TCB authority; the DarpaBrowser is in a confined trust realm with only an HTTP protocol authority; the renderer inside the browser is running in a trust realm with only access to a single window panel, and a single URL at any given time as specified by the browser; the text-editing Caplet is running in a trust realm with read/write authority to a single file. The applications all look and feel like ordinary unconfined applications; no passwords are needed, no certificates need to be studied for validity; interaction between the trust realms proceeds smoothly and intuitively, but only at the behest of the user, never under the control of the less-trusted applications.

However, even with the power of capability security, truly securing our computers from cyberattack is hard work. In particular, the core infrastructure--components such as CapDesk, the Powerbox and the E Language itself--must be developed by seasoned capability security professionals, and must be reviewed by peers of equal skill. We suggest that, for this reason among others, such infrastructure needs to be built under public scrutiny, using open source licenses, and that professional security reviews are still a crucial part of the process of building secure systems.

Nonetheless, one of the truly remarkable powers capability security gives us is the ability to turn the bulk of the work in building secure systems over to people who have no security expertise whatsoever. Indeed, untrusted developers (even professional crackers!) can build large parts of the most sensitive computing systems. How can this be? It can be because any module that does not receive powerful authorities can be written by anyone in the world with no security consequences. This is explicitly demonstrated by the Malicious Render's inability to achieve authority-stealing security penetrations.

Capability patterns of software modularization as simple as the basic E module mechanism (which grants no authority whatsoever upon importation) and as sophisticated as the Powerbox developed to confine the DarpaBrowser can isolate untrusted subsystems, be they modules written by subcontractors of the British government or agents of the Chinese intelligence services. While we do not expect to see our military depending on the Chinese government for sensitive software development any time soon, this scenario demonstrates the power of the capability paradigm, and the brightness of the future in which capabilities become ubiquitous.

References

[Bishop79] Matt Bishop, "**The Transfer of Information and Authority in a Protection System**", in *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, published as *Operating System Review*, vol. 13, #4, 1979, pp 45-54.

[Boebert84] W. E. Boebert, "**On the Inability of an Unmodified Capability System to Enforce the *-Property**", in *Proceedings of the 7th DoD/NBS Computer Security Conference*, 1984.

[Chander01] Ajay Chander, Drew Dean, John Mitchell, "**A State Transition Model of Trust Management and Access Control**", *14th IEEE Computer Security Foundations Workshop*, Online at **http://citeseer.nj.nec.com/rd/95292128%2C502365%2C1%2C0.25%2CDownload/http%3AqSqqSqciteseer.nj.nec.comqSqcacheqSqpapersqSqcsqSq25723qSqhttp%3AzSzzSzcrypto.stanford.eduzSzdczSzpaperszSzacl-cap-tm.pdf/chander01statetransition.pdf**.

[Close99] Tyler Close, "**Announcing Droplets**", 1999. email archived at **http://www.eros-os.org/pipermail/e-lang/1999-September/002771.html**.

[Dennis66] Jack Dennis, E. C. van Horn, "**Programming Semantics for Multiprogrammed Computations**", in *Communications of the ACM*, vol. 9, pp. 143-154, 1966.

[Donnelley81] Jed E. Donnelley, "**Managing Domains in a Network Operating System**" (1981) *Proceedings of the Conference on Local Networks and Distributed Office Systems*, pp. 345-361. Online at **http://www.nersc.gov/~jed/papers/Managing-Domains/**.

[Ellison99] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, B. Thomas, and T. Ylonen, "**SPKI Certificate Theory**" IETF RFC 2693. Online at **http://www.ietf.org/rfc/rfc2693.txt**.

[Gong89] Li Gong, "A Secure Identity-Based Capability System", IEEE Symposium on Security and Privacy, 1989. Online at **http://citeseer.nj.nec.com/rd/95292128%2C3427%2C1%2C0.25%2CDownload/http%3AqSqqSqciteseer.nj.nec.comqSqcacheqSqpapersqSqcsqSq1728qSqhttp%3AzSzzSzweb3.javasoft.com%3A81zSzpeoplezSzgongzSzpaperszSzcap.pdf/gong89secure.pdf**

[Granovetter73] Mark Granovetter, "**The Strength of Weak Ties**", in: *American Journal of Sociology* (1973) Vol. 78, pp.1360-1380.

[Hardy85] Norm Hardy, "**The KeyKOS Architecture**", *Operating Systems Review*, September 1985, pp. 8-25. Updated at **http://www.cis.upenn.edu/~KeyKOS/OSRpaper.html**.

[Hardy86] Norm Hardy, " **U.S. Patent 4,584,639: Computer Security System**", Key Logic, 1986 *(The "Factory" patent)* , Online at **http://www.cap-lore.com/CapTheory/KK/Patent.html**.

[Hardy88] Norm Hardy, " **The Confused Deputy, or why capabilities might have been invented**", *Operating Systems Review* , pp. 36:38, Oct., 1988, **http://cap-lore.com/CapTheory/ ConfusedDeputy.html**.

[Harrison76] Michael Harrison, Walter Ruzzo, Jeffrey Ullman., " **Protection in Operating Systems**", Comm. of ACM, Vol. 19, n 8, August 1976, pp.461-471. Online at **http://www.cs.fiu.edu/~nemo/cot6930/hru.pdf**.

[Hewitt73] Carl Hewitt, Peter Bishop, Richard Stieger, " **A Universal Modular Actor Formalism for Artificial Intelligence**", *Proceedings of the 1973 International Joint Conference on Artificial Intelligence* , pp. 235-246.

[Jones76] A. K. Jones, R.J. Lipton, Larry Snyder, " **A Linear Time Algorithm for Deciding Security**", in *Proceedings of the 17th Symposium on Foundations of Computer Science* , Houston, TX, 1976, pp 33-41.

[Kahn87] Kenneth M. Kahn, Eric Dean Tribble, Mark S. Miller, Daniel G. Bobrow: " **Vulcan: Logical Concurrent Objects**", in *Research Directions in Object-Oriented Programming* , MIT Press, 1987: 75-112. Reprinted in *Concurrent Prolog: Collected Papers* , MIT Press, 1988.

[Kahn96] Kenneth M. Kahn, " **ToonTalk - An Animated Programming Environment for Children**", *Journal of Visual Languages and Computing* in June 1996. Online at **ftp://ftp-csli.stanford.edu/pub/Preprints/tt_jvlc.ps.gz**. An earlier version of this paper appeared in the *Proceedings of the National Educational Computing Conference (NECC'95)* .

[Kain87] Richard Y. Kain, Carl Landwehr, "On Access Checking in Capability-Based Systems", in *IEEE Transactions on Software Engineering* SE-13, 2 (Feb. 1987), 202-207. Reprinted from the *Proceedings of the 1986 IEEE Symposium on Security and Privacy* , April, 1986, Oakland, CA; Online at **http://chacs.nrl.navy.mil/publications/CHACS/Before1990/1987landwehr-tse.pdf**.

[Karp01] Alan Karp, Rajiv Gupta, Guillermo Rozas, Arindam Banerji, " **Split Capabilities for Access Control**", HP Labs Technical Report HPL-2001-164, Online at **http://www.hpl.hp.com/techreports/2001/HPL-2001-164.html**.

[Lampson71] Butler Lampson, "**Protection**", in *Proceedings of the Fifth Annual Princeton Conference on Informations Sciences and Systems*, pages 437-443, Princeton University, 1971. Reprinted in Operating Systems Review, 8(l), January 1974. Online at **http://citeseer.nj.nec.com/rd/95292128%2C287804%2C1%2C0.25%2CDownload/http%3AqSqqSqciteseer.nj.nec.comqSqcacheqSqpapersqSqcsqSq13282qSqhttp%3AzSzzSzwww.cs.purdue.eduzSzhomeszSzjvzSzsmczSzpubszSzLampson-OSR74.pdf/protection.pdf**.

[Levy84] Henry Levy, "**Capability-Based Computer Systems**", Digital Press, 1984. Online at **http://www.cs.washington.edu/homes/levy/capabook/**.

[Miller00] Mark S. Miller, Chip Morningstar, Bill Frantz, "**Capability-based Financial Instruments**", in Proceedings of Financial Cryptography 2000, Springer Verlag, 2000. Online at **http://www.erights.org/elib/capability/ode/index.html**.

[Morningstar96] Chip Morningstar, "**The E Programmer's Manual**", Online at **http://www.erights.org/history/original-e/programmers/index.html**. *(Note: The "E" in the title and in this paper refers to the language now called "Original-E".)*

[Raymond99] Eric Raymond, "**The Cathedral and the Bazaar**", O'Reilly, 1999, Online at **http://www.tuxedo.org/~esr/writings/cathedral-bazaar/**.

[Rees96] Jonathan Rees, "**A Security Kernel Based on the Lambda-Calculus**", (MIT, Cambridge, MA, 1996) MIT AI Memo No. 1564. Online at **http://mumble.net/jar/pubs/secureos/**.

[Saltzer75] Jerome H. Saltzer, Michael D. Schroeder, "**The Protection of Information in Computer Systems**", *Proceedings of the IEEE*. Vol. 63, No. 9 (September 1975), pp. 1278- 1308. Online at **http://cap-lore.com/CapTheory/ProtInf/**.

[Sansom86] Robert D. Sansom, D. P. Julian, Richard Rashid, "**Extending a Capability Based System Into a Network** Environment" (1986) Research sponsored by DOD, pp. 265-274.

[Shapiro83] Ehud Y. Shapiro, "**A Subset of Concurrent Prolog and its Interpreter**". Technical Report TR-003, *Institute for New Generation Computer Technology*, Tokyo, 1983.

[Shapiro99] Jonathan S. Shapiro, "**EROS: A Capability System**", Ph.D. thesis, University of Pennsylvania, 1999. Online at **http://www.cis.upenn.edu/~shap/EROS/thesis.ps**.

[Shapiro00] Jonathan Shapiro, "**Comparing ACLs and Capabilities**", 2000, Online at **http://www.eros-os.com/essays/ACLSvCaps.html**.

[Shapiro01] Jonathan Shapiro, "**Re: Old Security Myths Continue to Mislead**", email archived at **http://www.eros-os.org/pipermail/e-lang/2001-August/005532.html**.

[Sitaker00] Kragen Sitaker, "**thoughts on capability security on the Web**", email archived at **http://lists.canonical.org/pipermail/kragen-tol/2000-August/000619.html**.

[Snyder77] Larry Snyder, "**On the Synthesis and Analysis of Protection Systems**", in *Proceedings of the 6th ACM Symposium on Operating System Principles*, published as *Operating Systems Review* vol 11, #5, 1977, pp 141-150.

[Stiegler00] Marc Stiegler, "**E in a Walnut**", Online at **http://www.skyhunter.com/marcs/ewalnut.html**.

[Tanenbaum86] Andrew S. Tanenbaum, Sape J. Mullender, Robbert van Renesse, "**Using Sparse Capabilities in a Distributed Operating System**" (1986) Proc. *Sixth Int'l Conf. On Distributed Computing Systems*, IEEE, pp. 558-563. Online at **ftp://ftp.cs.vu.nl/pub/papers/amoeba/dcs86.ps.Z**.

[Tribble95] Eric Dean Tribble, Mark S. Miller, Norm Hardy, Dave Krieger, "**Joule: Distributed Application Foundations**", Online at **http://www.agorics.com/joule.html**, 1995.

[Wagner02] David Wagner & Dean Tribble, " **A Security Analysis of the Combex DarpaBrowser Architecure**", Online at **http://www.combex.com/papers/darpa-review/**.

[Wallach97] Dan Wallach, Dirk Balfanz, Drew Dean, Edward Felten, "**Extensible Security Architectures for Java**", in *Proceedings of the 16th Symposium on Operating Systems Principles* (Saint-Malo, France), October 1997. Online at **http://www.cs.princeton.edu/sip/pub/sosp97.html**.

[Yee02a] Ka-Ping Yee, " **User Interaction Design for Secure Systems**", Berkeley University Tech Report CSD-02-1184, 2002. Online at **http://www.sims.berkeley.edu/~ping/sid/uidss-may-28.pdf**.

[Yee02b] Ka-Ping Yee, Mark Miller, " **Auditors: An Extensible, Dynamic Code Verification Mechanism**", Online at **http://www.sims.berkeley.edu/~ping/auditors/auditors.pdf**.

Appendices

Table of contents

Appendix 1: Project Plan

Project Plan
Capability Based Client
Combex Inc.

BAA-00-06-SNK; Focused Research Topic 5

Technical Point of Contact:
Marc Stiegler
marcs@skyhunter.com

INTRODUCTION AND OVERVIEW

Capability security is Combex will develop a capability-secure Web browser that confines its rendering engine so that, even if the rendering engine is malicious, the harm which the renderer can do is severely limited.

HYPOTHESES

We hypothesize that capability secure technology can simply and elegantly implement security regimes, based on the Principle of Least Authority, that cannot be achieved with orthodox security architectures including firewalls and Unix Access Control Lists. Specifically, we hypothesize that capability security can confine the authority granted to an individual module of a Web Browser, the rendering engine, such that the rendering engine has no authority over any component of the computer upon which it resides except for:

authority over a single window panel inside the web browser, where it has full authority to draw as it sees fit

authority to request URLs from the web browser in a manner such that the web browser can confirm the validity of the request

authority to consume compute cycles for its processing operations.

We cannot directly test and prove that all theoretically possible authorities beyond these three are absent. Therefore, for experimental purposes, we invert our experimental basis to specifically prove that several other traditionally easy-to-access authorities are unavailable: We hypothesize that the rendering engine will *not* have any of the following specific authorities:

No authority to read or write a file on the computer's disk drives

No authority to alter the field in the web browser that designates the URL most recently retrieved

No authority to alter the web browser's icon image in the top left corner of the window

No authority to alter the title bar in the web browser's window

No authority to receive information from an URL that is not on the most recently requested web page (the HTML text URL, the image URLs for that page, and other URLs that specify page content for the main HTML text URL; it may also request a change of URL to another page specified by a hyperlink on the page).

No authority to move to another URL (via hyperlink) without having the web browser update the browser field that designates the current page being displayed

No authority to send information to any URL on the Web

Even with the limited set of authorities granted to the renderer, there are a couple of malicious acts it can perform, though these acts are severely constrained. In particular, we hypothesize that the renderer can:

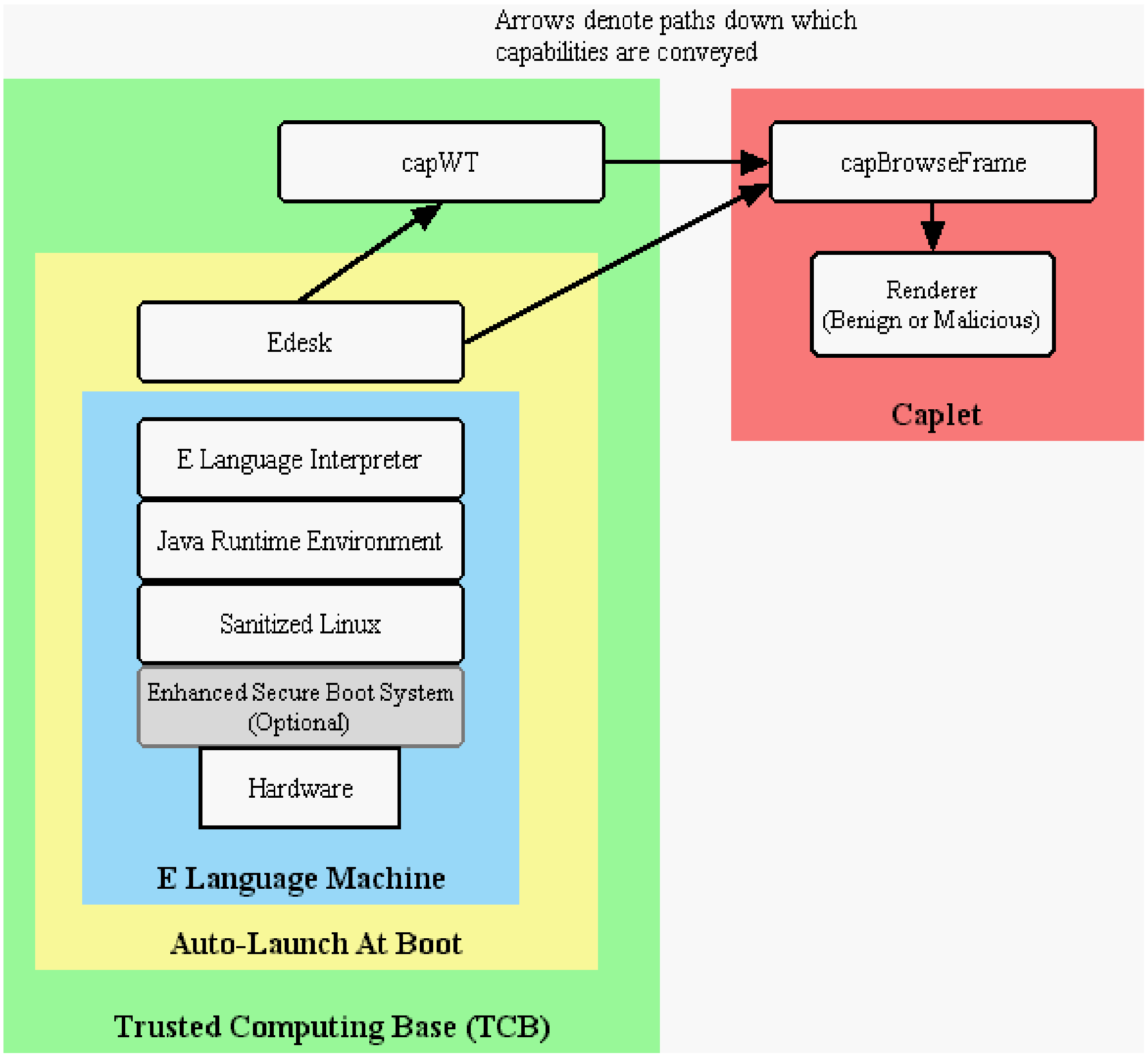
Undertake a Denial of Service attack by consuming as many compute cycles as it can acquire; the counterstrategy for the computer owner will be to shut down the application.

Render the current web page incorrectly. Incorrect rendering may even take the form of rendering what appears to be a different page, though this false page must be based on data embedded in the renderer's source code, and cannot be a true live representation of another actual page off the Web.

EXPERIMENTAL SETUP

To conduct the experiment, we shall first build a web browser that supports modularly pluggable alternate rendering engines on top of an E Language Machine(ELM).

The E programming language supplies a capability secure, strongly encrypted software development infrastructure, along with a deadlock free promise-based concurrency architecture. Using E and the Capability Windowing Toolkit API which comes with E, one can built software applications, known as *caplets*, that have individually confined authority to separate window panels, and unforgeable, unspoofable window frames. The E Language Machine will be built by putting an E virtual machine on top of a sanitized Linux kernel as the only application running on the platform; in effect, this turns the entire computer into a pure capability secure system (See Diagram below).


Arrows denote paths down which capabilities are conveyed
capWT
capBrowseFrame
Renderer (Benign or Malicious)
Edesk
Caplet
E Language Interpreter
Java Runtime Environment
Sanitized Linux
Enhanced Secure Boot System (Optional)
Hardware
E Language Machine
Auto-Launch At Boot
Trusted Computing Base (TCB)

Having built a modular web browser on ELM, we shall prove the basic operational success of the browser by building a simple Benign Renderer that merely performs its rendering function, presenting Web pages to the best of its ability within the context of its understanding of HTML syntax and semantics. Since this Benign Renderer will be plugged in using the same modular interface, it will be living in the same capability confinement as the confinement in which the Malicious Renderer will later operate.

Once this has been demonstrated successfully, we will build a Malicious Renderer that will attempt to exercise the authorities explicitly disallowed in the Hypotheses. This Renderer will also exercise the two types of malicious behavior that are allowed by the granted authorities.

PROOF OF HYPOTHESES

Having built a basic Malicious Renderer, we shall invite two prominent members of the security community to consult with us by exploring the capability confinement around the Renderer, and modifying the Renderer as they see fit to exploit any opportunities for malicious activity we have missed. We anticipate that security breaches identified by the consultants may fall into one of these categories:

Simple implementation flaws: An easily corrected implementation flaw caused by an error in the implementation of the capability architecture. Should our security consultants identify such flaws, we will fix them prior to final delivery and demonstration, though we will report them in the final report. Such flaws are not considered to be proofs of invalidity of the hypothesis.

Complex implementation flaws: An implementation flaw that is not easily corrected in the experimental system. Such a flaw will be not be corrected for final delivery, will be reported, but will not be considered a proof of the invalidity of the hypothesis.

Architectural flaws: An architectural flaw is a flaw that cannot be corrected within the domain of a pure capability system. Such a flaw would be considered proof that the hypotheses were invalid.

Because the distinction between a complex implementation flaw and an architectural flaw could be blurry, if a flaw is identified that falls into either of these two categories, the consultants themselves will write sections of the final report detailing their assessment of the correct categorization and the reasons for that categorization.

DEMONSTRATIONS

Demonstration 1: Our first demonstration will present a web browser using a Benign Renderer, to show that we have achieved the construction of a web browser with pluggable rendering engines. We expect to make this demonstration on or about November 4, 2001.

Demonstration 2: Our second demonstration will present the web browser using the Malicious Renderer. This Malicious renderer will attempt to exercise the disallowed authorities specified under Hypotheses, and will demonstrate the two allowed types of malicious behavior described in the Hypotheses. We expect to make this demonstration on or about March 1, 2002.

Demonstration 3: Our third demonstration will present the web browser using the enhanced Malicious Renderer, i.e., the Renderer as altered by our security consultants to exploit security weaknesses they have identified. This demonstration will highlight flaws in the categories of "complex implementation flaws" and "architectural flaws". We expect to make this demonstration on or about June 28, 2002.

Appendix 2: DarpaBrowser Security Review

A Security Analysis of the Combex DarpaBrowser Architecture

David Wagner
Dean Tribble
March 4, 2002

INTRODUCTION

We describe the results of a limited-time evaluation of the security of the Combex DarpaBrowser, built on top of Combex's E architecture. The goal of our review was to evaluate the security properties of the DarpaBrowser, and in particular, its ability to confine a malicious renderer and to enforce the security policy described in the Combex Project Plan. Our mission was to assess the architecture. We were also asked to analyze the implementation, but only for purposes of identifying whether there were implementation bugs that could not be fixed within the architecture.

This report contains the results of in excess of eighty person-hours of analysis work. Tribble and Wagner spent a week intensively reviewing E version 0.8.12c and the DarpaBrowser implementation contained therein. Stiegler and Miller were on hand to answer questions.

1. DARPABROWSER PROJECT

This section restates the security goals to be accomplished and expands on the detailed threats to be considered in the review process.

1.1 General goals

As described in the original Focused Research Topic (FRT) for which this capability based client was developed,

"The design objective for the client is to render pages in such a manner that pages are effectively confined and prevented from corrupting each other or the underlying operating system. The capability-based protection is to be afforded by the confinement mechanism even in the presence of vulnerabilities in the rendering engine, presence of malicious code, or malicious data as input. Moreover, under all circumstances the Universal Resource Locator (URL) must either be accurately displayed or an appropriate fault condition displayed as to why the URL cannot safely or accurately be displayed."

As delineated in more detail in the Combex Project Plan, the renderer for the capability based client shall be confined to the extent of not having any of the following abilities:

1. No ability to read or write a file on the computer's disk drives
2. No ability to alter the field in the web browser that designates the URL most recently retrieved
3. No ability to alter the web browser's icon image in the top left corner of the window
4. No ability to alter the title bar in the web browser's window
5. No ability to receive information from an URL that is not on the most recently requested web page (the HTML text URL, the image URLs for that page, and other URLs that specify page content for the main HTML text URL; it may also request a change of URL to another page specified by a hyperlink on the page). See note below.
6. No ability to move to another URL (via hyperlink) without having the web browser update the browser field that designates the current page being displayed
7. No ability to send information to any URL on the Web. See note below.

For item 5, as mentioned in the Project Plan, it is important to draw a distinction between a renderer that is rendering badly, as opposed to a renderer that is rendering based on information from unauthorized sources. A renderer could simply display “Page not available” regardless of what input it receives; this would be an example of bad rendering, rather than a breach of security. In a subtler example, if the renderer draws only a single image that has been specified in the authorized Web page, it could in principle be viewed as a rendering of an URL other than the designated one; nonetheless, we consider it to be a bad rendering, since it is displaying a piece of the specified page.

For item 7, we interpret the phrase “any URL” to mean “any arbitrary URL, or any URL not specified in the HTML of the current page to receive information”; clearly if the HTML of the current page specifies a form to be filled out, it is valid to send the form data to the specified location.

We consider these objectives in the presence of two threat models:

1.1.1 Threat 1: The Lone Evil Renderer

In this threat model, the renderer is acting alone to breach its confinement. It will attempt to compromise the integrity of the user’s system, collect private data, use the user's authority to reach unrelated web pages, and attempt to sniff passing LAN traffic, without outside assistance.

1.1.2 Threat 2: Conspiring Server

In this scenario, the malicious renderer is working with a remote web site to breach confinement. At first it might seem that such a match-up of a malicious renderer with a malicious server is unlikely: why would a user happen to wander over to the conspiring Web site? In fact, this scenario is quite reasonable: if the renderer starts drawing poorly, what could be more natural than to go to the developers' Web site to see if there is an upgrade or patch available? In any case, this is the extreme version of the simpler scenario in which a benign but flawed renderer is attacked by a malicious web page: in principle, a sufficiently vulnerable benign renderer could be totally subverted to do the web site operator's bidding, becoming a malicious renderer with a conspiring server.

In the context of this threat model, it is important to discriminate the meaning security can have within the scope of the basic nature of HTML. First of all, there is necessarily an explicit overt channel available to the web site, using the form tag as defined in HTML. Using this channel does not violate any of the criteria set forth in the FRT or the Project Plan, but it does impose an important constraint on the quality of security when faced with a conspiring server.

An even more interesting related issue was identified early in the review: HTML itself assumes the ubiquitous usage of *designation without authority*, a fundamental violation of capability precepts. As a consequence, any correctly designed renderer suffers from the *confused deputy* problem, first elaborated by Norm Hardy, described at http://www.cap-lore.com/CapTheory/ConfusedDeputy.html.

A worst-case example of this problem can be found in the following situation. Suppose the malicious web site is operated by an adversary who knows the URL of a confidential page on the user's LAN, behind the firewall. When the user comes to the web site (perhaps in search of an upgrade version of the renderer), the site sends a framed page using the HTML frame tag. The frame designates two pages: one page is a form to be submitted back to the malicious web site, and one page is the confidential page whose URL is known to the adversary. Given this framed set of pages, the malicious renderer has all the authority it needs to load the confidential data (in framed page 2) and send it to the adversary (as the query string submitted with the form of framed page 1).

Even this does not violate the goals stated in the FRT or the Project Plan, as outward communication to the operator of the current page is not required to be confined. But it does highlight a need to be clear about what can and cannot be achieved without redefining HTML and other protocols whose strategy of unbundling designation and authority leave users vulnerable to confused deputy attacks.

1.1.3 Other Threat Models

Several other threat models were rejected as part of the analysis since they were not included, explicitly or even implicitly, in either the FRT or the Project Plan. Conspiracies of confined malicious renderers, using wall-banging or other covert channels to communicate, are considered out of scope. Conspirators playing a man-in-the-middle role on the network (at the user's ISP, for example) are out of scope. And denial of service is explicitly stated to be out of scope in the Project Plan.

2. REVIEW PROCESS

The first day of the review was spent walking through the overall architecture of the system, starting from the User Interface components, identifying the underpinning elements and their interrelationships. This overall architecture was assessed for "hot spots", i.e., critical elements of the system whose failure could most easily create the most grievous breaches. The hot spots identified were

- Kernel E: the compact representation into which all E code is translated before execution. A flaw in Kernel E could produce unpredictable vulnerabilities throughout the system.

- Universal Scope: if the Universal Scope, to which all caplets and library packages are granted access at startup, contained an inappropriate authority, this authority would undermine the confinement.

- Taming: The taming mechanisms are a wrapper for the Java API that suppress improperly conveyed authority, making it possible to acquire authority only through proper interaction with the user (typically through the Powerbox, described next). Improper authorities that escape suppression by taming are immediately available for all caplets and libraries, including the renderer. Two taming mechanisms are present in E: a legacy mechanism that is being phased out, and the SafeJ mechanism that is replacing it.

- The Powerbox: this is the component through which special powers are conveyed to caplets. If the Powerbox granted improper authority to the caplet (the DarpaBrowser in this case), there would be a risk that it could leak to the renderer, where it could be exploited.

- The Browser Frame: if the browser frame, which controls confinement of the renderer, leaks authority to the renderer, this is the basis for an immediate security breach.

The Browser Frame and the Powerbox were small enough to be reviewed line-by-line for risks. Kernel E and the Universal Scope were small enough to allow direct review of the critical core elements where the most serious risks were most likely to occur. The SafeJ system was too large for such a targeted review in the time available. Instead, it was analyzed by inspection of the documentation automatically generated for it from the SafeJ sources, and by “dipping in” at places that seemed likely to convey authority. Potential attacks were often confirmed or denied using the Elmer scratchpad that allowed the construction of quick experimental code passages in E.

3. THEIR APPROACH

This section describes the approach taken by Combex to build a system that could achieve the security goals of the project.

3.1 Capability model

The Combex team’s architecture is fairly simple from a high-level view: they build an execution environment that restricts the behavior of untrusted code—i.e., they build a “sandbox”—and they use this to appropriately confine the renderer. Once we have a sandbox that prevents the renderer from affecting anything else on the system, we can then carefully drill holes in the hard shell of the sandbox to let the renderer access a few well-chosen services (e.g., to allow it to draw polygons within the browser window). The crucial feature here is that by default the renderer starts with no access whatsoever, and then we allow only accesses that are explicitly granted to it. This is known as a “default deny” policy, and it has many security advantages: by following the principle of least privilege (also known as the principle of least authority, or POLA), it greatly decreases the risk that the renderer can cause harm by somehow exploiting the combination of powers granted to it. We want to emphatically stress that Combex’s “default deny” policy seems to be the right philosophy for the problem, and in our opinion anything else would carry significant risks.

So far this is fairly standard, but the real novelty comes in how Combex has chosen to implement its sandbox. Combex uses a capability architecture to restrict the behavior of the sandboxed renderer. In particular, every service an application might want to invoke is represented by an object, and each application can only use a service if it has a reference to that service. In E, references are unforgeable and are known as *capabilities*.

A crucial point of the capability architecture is that every privilege an application might have is conveyed by a capability. The set of operations an application can take is completely defined by the capabilities it has: i.e., there is no other source of “ambient authority” floating around that would implicitly give the application extra powers. To sandbox some application, then, we can simply limit the set of capabilities it is given when it comes to life. An application with no capabilities is completely restricted: it can execute E instructions of its choice (thereby consuming CPU time), allocate memory, write to and read from its own memory (but not memory allocated by anyone else), and invoke methods (either synchronously or asynchronously) on objects it has a capability/reference to.

Applications can be partially restricted by giving them a limited subset of capabilities. The E architecture enforces the capability rules on all applications.

Now the Combex game plan for confining malicious renderers is apparent. To prevent a malicious renderer from harming the rest of the system, we must simply be sure it can never get hold of any capability that would allow it to cause such harm. Note that this has a very important consequence for our security review. We need only consider two points:

- Does the E implementation correctly enforce capability discipline?
- Can a malicious renderer gain access to any capability that would allow it to violate the desired security policy?

Our review was structured around verifying these two properties.

The first point—full enforcement of capability discipline—requires reviewing the E interpreter and TCB (trusted computing base). We will tackle this in the next section.

The second point—evaluating the capabilities a malicious renderer might have—requires studying every capability the renderer is initially given and every way the renderer could acquire new capabilities. We will consider this in great detail later, but a few general comments on our review methodology seem relevant here.

First, listing all capabilities that the renderer comes to life with is straightforward. Because the renderer is launched by the DarpaBrowser, we simply examine the parameters passed into the renderer object when it is created, and because applications do not receive at startup time any powers other than those given explicitly to them (the “default deny” policy, as implemented by E’s “no ambient authority” principle), this gives us the complete list of initial capabilities of the renderer.

Identifying all the capabilities that a malicious renderer might be able to acquire is a more interesting problem. A malicious renderer who can access service S might be able to call service S and receive as the return value of this method a reference to some other service T. Note that the latter is a new capability acquired by the renderer, and if service T allows the malicious renderer to harm the system somehow, then our desired security policy has been subverted. In this way, a renderer can sometimes use one capability to indirectly gain access to another capability, and in practice we might have lengthy chains of this form that eventually give the renderer some new power. All such chains must be reviewed.

This may sound like a daunting problem, but there was a useful principle that helped us here: an application can acquire capability C only if the object that C refers to is “reachable” from the application at some time. By reachability, we mean the following: draw a directed graph with an edge from object O to object P if object O contains a reference to P (as an instance variable or parameter), and say that object Q is reachable from object O if there is a sequence of edges that start at O and end at Q. Consequently, reachability analysis allows us to construct a list of candidate capabilities that a malicious renderer might be able to gain access to, with the following guarantee: our inferred list might be too large (it might include capabilities that no malicious renderer can ever obtain), but it won’t be too small (every capability that can ever be acquired by any malicious renderer will necessarily be in our list). Then this list can be evaluated for conformance to the desired security policy.

We used static reachability analysis on E code frequently throughout our review. The nice feature of reachability analysis is that it is intuitive and quite easy to apply to code manually: one need only perform a local analysis followed by a depth-first search. In many cases, we found that some object O was not reachable from the renderer, and this allowed us to ignore O when evaluating the damage a renderer might do. We’d like to emphasize that knowing which pieces of code we don’t need to consider gave us considerable economy of analysis, and allowed us to focus our effort more thoroughly on the remaining components of the system. We consider this a decidedly beneficial property of E, as it allows us to improve our confidence in the correctness of the Combex implementation and thereby substantially reduce the risk of vulnerabilities.

In addition, since most components start with no authority (beyond the ability to perform computation, such as creating lists and numbers), even though they are transitively reachable from another component, they cannot provide additional authority to their clients (because they do not have any authority to give), and so cannot lead to a security vulnerability.

3.2 Security Boundaries

Confinement is not sufficient for the DarpaBrowser (and many other systems). Instead, a mostly confined object (the renderer) must be able to wield limited authority outside itself, across a security boundary that restricts the access of the confined object. The object that provides it that limited authority (the security management component) has substantially more authority (for example, the authority to render into the current GUI pane is a subset of the authority to replace the pane with another). Transitive reachability shows that the confined component could potentially reach anything that the security management component could reach, and indeed a buggy or insecure security management component could provide precisely that level of access to the intended-to-be-confined component.

In general, security boundaries between each of the components of the system are achieved almost for free using E. To allow object A to talk to B, but in a restricted way, we create a new object BFacet exporting the limited interface that

A should see, and give A a reference (a capability) to BFacet. Note that E's capability security ensures that A can only call the methods of BFacet, and cannot call B directly, since A only has a reference to BFacet and not to B.

The Combex system extends this style into a pattern that further simplifies analysis of confined components. The target component is launched with *no* initial authority, and is then provided a single capability analogous to the BFacet above, called the Powergranter, that contains the specific authorities that the confined component may use. The Powergranter becomes the only source of authority for the confined component and embodies the security policy across the boundary. Thus, the pattern makes it clear which code must be reviewed to ensure that the security policy is enforced correctly.

3.3 Non-security Elements that Simplified Review

This section describes some elements of the E design that were not motivated by security, but that contributed either to security or the ease of reviewing for security.

3.3.1 E Concurrency Model

The review was substantially simplified by the concurrency model in E. In the E computational model, each object only ever executes within the context of a single Vat. Each Vat contains an event queue and a single thread that processes those events. Messages between objects in different vats use an "eventual" send, that immediately returns to the sender after posting an event on the receiver's vat for the message to be delivered synchronously within that vat. As a result, objects in E never deal with synchronization. Consequently, all potential time-of-check-to-time-of-use (TOCTTOU) vulnerabilities could be evaluated within a single flow of control, and thus took little time to check for. By contrast, in systems in which multiple threads interact within objects, such determinations can be extremely difficult or infeasible to determine.

3.3.2 Mostly-functional Programming Support

An interesting side note is that E's support for mostly functional programming seems to have security benefits. Mostly-functional programming is a style that minimizes mutable state and side effects; instead, one is encouraged to use immutable data structures and to write functions that return modified copies of the inputs rather than changing them in place. (Pure functional programming languages allow no mutable state, and often also stress support for higher-order functions and a foundation based on the lambda calculus. E does provide similar features; however, these aspects of functional programming do not seem to be relevant here.) The E library provides some support for functional programming in the form of persistent (immutable) data structures, and we noticed that E code also seemed to often follow other style guidelines such as avoiding global variables. This seems to provide two security benefits.

First, immutable data structures reduce the risk of race conditions and time-of-check-to-time-of-use (TOCTTOU) vulnerabilities. When passing mutable state across a trust boundary, the recipient must exercise great caution, as the value of this parameter may change at unexpected times. For instance, if the recipient checks it for validity and then uses it for some operation if the validity check succeeds, then we can have a concurrency vulnerability: the sender might be able to change the value after the validity check has succeeded but before the value is used, thereby invalidating the validity check and subverting the recipient's intended security policy. Similarly, when passing mutable state to an untrusted callee, the caller must be careful to note that the value might have changed; if the programmer implicitly assumed its value would remain unchanged after the method call, certain attacks might be possible. Our experience is that it is easy to make both of these types of mistakes in practice. Using immutable data structures avoids this risk, for if the sender and recipient know that all passed parameters are immutable then there is no need to worry about concurrency bugs. To the extent that E code uses immutable data structures, it is likely to be more robust against concurrency attacks; we observed in our review that when one uses mutable state, vulnerabilities are more common.

Second, the use of local scoping (and the avoidance of global variables) in the E code we reviewed made it easier to analyze the security properties of the source code. In particular, it was easier to collect the list of capabilities an object might have when we only had to look at the parameters to its method calls and its local instance variables, but not at any surrounding global scope. Since finding the list of possessed capabilities was such an important and recurring theme of our manual code analysis, we were grateful for this aspect of the Combex coding style, and believe that it reduced the risk of undiscovered vulnerabilities.

4. ACHIEVING CAPABILITY DISCIPLINE

In this section, we evaluate how well E achieves capability discipline, i.e., how effective it is as a sandbox for untrusted code. The crucial requirement is that the only way an application can take some security-relevant action is if it has a capability to do so. In other words, every authority to take action possessed by the application must be conveyed by a capability (i.e., an object reference) in the application's possession.

There is a single underlying principle for evaluating how well E achieves this objective: there must be no "ambient authority". Ambient authority refers to abilities that a program uses implicitly simply by asking for resources. For example, a Unix process running for the "daw" user comes to life with ambient authority to access all files owned by "daw", and the application can exercise this ability simply by asking for files, without needing to present anything to the OS to verify its authority. Compare to E, where any request for a resource such as a file requires not just holding a capability to the file (and applications typically start with few authorities handed to them by their creator), but also explicitly using the capability to the file in the request.

We start by looking at the default environment when an application is started. In E, the environment includes the *universal scope*, a collection of variable/value bindings accessible at the outermost level. If the universal scope included any authority-conveying references, the "no ambient authority" principle would be violated, so we must check every element of the universal scope. We discuss this first. We shall also see that a critical part of the universal scope is the ability to access certain native Java objects, so we devote considerable attention to this second. Finally, we examine the E execution system, including the E language, the E interpreter, the enforcement of memory safety, and so on.

4.1 UniversalScope

The E language is quite spare, and many programming features that would be part of the language syntax in other languages are pushed to the universal scope in E. The E universal scope contains values for constructing basic values (such as integers, strings, lists of integers, and so on), promises, references, and exceptions. It also contains some utility functionality, for regexp matching, parsing of E scripts ("eval"), and so on. It contains support for control-flow constructs: loops, and the ability to throw exceptions. It also contains objects that let one control the behavior of E. All of these seemed appropriate and safe to grant to untrusted applications.

The universal scope also allows applications to create a stack trace, for debugging purposes. Such a backtrace would not reveal the value of internal variables of other stack frames, but could potentially reveal information present at the site of the exception. For example, an inner confined object could throw an exception containing a capability that was confined (e.g., a private key or database), through an intermediate caller, to a colluding outer object, thus breaking confinement. Also, the depth of the execution stack is visible, which could pose a risk in certain scenarios: for instance, consider trusted code containing a recursive function whose level of recursion depends on some sensitive data (e.g., a secret cryptographic key), and suppose the recursive function is called with arguments that induce it to hit an error condition and throw an exception from deep within the recursion. In such a case, the caller might be able to learn something about the callee's secrets by catching the exception, examining the resulting stack trace, and recovering the stack depth. These scenarios do not occur in the DarpaBrowser, but have been used in exploits on other systems. Accordingly, though the risk for DarpaBrowser is small, it should probably be repaired (Fixing this was determined not to be hard).

We note that the universal scope provides no way to gain access to the network, to remote objects, or to the hard disk (apart from the ***`resource__uriGetter`***; see below). Moreover, it is an invariant of the DarpaBrowser implementation that the renderer never receives any remote references nor any way to create them;

consequently, though the E language contains support for distributed computation, we do not need to consider this aspect in our review of the renderer sandbox.

There is one detail we have not discussed yet: the universal scope also contains several objects known as *uriGetters*, which deserve extra attention. Every application receives two such uriGetters: a ***`resource__uriGetter`***, and an ***`import__uriGetter`***.

The ***`resource__uriGetter`*** allows applications to load application-specific resources, such as an image file containing their icon and so on. The application is not (by default) allowed to modify those resources, and the resources are supplied when the application is installed on disk. This data is stored in a special directory on disk. Thus, the application can effectively read to this very special part of the file system, but not to any other files, nor can the application write to these files. Given the contents of these resources, the ***`resource__uriGetter`*** seems to pose little risk.

The ***`import__uriGetter`*** gives the application access to the E library (collections, messages, and more, all in E), the E interpreter and parser, and various utility code (e.g., a Perl5 regexp matcher and a XML parser). The ***`import__uriGetter`*** also allows the application to load certain Java classes, instantiate them, and obtain a reference to the resulting Java object. Once the application has a reference to such a Java object, the application can make method calls on that object and interact with it. Such Java objects run unrestricted, not under the control of the E architecture, and hence must be trusted not to violate capability discipline or otherwise convey authority. Since interaction with Java objects obviously provides a potential way in which a malicious renderer might be able to subvert the E sandbox, we discuss this very important aspect of E in detail next.

4.2 Taming the Java Interface

One of the goals of the E architecture is to allow Java programmers to easily transition to writing E code, and in particular, to continue using familiar Java libraries. For example, E lets programmers use the Java AWT and Swing toolkits for building graphical user interfaces. This means that E code needs access to legacy Java classes. This comes with significant risks, as the Java code may not have been written in accordance with capability discipline, and since Java code is not subject to E's capability enforcement mechanisms, this might allow security breaches if care is not taken. Unfortunately, not all Java classes are safe to give to untrusted applications: some would allow the renderer to escape the sandbox. For instance, calling ***`new File("/etc/passwd")`*** would give the renderer access to the password file on Unix, and hence sandboxed applications must not be allowed to call the constructor method on ***`java.io.File`***.

The E solution is to restrict the application's access to Java classes and methods. There are several mechanisms for enforcing these restrictions: SafeJ, the legacy mechanism, and fall-through behavior. The legacy mechanism hard-codes policy for a few Java classes. (The listed classes are considered as safe but with all methods allowed, unless there is a specific annotation giving the list of allowed methods, in which case all unmentioned methods are suppressed.) SafeJ, the successor mechanism, is more flexible, and will be described next.

In SafeJ, Java classes can be marked either *safe* or *unsafe*. All applications are allowed to invoke the allowed static methods and constructors of any class marked safe via the ***`import__uriGetter`***, which is available to all applications from the universal scope. Consequently, classes marked *safe* can typically be instantiated freely by all applications. In contrast, the only way to invoke static methods or constructors of unsafe classes is through the ***`unsafe__uriGetter`***, which is not part of the universal scope and should not be available to untrusted applications. Consequently, this lets us control which Java classes can be instantiated (e.g., by calling their constructor) by untrusted applications.

This coarse-grained measure is not enough by itself, of course, because some Java classes have a few methods that follow capability discipline and a few that do not[1]. We could simply mark these classes unsafe, but such a conservative approach would deny applications access to too much useful functionality. Instead, SafeJ allows each public method to be marked either *suppressed* or *allowed*. (Private and package-scope methods are treated as implicitly suppressed. Public constructors are handled as static methods with the method name "***`new`***", and suppressed or allowed as with any other method. Instance variables are wrapped with getter and setter methods, and then the methods are handled as before, with one exception: public final scalar variables are always implicitly marked allowed. Methods that aren't listed in the SafeJ database but are part of a class that is listed in the database are treated as implicitly suppressed.) Applications are only allowed to call unsuppressed methods. (No E application can directly call a suppressed method, no matter what capabilities it

[1] It is a little tricky to define exactly what it means for a method to follow capability discipline. Imagine if java.lang.String had a static method called formatHardDrive() that erased the entire filesystem. Would this be a failure of capability discipline? One could argue that the java.lang.String class is an abstraction of the entire hard disk and hence any reference to it conveys authority to delete the entire hard disk; this would be following capability discipline. (Such an interpretation would undoubtedly be surprising and confusing to many programmers: one might expect an instance of a class named java.io.HardDisk to represent the hard disk, and a java.io.HardDisk.formatHardDrive() method would be reasonable and expected, but surely not on a java.lang.String. This highlights the difficulty of declaring java.lang.String.formatHardDrive() a violation of capability discipline in any principled way, as the only real difference between java.lang.String and java.io.HardDisk is the name of the class.) However, in practice it is easy to detect a violation of capability discipline. We expect each Java object to represent some abstract or real-world entity, service, or resource; a reference to that Java object conveys authority to the represented entity, and we may reasonably insist that method calls should only allow the caller to affect the represented entity, and nothing else. Thus, a java.io.File object represents a file on the hard disk, a javax.swing.JEditorPane represents an editing window on the screen, and so on. Many—but not all—Java objects mostly respect this intuitive notion of capability discipline.

has.) Java classes that have been made safe to export to E in this way are sometimes known as *tamed* classes.

Finally, if a class is not listed in the SafeJ database and not controlled by the legacy mechanism, the fall-through behavior applies. In particular, the default for such classes is that they are treated as though they had been marked unsafe (so they can only be instantiated by applications trusted with the ***`unsafe__uriGetter`***), but all methods are treated as allowed.

Consequently, the presence of the ***`import__uriGetter`*** in the universal scope and the other ways of obtaining references to Java objects give all applications access to a great deal of trusted Java code, and this must be carefully reviewed. In particular, a sandboxed application can invoke any allowed static method or constructor on any Java object marked safe, and so can typically instantiate these, and can call any unsuppressed method on any object it has obtained a reference to (either by instantiating that object itself, or by obtaining such a reference indirectly from other objects). To ensure that untrusted applications cannot escape the sandbox, we must examine every unsuppressed method on every safe Java class to be sure that they do not contain ambient authority.

An additional design goal of E was that unsuppressed methods do not permit reading covert channels: for example, E applications should not be able to get access to any information about the current time or to access any form of non-determinism (the latter restriction helps avoid covert channels, and also makes deterministic checkpointing easier), and there should be no global mutable state shared between applications (except where such state is explicitly exchanged between communicating applications). Java code can potentially violate these restrictions, and we spent a little time on reviewing these properties. However, because these restrictions are not necessary for the security of the DarpaBrowser exercise, we did not put much attention into them.

We reviewed a great deal of Java code. However, there is simply too much accessible Java code to review it all. Therefore, we used various methods to identify classes likely to be at greatest risk for security breaches, and focused our effort there. Also, we reviewed the process by which Combex authorized classes as safe and methods as unsuppressed to look for any systematic risks. We will describe first the result of our focused code analysis efforts, and then discuss the process.

4.3 Security holes found in the Java taming policy

We found a vulnerability: ***`java.io.File`*** does not follow capability discipline, yet its methods are allowed by the legacy mechanism. In particular, the getParentFile() method returns a File object for the parent directory, and thus given a reference to any one file on the filesystem a malicious application could use this method to get a reference to any other file on the filesystem, breaking confinement or worse. Obviously the getParentFile() method ought to be suppressed. Moreover, we suggest that File objects should be fully opaque with respect to their location in the filesystem, and methods such as getAbsolutePath(), etc., should also be suppressed. How did this error arise? The ***java.io.File*** class was mediated by the legacy mechanism, and because no method was annotated, by default all methods were inadvertently allowed. This illustrates a general risk in the legacy mechanism: it uses a “default allow” policy, which is very dangerous and should be avoided; in contrast, SafeJ uses a “default deny” policy, which is much better suited to security.

We found a second security weakness: sandboxed applications can call show() on their AWT and Swing windows and grab the focus. As a result, a malicious renderer could steal the input focus from some other window; for instance, if the user is typing her password into some other window, a malicious renderer might be able to steal the focus from the other window and thereby observe the user’s next few keystrokes before the user notices what has happened. The show() method was suppressed in some classes, but unsuppressed in others, so there are some holes. We suggest that all show() methods should be suppressed. Moreover, it may be a good idea to check if any other methods internally call show(); if so, they should be suppressed, too.

We found a significant vulnerability: the ***javax.swing.event.HyperlinkEvent*** class, which is marked safe, contains an allowed method getURL() that returns a ***java.net.URL*** object. Since ***java.net.URL*** objects convey the authority to open a network connection and fetch that URL, returning such an object gives the application considerable power. We believe that this was not intended, and in particular, it may allow a malicious renderer to violate the desired security policy as follows. Suppose the malicious renderer registers itself to receive ***HyperlinkEvent*** callbacks whenever the user clicks on a link in the renderer window. Then if the malicious renderer can lure the user into clicking on some link in this window, the renderer can receive authority to fetch the corresponding URL. In this way, a malicious renderer could arrange to fetch any URL linked to on the current web page, without involving the ***CapBrowser***. This allows a malicious renderer to download such a URL and display it, even though the URL field in the browser is not updated. The renderer can continue in this way, fetching any URL transitively reachable among the tree of links rooted at the current page, and thus can gain access to all of the web that is reachable from the current page. This would allow a malicious renderer to violate the “synchronization” security goal by rendering one web page while the URL displayed above refers to another.

We found another, much less serious, weakness: ***java.awt.ComponentEvent*** contains an allowed method, ***getComponent()***, that works as follows. If the user interacts with some AWT component (e.g., the browser window) in some way (say, by resizing it or clicking within it), then an AWT event is generated and sent to all listeners. The listener can find out which component this event is relevant to by calling the ***getComponent()*** method to get a reference to the appropriate AWT component object. Note that the ***CapBrowser*** is able to register a listener on its containing window. This could allow a malicious browser to escape its confinement and draw outside its window. Our hypothetical malicious browser would first register a listener on its parent window. Modifying the parent window is off-limits for the browser, because the parent window contains elements mediated by the PowerGranter. . If the user can be lured into clicking or otherwise generating an event relevant to the parent window, the malicious browser will receive a ***java.awt.ComponentEvent*** object that can be queried via the ***getComponent()*** method to return a reference to the parent window. Please note that this affects only the confinement of the ***CapBrowser***, and not of the renderer, because the renderer has no way to set a listener on the parent window. Since confining the browser was not a goal of this project, this weakness seems to have no impact on the security of the Combex DarpaBrowser. We mention it only for completeness. We re-discovered a vulnerability that was already known to the Combex team: the ***java.awt.Component*** class has ***getDropTarget()*** and ***setDropTarget()*** methods, marked allowed. This allows a malicious renderer to subvert the trusted path to the user, spoof drag-and-drop requests from the user, and steal capabilities that the renderer should not be able to get access to. In the Combex CapDesk system, the user is considered trusted, and if the user uses the mouse to drag-and-drop a file onto an application, this is considered as a request by the user to authorize that application to access the corresponding file. In Java, when a window is the target of a user's drag-and-drop request, if that window has a DropTarget enabled (which can be done by calling ***setDropTarget()***), then a drag-and-drop event will be created and sent to the DropTarget object. Consequently, drag-and-drop events are treated as trusted designators of authority, and thus Java's DropTarget management is a critical part of the trusted path to the user. However, an E malicious renderer might be able to spoof drag-and-drop requests by calling ***getDropTarget()*** and then sending a synthetic drag-and-drop event to the drop target. More seriously, a malicious renderer could steal file capabilities by registering itself as a DropTarget for some window (note that effectively all windows and GUI elements are subclasses of ***java.awt.Component*** and thus have their ***setDropTarget()*** method allowed) and then proceeding to receive each drag-and-drop event and extracting the capability for the file designated by the user. This may give the malicious renderer access to files it should not have received, and the user may not even realize that the malicious renderer is gaining access to these files, as the user may have intended to drop that file onto the DarpaBrowser, not the renderer.

We found an unimportant minor violation: ***java.lang.Object.toString()*** may enable covert channels, since it can reveal the hash code of the object, an undesired property (it ruins the equivalence of transitively immutable objects). Similarly, by grepping the SafeJ database we found about four unsuppressed

hashCode() methods, about 20 unsuppressed ***equals()*** methods, and about thirteen unsuppressed ***toString()*** methods. (Most ***hashCode()***, ***equals()***, and ***toString()*** methods were correctly suppressed, but a few apparently slipped through the review process.) We note that this does not affect the security of the DarpaBrowser. We mention it only to point out the difficulty of perfectly taming the Java interface: though suppressing these types of methods seemed to be a goal of the taming process, a few instances were overlooked inadvertently during the taming process, and so one might reasonably conclude that the taming process is potentially somewhat error-prone.

We found another similar minor violation: it may be possible to use the java.awt.Graphics2D class as a covert channel, because the ***java.awt.RenderingHints*** object stored in the graphics context acts as globally accessible, mutable state. In particular, the ***RenderingHints*** class behaves much like a Java hashtable, with put() and get() methods. To remedy this, the get() methods were all suppressed, apparently in an effort to make ***RenderingHints*** behave like a piece of write-only state (more precisely, the intent was apparently that E applications should only be able to write to it, whereas Java objects could both read from and write to it). However, this intent was not carried out perfectly, as we found that ***RenderingHints*** has an unsuppressed remove() method that takes a key and returns the value previously associated with that key. This can allow E applications to read the contents of this pseudo-hashtable. Similarly, a put() method also returned the previous value associated with a key, and this unsuppressed method could also create a covert channel. Because covert channels are not relevant to the DarpaBrowser, this small bug does not affect the security of the DarpaBrowser; we mention it only for completeness.

Along similar lines, it turned out that though the get() methods of ***RenderingHints*** were marked suppressed in the SafeJ database for the ***RenderingHints*** database, they weren't actually denied to the application. In fact, the application could call the get() methods nonetheless, and this seems to be because a superclass of ***RenderingHints***, Map, controlled by the legacy taming mechanisms, effectively marked get() as allowed and the superclass's annotation took precedence in this case. Again, this does not affect the security of the DarpaBrowser, but is a defect that should be repaired.

We found a minor risk: the AWT keyboard focus manager (***java.awt.KeyboardFocusManager***) was declared safe and allows the ***getFocusKeyboardManager***() method, which might allow stealing the focus or even stealing certain keypress events—while we do not know of any attacks, we do not have any confidence that attacks are impossible, and so we recommend that this be marked unsafe. (Also, the Swing keyboard focus manager, ***javax.swing.FocusManager***, is declared unsafe but has other similar methods allowed.) This does not appear to cause a risk for the DarpaBrowser, because these methods are only implemented in JDK 1.4 while the Combex system is designed to run on JDK 1.3. However, there does seem to be an important lesson here. The Java taming decisions are specific to a single version of Java, and upgrading the Java libraries could create significant vulnerabilities if those decisions are no longer appropriate. (We noticed several instances of this risk: for instance, ***`java.awt.event.MouseEvent.getButton()`*** is allowed and is dangerous, but fortunately appears only in JDK 1.4 so cannot be used in an attack when E is run using JDK 1.3, as it is for the DarpaBrowser.) Consequently, we recommend that Combex warn users that E should only be used on the intended version of Java and take steps to defend against this risk if Combex should ever decide to upgrade to a later version of Java.

We identified several risks associated with the javax.swing.Action interface, which encapsulates the ability to perform certain user interface actions (such as a cut-and-paste operation). Many classes have getAction() and getActions() methods allowed, which might leak a surprising capability to invoke one of these actions to the malicious renderer. Similarly, many classes have setAction() and setActions() methods allowed, which might allow a malicious renderer to change the action associated with a user interface event and thereby change the behavior of some trusted interface. We did not have time to look at all code paths for all possible attacks, but we are worried about the risk of exposing these capabilities to a malicious renderer: thus, this is not a known risk, but rather is a “risk of the unknown”. We suggest that these methods should be suppressed unless there is a convincing argument that they are safe.

4.4 Risks in the process for making taming decisions

From the taming bugs we found, we can see several general risks associated with the E taming mechanism and the process used by the Combex team to make taming decisions.

First, the use of three different taming mechanisms is dangerous, as it can cause confusion. And the “default allow” policies of the legacy mechanism and the fall-back default are quite worrisome. For instance, the java.io.File.getParentFile() vulnerability described above arose directly as a result of a “default allow” policy, and we are concerned that there may be other vulnerabilities of this sort hiding undiscovered.

Second, the complexities of Java contribute to this danger. For instance, consider subclassing. Suppose we suppress some method in the superclass using SafeJ. Later, we encounter a subclass where this same method is overridden. SafeJ will prompt us whether this should be allowed just as if it were any other method, yet it clearly is not: if we’ve suppressed it in the superclass, that’s an indication that something unusual is going on here, and in this case, we feel that the user ought to be warned. The fact that the user is not warned in this case introduces the risk that the user might inadvertently allow this method in some subclass. Any time that humans must make the same decision several times is an invitation for one of those decisions to be incorrect, and all the adversary needs is a single error. We saw several instances of this failure mode in practice: in the hashCode(), equals(), and toString() methods. A related failure can occur if subclasses have superclasses with methods that are dangerous. It may not always be obvious when reviewing the superclass that the method is dangerous, and the user might decide to allow the method while taming the superclass and never consider it again, when it should be denied in a subclass. Of course, in practice if the method is dangerous in some subclass it is prudent to suspect that it may be dangerous in other subclasses as well. We saw an example of this with the getComponent() method, which was enabled in java.awt.ComponentEvent() but suppressed in some subclasses. One possible solution to these two problems might be as follows: if a method is allowed in the superclass, then the user is still queried about it in all subclasses, but if the method is suppressed in the superclass, then it will be suppressed in all subclasses as well. More research in this area would seem to be helpful, though.

A more dangerous bug related to subclassing is that if a method is allowed in the superclass, overridden in some subclass, and marked as suppressed in that subclass, then the method is still treated by E as allowed. This was responsible, for instance, for the RenderingHints vulnerability. We view this as simply a design mistake in the semantics of SafeJ, and recommend that it be fixed.

In general, the taming mechanism is too complicated for us to have much trust in our review of it. There are special cases for public final scalar variables, for sugared classes, for unsafe classes not in the SafeJ database (the fall-through behavior), and so on. It is not easy to find a single database listing all the taming decisions (the SafeJ one is incomplete, because it doesn't handle the legacy mechanism, the fall-through behavior, or the other special cases), and this makes it hard to review earlier taming decisions. In addition, the complexity of Java adds extra potential for unexpected behavior. We are uneasy about all these possible interactions, and we urge that more attention be paid to this critical aspect of the E architecture.

4.5 E

The final piece of the E architecture that is needed to build a secure sandbox is the E language. E is an interpreted language, and the interpreter is of course trusted to enforce the sandboxing restrictions. Due to time limitations, we did not review the entire implementation of the interpreter; instead, we focussed on understanding the semantics of a few critical aspects of E.

First, we examined all Kernel E constructs. The E language is actually translated down to a subset, called Kernel E, before execution, and so reviewing Kernel E is sufficient to check whether there are any instructions that might allow a sandboxed application to break out of its sandbox. We did not find any worrisome instructions.

We reviewed certain aspects of the E parser and interpreter. We found that the architecture seems to be well-suited to enforcing memory safety, and we would not expect to find any vulnerabilities here. We did find one bug: the E parser memorizes the result of parsing, so that parsing the same script a second time will immediately return a cached copy of the parse tree. This is an important performance optimization. However, the memo-ization was not based on all of the relevant parsing context: scripts were matched on their text but not on extra arguments that affected the parsing process. As a result, it was possible to cause the E parser to return incorrect parse trees. This does not appear to lead to a security vulnerability in the context of the DarpaBrowser, but we suggest a systematic review of all uses of caching to check for other instances of this error and to make sure that all cached data is transitively immutable.

Next, a subtle aspect of E is that a few extra synthetic methods are constructed and added to every object. These are known as *Miranda methods*, and they permit functionality such as asking an object to return its type. We were initially concerned that this creation of artificial methods not specified by the programmer might allow attacks, but after reviewing all the Miranda methods, we did not find any reason for concern.

Along the way, we discovered a few interesting properties of the E language. The first relates to exceptions. In Java, there are two kinds of exception-like types: exceptions, and errors. (The former are normally expected to be repairable, while the latter indicate more serious conditions.) In E, one can throw and catch any Java exception, but not Java errors. This means that if one calls a Java object inside an E try-catch construct and the Java code throws an error, the E catch construct will not be triggered, and the error will propagate up to the top-level. An obvious next question is: What is at the top level? In E, the top level is an E object that services the run queue and executes each queued task. If execution of some task throws an uncaught exception, this is caught by the E run-queue servicing routine, execution of that task halts, and the top level moves on to the next task in the queue. However, if an error is thrown during processing of some task, this is not caught by the E run-queue servicing routine (errors cannot be caught by E code), and hence the error will terminate the Vat. This means that malicious code might be able to kill the containing Vat, perhaps by calling some Java code and tricking it into throwing an error. This is a denial-of-service attack. Of course, in E denial-of-service attacks are out of scope, but we thought we would mention this unexpected property anyway for completeness.

The run-queue is an interesting entity. All E code can append tasks to it by issuing eventual sends, and the top-level run-queue servicing routine will dequeue these one by one and execute them in turn. Thus, on first glance the run-queue might appear to be a piece of mutable shared state. Fortunately, it turns out that this seems to be safe: there does not seem to be any way to use this to communicate between confined E applications, and so the existence of a global run-queue does not violate E's security goals. We suggest that verifying and possibly generalizing this property might make for an interesting piece of research, though. Also, we note that every application can enqueue tasks for execution. This is a form of ambient authority, albeit one that does not seem to cause any harm. Nonetheless, in keeping with capability principles, we feel required to ask: should applications be required to have a “enqueue” capability before being allowed to enqueue tasks for later execution? We think the answer is “no”, but perhaps there could be cases where one would like to deny an application the ability to issue eventual sends, and in this case such a capability would be useful.

The most surprising (to us) property of the E language is that nearly everything is a method call. For instance, the E code “a+b” might appear like primitive syntax that invokes the addition operator, but in fact in E it translates to a method call to object “a” asking “a” to add “b” to itself. Similarly, comparisons also translate into method calls. This has implications for the construction of trusted code. Suppose, for example, we build trusted code that accepts an integer, tests to make sure that it is not too large, and passes it on to some other entity as long as this check succeeds. Suppose also that the interface is exported to untrusted applications. Unless great care is taken, it will likely be possible to fool this trusted code: though the programmer might expect its argument to be an integer, there is typically nothing enforcing this. An attacker might pass in a specially constructed object, not an integer; this object might respond “true” to the first comparison methods, so that the trusted code’s size check will succeed, but otherwise behave as though it were a very large integer, so that when the object is passed on to the final entity it will be treated as a big number. This is an example of a simple time-of-check-to-time-of-use (TOCTTOU) flaw.

Our experience was that this sort of TOCTTOU error was surprisingly easy to commit. For example, we found at least one successful attack on the DarpaBrowser based on this idea; see below for details. As a result of this analysis, we suggest a new design principle for writing secure E code: whenever one relies on values from an untrusted source, one should explicitly check that those values have the expected implementation-type (using, e.g., an E type guard on the parameter list for the method). Moreover, when executing any expression like “a+b”, the result is according to object “a” and so is no more trustworthy than “a” (though “b” often does not need to be trustworthy). Though it is reasonable to trust the behavior of base implementation-types like integers and strings, one should exercise care with user-defined implementation-types. This, of course, is an issue more about how to build secure user-defined code that is exposed to untrusted applications, and does not impact sandboxed applications which have no access to any trusted services.

Finally, E separately considers confinement of capabilities (the authority to do things) from confinement of bits. For bit confinement, E also separates the consideration of outward confinement (can a process write on a covert channel?) from inward confinement (can a process read a covert channel?). The mechanisms described above provide confinement of capabilities. E further supports inward bit confinement in some limited circumstances. The mechanisms for this are not described here because the requirements of the DarpaBrowser put it outside the circumstances in which bit confinement is possible (specifically, it can get to external sources of time and non-determinism) and covert bit channels are out of the project's scope.

4.6 Summary review of Model implementation

Our general conclusion is that the E architecture seems to be well-chosen for sandboxing untrusted applications. It seems to have several advantages over the more familiar Java sandbox, which might allow it to provide a higher level of security than Java does. Moreover, though we did not have time to exhaustively review all source code, we found that the implementation seems to live up to these goals (modulo a few repairable implementation bugs). E's elegant design is marred only by the need for taming: the integration with legacy Java code is a significant source of serious risks.

5. DARPABROWSER IMPLEMENTATION

This section describes the DarpaBrowser architecture, and reports on the analysis and review of its implementation.

5.1 Architecture

The observed high-level architecture of the DarpaBrowser is shown below, with particular focus on where the various security features were used.

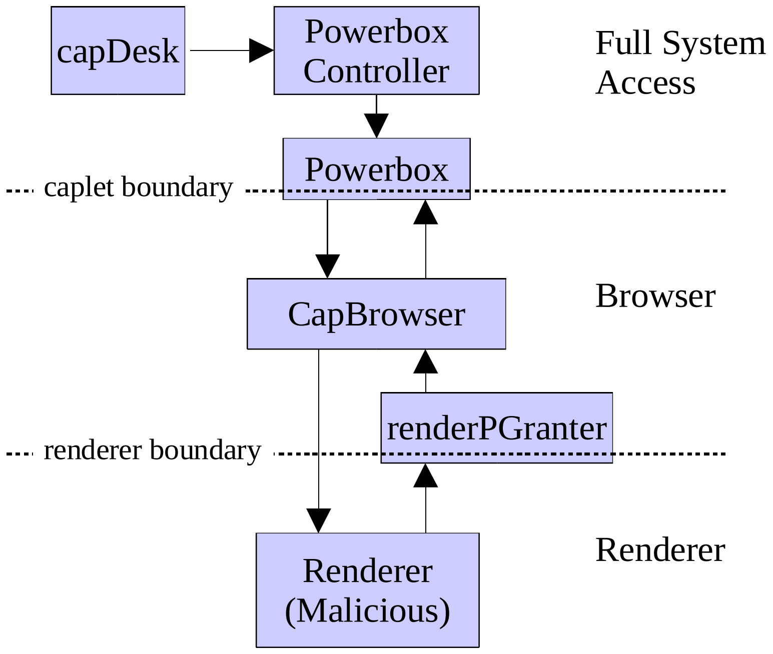


Figure 1 DarpaBrowser Overview

The DarpaBrowser architecture is fairly simple. The renderer is created by the ***`CapBrowser`***, with access to the ***`RenderPGranter`*** and no other authority. The ***`RenderPGranter`*** acts as the Powergranter for the Renderer; it enforces the security boundary and ensures that the renderer stays confined. In addition, the entire browser runs as a confined object in the CapDesk system. The Powerbox acts as the Powergranter for the CapDesk system to the DarpaBrowser. Thus, the architecture diagram shows the high-level components for each of the various layers, and shows where the security boundaries are between the layers.

The security goals are accomplished as follows. When the ***`CapBrowser`*** wants to view a new web page, it instantiates a new renderer and passes it a capability (reference) to the ***`RenderPGranter`***. In the implementation we reviewed, there is only one ***`RenderPGranter`*** per ***`CapBrowser`***, so all renderers share the same ***`RenderPGranter`***. The ***`RenderPGranter`*** allows renderer to invoke certain powers described below, such as changing the displayed page. When the Renderer wants to display a new page, it requests that the new page be displayed via an appropriate call to the ***`RenderPGranter`***. This ensures that the renderer and ***`CapBrowser`*** can update the address bar prior to showing the new contents.

We can see from the diagram that the renderer has the authority of the ***`RenderPGranter`***, and might have been handed authority from the

CapBrowser. However, it cannot reach the ***PowerboxController*** or the ***CapDesk***: there is no path from the renderer to either of these components. This illustrates the power of reachability analysis: we can already see that the only components we need to analyze are the ***RenderPGranter***, the ***CapBrowser***, and the ***Powerbox***.

5.2 What are all the abilities the renderer has in its bag.

One of the first tasks we performed in our review was to identify all the abilities the renderer has in its "bag of powers". This was identified by a reachability analysis. We found the following:

1 The renderer can call a tracing service to output debugging messages.

2 The renderer get a reference (capability) to the **javax.swing.JScrollPane** object representing the renderer's portion of the browser window. Note that the renderer does not receive a capability to the URL field or the containing window, but does obtain a capability to the scrollbar.

3 The renderer can request the browser to change to a new URL by calling **RenderPGranter.gotoURL()** and passing the URL as a string. This will cause the **CapBrowser** to take several actions to keep the URL field in sync and to enforce the desired security policy. The response of the **CapBrowser** will be discussed in detail later.

4 The renderer can request that the browser fetch an image for it. However, this is not yet implemented, and so no action is taken other than printing a debugging message.

This is the list of all its powers. The first two are implemented as follows: the ***RenderPGranter*** has a hashtable of object references that the renderer can request. (In practice, the hashtable will have exactly two elements: a reference to the tracing method and a reference to the scroll pane.)

5.3 Does this architecture achieve giving the renderer only the intended power?

We note that, if these services are implemented correctly by the ***RenderPGranter***, the ***CapBrowser***, and the ***Powerbox***, then the renderer will be successfully confined. In other words, this gives a workable architecture for achieving the desired security goals.

Next, we discuss how well the implementation of these three components lives up to the architecture discussed above.

5.4 CapDesk

As mentioned above, the CapDesk is unreachable from the renderer, and hence a malicious renderer cannot exploit it in any way to escape the sandbox.

5.5 PowerboxController

A similar comment goes for the ***PowerboxController***. Of course, the ***PowerboxController*** can help us achieve extra security by giving the ***CapBrowser*** only the limited set of capabilities it will need. Withholding capabilities from the ***CapBrower*** is doing it a favor: reducing the powers of the ***CapBrower*** means that the ***CapBrowser*** cannot accidentally pass on those powers to the renderer, even if there are bugs in the implementation of the ***CapBrowser***. This is a direct application of the principal of least privilege (give no more authority than necessary). It provides belt-and-suspenders security: even if the ***CapBrowser*** fails to implement the intended security restrictions on the renderer and somehow leaks all its powers to the renderer, the impact of such a failure will be limited to (at most) only those powers granted to the ***CapBrowser*** by the ***PowerboxController***. In the Combex implementation, the ***PowerboxController*** and ***Powerbox*** do indeed limit the set of powers given to the ***CapBrowser***, allowing it to create new windows and to fetch URLs across the network but not to write to arbitrary files on the hard disk. This “defense in depth” is a very beneficial feature.

5.6 Powerbox

The ***Powerbox*** further limits the capabilities granted to the ***CapBrowser***. It allows the ***CapBrowser*** to call a restricted set of nineteen methods on Java AWT/Swing windows, to load arbitrary files under the ***CapBrowser***’s directory, to load, run, and install new applications on the fly, to register itself as a target of drag-and-drop operations, to request the user to approve access to a file on the hard disk, and to request cut-and-paste operations. (The cut-and-paste operations are mediated by the ***Powerbox***, and hence not under total control of the ***CapBrowser***. This is because cut-and-paste operations can convey authority from the user to the ***CapBrowser***.)

We found one bug in the ***Powerbox***: the capability granted to the ***CapBrowser*** to read arbitrary files under its subdirectory was mis-implemented. In particular, the ***Powerbox*** implemented this restriction by building a method that took a relative filename, appended it to the ***CapBrowser***’s directory, loaded this file (using the ***Powerbox***’s extra powers), and returning the result to the ***CapBrowser***. However, if the ***CapBrowser*** requested a filename of the form “../foo”, then the ***Powerbox*** would happily let the ***CapBrowser*** read a file outside of its directory, in violation of the ***Powerbox*** programmer’s intentions.

Though this flaw does not lead to vulnerabilities in the security goals of the project, it does reduce the benefit of the “defense in depth” accorded by confining the ***CapBrowser***. If some security hole in the ***CapBrowser*** interface were found that allowed a malicious renderer to compromise the ***CapBrowser***, the flaw in the ***Powerbox*** would heighten the impact of the second security hole. However, as we did not find any flaw of this sort in the ***CapBrowser***, this flaw in the ***Powerbox*** can be best seen as a reduction in assurance (the level of confidence we have that the system is secure) rather than a serious security problem.

In any case, this bug was merely an implementation flaw, and in fact can be seen as further substantiation for the value of the E capability architecture. This bug came from a failure to follow capability discipline. Capability discipline would be to return a ***java.io.File*** object representing the ***CapBrowser***'s directory, rather than using strings to designate files; doing checks on endpoint objects is usually safer than doing checks on strings.

5.7 CapBrowser

The ***CapBrowser*** exports a method, ***gotoURL()***, to the ***RenderPGranter***. This method takes a URL as a string, confirms that this URL is a valid URL that appears on the current page, adds this string to the history list, creates a ***java.net.URL*** object for this URL (which conveys authority to fetch the document named by this URL), and returns this ***java.net.URL*** object to the renderer. The renderer can then use this capability to fetch and display the corresponding document.

In addition, the ***CapBrowser gotoURL()*** method will fetch the document a second time and store it in textual format for later use by the ***RenderPGranter*** to validate future URL requests. (There is a special case: if the URL names a caplet, then the ***CapBrowser*** will download the caplet source and start it running in the background.)

Note that this interface is exported only to the ***RenderPGranter***, not to the renderer. Also note that the ***RenderPGranter*** is allowed to ask for any URL whatsoever to be loaded, whether or not it occurs on the current page.

5.8 RenderPGranter

The ***RenderPGranter*** is the primary enforcer of the security policy. It provides the renderer the four powers listed above.

We found a significant vulnerability in the ***RenderPGranter***'s hashtable of capabilities. There is a single hashtable for all time, and each time a new renderer is created, we create a new scroll pane for the renderer to draw in, update the hashtable to contain a reference to the new scroll pane, and start up the new renderer with a reference to the ***RenderPGranter***, which will hand out a reference to the current scroll pane upon request by doing a lookup in the hashtable. Now the vulnerability is apparent: the previous renderer retains its reference to the ***RenderPGranter*** and hence can request the scroll pane from the ***RenderPGranter***'s hashtable, thereby receiving a capability to the new scroll pane. Now both the old renderer and the new renderer share a capability to the new scroll pane. After this, all sorts of mischief are possible: for instance, the old renderer can overwrite the scroll plane with the contents of a previous web page, even though the URL field shows the new URL, thereby violating the "synchronization" property. As another example consequence, the old and new renderers can now communicate and collude to violate the security policy, even though they were intended to be isolated from each other.

This was an implementation mistake that is easily repaired. The lesson we learned is twofold: first, functional programming and immutable state carry security benefits; and second, wherever multiple applications are intended to be isolated from each other yet all carry a reference to a single shared entity, we should be very careful about security flaws. (Unfortunately, the latter property is one that is not evident from the reachability graph, and so slightly more sophisticated reasoning is needed. Fortunately, the reasoning required is not too terribly difficult, and becomes considerably easier if functional programming style and transitively immutable values are used.)

We found a second significant vulnerability in the ***`RenderPGranter`***'s enforcement of restrictions on the renderer. The ***`RenderPGranter`*** allows the renderer to request loading of a new URL, and is supposed to ensure that this URL is mentioned in the current web page; only if this check succeeds should the ***`RenderPGranter`*** pass on this request to the DarpaBrowser. The ***`RenderPGranter`*** performed this check by doing a substring match between the requested URL string and the current HTML document. However, this substring match is easily fooled: as a simple example, if the current document contains a link that has been commented out, the substring check will still succeed on this URL, and hence a malicious renderer could change to displaying this commented-out link. Though the renderer does not gain unrestricted access to the entire web, this is a violation of the security policy. The problem here can be viewed as a failure of capability discipline: security checks are being performed on strings rather than on the underlying objects that represent the entities to which access is being requested.

We found a third vulnerability in this same code: in particular, it has a TOCTTOU flaw, i.e., a race condition vulnerability. The code implicitly assumes that it is being passed a String object containing the text of the requested URL, but it does not check this assumption anywhere. Moreover, the code uses the parameter twice, with the assumption that it will have the same value both times: once in the substring check, and then later to actually ask the ***`CapBrowser`*** to fetch this URL. In fact, it appears that these assumptions are invalid, and an attack is possible. A malicious renderer could pass a specially crafted object that replies with a valid-looking URL when it is first queried (during the substring check; note that this substring check is performed using an E quasi-parser, which issues a method call to the underlying object), but replies with a maliciously chosen URL the second time it is queried. This will bypass the security policy that the ***`RenderPGranter`*** is trying to enforce and allow the malicious renderer to take the browser to any URL whatsoever it can think of, whether on the Internet at large, in the local intranet, or perhaps even on the local file system. This TOCTTOU implementation bug illustrates the need for a more disciplined style of programming, such as use of type guards and care taken when handling untrustworthy objects.

We found a fourth vulnerability as well, also a concurrency bug. Note that when a new page is requested, the ***CapBrowser*** first calls the renderer to display the page, then loads the HTML of this page and stores it in a variable shared with the ***RenderPGranter***; the ***RenderPGranter*** uses this variable to validate requests to change pages. The problem is that nothing guarantees that the renderer can't request a new URL before the current page's HTML has been stored in the shared variable. For instance, consider the following sequence of events. The web page for "www.yahoo.com" is currently showing. Suppose the page "www.erights.org" is now requested. The ***CapBrowser*** will call ***renderer.reactToURL("www.erights.org")***. In executing this method call, the renderer calls ***RenderPGranter.gotoURL("movies.yahoo.com"***), making sure to do this before the ***CapBrowser*** gets a chance to download and save the HTML for the erights.org page into the shared variable. This request will succeed, because "movies.yahoo.com" is linked to on the yahoo.com page and because the shared variable has not yet been updated to contain the contents of the erights.org web page. Yet this request should not have succeeded, according to the security policy, because "movies.yahoo.com" is not linked to on the erights.org web page. The impact of this race condition seems likely to be small in practice, but it is a small implementation bug. This bug is fixable.

We understand that the problematic code in the ***RenderPGranter*** will be replaced by a new implementation that follows capability discipline more closely. In particular, the idea is for the ***RenderPGranter*** to parse the HTML document, build a DOM tree, and replace all links by special objects representing the URL and conveying authority to call back the ***RenderPGranter*** and request it to change to that page. This modified parse tree will then be handed directly to the renderer, and the renderer will request display of a new URL by selecting a callback object from its parse tree and invoking it. We believe that this will greatly increase the security of the implementation: it will defend against data-driven attacks (so long as the web server is not in cahoots with the renderer), and it will avoid the attacks described above. This is yet another example where following capability discipline more closely would have protected the DarpaBrowser against various attacks; thus, this experience seems to be at least as much of a vindication of the capability architecture as it is a criticism of the current implementation.

5.9 JEditorPane (HTMLWidget)

We paid special attention to the Java class ***javax.swing.JEditorPane***, which contains a Java HTML parsing and rendering engine. This class was made available to all applications (by marking it "safe" in the SafeJ database) so that one could very easily construct a simple renderer: rather than having to write a HTML parser and renderer afresh, one could simply reuse the existing Java code. However, this strategy also carried some risks, as the ***`JEditorPane`*** is a complex piece of code, not written with capability discipline in mind and potentially replete with ambient authority. For instance, when handed a HTML document, the ***`JEditorPane`*** will automatically load and display inline images mentioned in the document. Since this is done without needing a ***`java.net.URL`*** capability for the image from the caller, it means that the Java code encapsulates extra authority beyond that explicitly handed by its caller, a violation of capability discipline.

We did not identify any security vulnerabilities associated with the ***`JEditorPane`***. However, it does come with some risk: it is difficult to be sure that the ***`JEditorPane`*** is not wielding ambient authority in some subtle way or in some unexplored corner case. We note that Combex's proposed new implementation, based on parsing the HTML in the ***`RenderPGranter`***, seems to significantly reduce these risks, and we expect that the new implementation will have excellent security properties.

5.10 Installer

E contains a novel mechanism for downloading and installing untrusted or mobile code. However, renderers are not installed permanently: they are transient code loaded by the DarpaBrowser. Therefore, the installer seems to have no security implications for the security goals of this exercise.

6. RISKS

This section describes the highest risk areas identified during the review.

6.1 Taming

The approach of taming the Java API has several risks.

The most serious risk comes from size and complexity: because the Java API is large and growing, it is likely that incorrect taming decisions will be made, thereby giving a malicious renderer a power it would not otherwise have had in the absence of Java code. Indeed, specific incorrect taming decisions were identified during the security review, and these could enable malicious renderers to violate the intended security policy.

This risk is partially mitigated *by the use of a semi-objective rule for a taming decision: "Does a particular method call use ambient authority?"* Methods that don't provide the caller with any extra powers beyond that implied by the current object can be safely allowed. In contrast, methods that wield ambient authority should be suppressed. (In effect, we are trying to impose on Java the same restrictions that the E interpreter already enforces on E code.) This guideline provides a framework for thinking about taming decisions, and permits independent review of the taming decision.

It should be noted that the structure imposed by the taming question is crucial. By comparison, a question of the form, “is a function *safe enough* to put into a sandbox?” is hard to answer reliably (which raises the risk of incorrect taming decisions) and highly subjective (and thus less amenable to satisfactory independent review).

The taming process does not appear adequate for individual components with substantial internal complexity or that use substantial authority. The HTMLWidget illustrated this. For such cases, wrappers must be provided that explicitly manage the authority allowed to the caplet. This is not feasible for large libraries. Because of the object-oriented nature of Java, however, most of the Java libraries do not fall into this category.

All the taming risks described above are amplified because the taming interface is provided as a single bucket to applications. The result is that any incorrect decision in all tamed API is exposed to all caplets, emakers, etc. Because the Java APIs cover a broad range of applications, there is no need to allow such exposure, and indeed it seems contrary to the overall POLA approach to the system design. By factoring the Java APIs in separate tamed buckets, an incorrect taming in one part of the API (e.g., user interface components) would not be exposed to all components.

6.2 Renderer fools you

A second risk is that the renderer might try to fool you by displaying material that is not faithful to the intentions of the current web page’s author. Some examples:

1. The renderer could display the current web page poorly. It could show images at degraded resolution, it could format the text strangely, and so on. This is unavoidable.
2. Similarly, the renderer could ignore the current web page and display something else entirely. One way this could happen is the renderer could come with a malicious web page hard-coded: for example, perhaps with an embedded link to your stock broker containing the “buy lots of stock symbol S” action, in an attempt to fool the stock broker into thinking this request was authorized by the user. Another example might be to include a link to “/dev/mouse”; on some Unix systems, clicking on such a link may hang your X Windows session. Alternatively, the malicious renderer might try to fool you by showing a hard-coded web page containing spoofed content: e.g., a forgery of a CNN page with a fake headline.
3. The renderer could show a modified version of the current web page. For example, the malicious renderer could remove all ad banners. Or, the malicious renderer could insert the word “not” randomly into a few well-chosen sentences. As a third example, a malicious renderer could remove all stories that say negative things about Democrats whenever you browse an online news site.

Of course, in each of these examples such a renderer cannot spoof the URL field shown by the ***`CapBrowser`***. Moreover, the renderer is limited by the fact that it has no memory (a fresh copy is started up each time the user views a new web page) and that the only web page it can fetch is the one specified in the URL field shown to the user. The idea is that this will defend against an attack where a malicious renderer, when asked to view one web page, fetches another one and shows you the latter. In the Combex DarpaBrowser, a malicious renderer cannot mount this attack, because the renderer can only show what is hard-coded in it. A consequence of this is that if you download the renderer at time T, then the renderer cannot know about anything that has happened after time T except for what is contained in the web page it has been asked to display. Moreover, after the current web page is discarded, the copy of the renderer that displayed this page will be discarded as well, and so the renderer should be forced to effectively “forget” what it learned from the current web page.

That’s the idea, anyway. In practice, the security level actually attained is complicated by another consideration: covert channels. There are many ways that a malicious renderer could try to subvert this basic approach by using covert channels to gain extra information or to synthesize a form of memory. We give two examples here:

1. The renderer could collude with another attacker across the network to show you a page other than the one currently visited. Note that though the malicious renderer cannot directly fetch any page other than the one currently requested, it can communicate with its external co-conspirator by using any of a number of covert channels, and the external co-conspirator, who is not limited like the renderer, could look up other web pages on the malicious renderer’s behalf and transmit them back.
2. Multiple instantiations of a malicious renderer could collude with each other. Recall that a separate instantiation of the renderer is started for each web page the user visits. However, the old renderer can arrange to continue running in the background by issuing eventual sends. Consequently, we can have multiple copies of the renderer running. Suppose we designate the first copy as the “brain” and all subsequent copies as “slaves”. Note that the “brain” can communicate with the “slaves” using any of a number of covert channels (e.g., modulating the system load). Then each time a new slave is created to render some web page, the slave could send a copy of that web page to the brain, and the brain could direct the slave what to show in an attempt to display something particularly pernicious. Note that this subverts the intention that renderers be memory-less, although it does not really give malicious renderers any other powers.

Covert channels seem very difficult to avoid. For instance, a malicious renderer can communicate with an external colluder by modulating the order or timing in which it fetches inline images (which it must surely be allowed to do). Multiple malicious renderers running on the same machine could communicate by modulating the system load, and could derive a sense of time using the time-stamps returned by web servers or by watching UI events. Covert channels have been studied extensively in the security literature, and billions of dollars have been spent trying to eradicate them, to little effect.

In general, we consider all of the above attacks instances of “rendering poorly”. It is impossible to avoid the possibility that a malicious renderer will render poorly; even some well-intentioned renderers occasionally render web pages poorly, so it is too much to expect that we can prevent malicious renderers from doing the same at times. Rather than being a limitation of the Combex DarpaBrowser architecture, the “rendering poorly” attacks seem to be unavoidable given the problem statement.

6.3 HTML

HTML has evolved into a large and complex set of requirements with no coherent underlying architecture. For the purposes of accomplishing the goals of the project, there is substantial risk that HTML has features that are fundamentally incompatible with the security goals (e.g., features that are inherently susceptible to the confused deputy attack). For example, a Web form delivered from an external site may designate a file inside the firewall as an inline image. As a result, the page is expressing that the renderer should be able to read the file behind the firewall, and should be able to send data outside the firewall. There are more complicated variants of this scenario for which there are no simple solutions that also consistent with the expectations of HTML authors, browser, and users. In this example, features of HTML that evolved independently interact to specify a circumstance in which gross security violations are possible.

6.4 HTML Widget complexity

A fourth risk is that the ***`JEditorPane`*** (***`HTMLWidget`***) uses substantial ambient authorities that might potentially lead to a security breach, if we are unlucky. Due to the complexity of the Java code, we were unable to rule out this possibility in our analysis. On the positive side, this risk definitely seems to be a distant fourth place, compared to the other risks enumerated above.

Moreover, this risk seems to be not so much a limitation of the E capability architecture as it is a fundamental issue with interfacing with legacy code. Safely integrating legacy code that wasn’t written with capability discipline in mind into a capability-based architecture is not straightforward (if anything, it seems to be a good question for further research), but is also outside of the scope of the FRT project goals. Of course, it would be straightforward, but deeply time-consuming, to re-implement a HTML parsing and rendering engine in E, and this would completely avoid the risks associated with the ***`JEditorPane`***, but we see little point in doing so. The current implementation already adequate makes the case for the E architecture. In short, legacy code integration seems to be orthogonal to the goals of the DarpaBrowser exercise, so we do not interpret the risks associated with the ***JEditorPane*** as a shortcoming of the DarpaBrowser architecture.

Finally, we mention that Combex has proposed a new implementation strategy (discussed above) for mitigating this risk by following capability discipline more closely. We believe that this second-generation implementation approach would have excellent security properties, so there is a promising path for eliminating the ***JEditorPane*** risks at fairly low cost.

7. CONCLUSIONS

This report detailed the results of an intensive analysis of the DarpaBrowser architecture and the E capability system. We were asked to assess the architecture, and to look for classes of implementation bugs that might be infeasible to fix within the architecture. We did not find any. (In this report, we noted all implementation bugs we found, but implementation bugs that are easily fixed within the E/DarpaBrowser architecture were considered out of scope.) We did find ways that the design of the system could be easily improved to further increase the security provided by the Combex DarpaBrowser, but the existing architecture already seems to provide a good basis for reaching the objectives set out in the FRT project goals.

If one looks specifically at the current implementation, the implementation we were provided was not adequately secure. It contained several serious implementation bugs that would allow a malicious renderer to violate the security policy. Therefore, the current DarpaBrowser implementation is not yet ready for use in high-risk settings. On the other hand, all of the bugs we found are fixable within the DarpaBrowser architecture, and they do not pose any serious architectural risks that would be likely to affect the security of the DarpaBrowser design.

Of course, the real goal of this review was to evaluate the DarpaBrowser architecture, and this is where we focused our effort. Our evaluation is that the architecture passes nicely. As a way to build sandboxes, the E capability system seems well-designed, and there are some reasons to expect that it may fare better at this than the Java SecurityManager and other state-of-the-art approaches, if given comparable resources. Indeed, if the DarpaBrowser exercise is viewed as a test of our ability to build security boundaries, then we believe the E capability architecture was quite effective at this. In fact, it was so easy to build security boundaries that the DarpaBrowser developers went ahead and built three of them as a matter of course, even though the problem statement only required a single security boundary. This is a testament to the E architecture.

The biggest security risk that stands out is the legacy Java interface. Because this Java code was not designed with a capability architecture in mind, there is a substantial risk that this integration with legacy Java code might create security breaches that allow sandboxed applications to escape their sandbox. However, safely integrating legacy code was not one of the goals of the DarpaBrowser exercise, so though we urge future research on this topic, we consider this issue outside the scope of the exercise goals.

All in all, we believe that, by following the E architecture, it is possible to build a web browser that satisfies most of the security goals of the exercise. There were a few goals that seemed fundamentally impossible to achieve—for instance, avoiding collusion is as hard as stopping covert channels, which there are good reasons to believe is an infeasible task—but this is not a shortcoming of the E architecture.

We wish to emphasize that the web browser exercise was a very difficult problem. It is at or beyond the state of the art in security, and solving it seems to require invention of new technology. If anything, the exercise seems to have been designed to answer the question: Where are the borders of what is achievable? The E capability architecture seems to be a promising way to stretch those borders beyond what was previously achievable, by making it easier to build security boundaries between mutually distrusting software components. In this sense, the experiment seems to be a real success. Many open questions remain, but we feel that the E capability architecture is a promising direction in computer security research and we hope it receives further attention.

Appendix 3: Draft Requirements For Next Generation Taming Tool (CapAnalyzer)

The following items are required for a taming tool powerful enough to support reliable yet cost effective imposition of capability discipline on large non-capability toolkits. These requirements are presented as extensions to the existing CapAnalyzer used during this project:

- Enable revision of existing safej files
- easily start anywhere
- do superclasses before subclasses
- do supertypes before subtypes
- checkboxes are: magic, suppressed, and settable set of buckets for statics for instance stuff it is suppressed or not suppressed
- "it is on when the following eprop is on" checkmark
- as cursor rolls across methods, move focus to current comment field
- show javadoc for each method
- remember, it's not just method, it is public variables, and constructors
- fix the inner classes $ problem
- go at random to any class
- go to the superclasses from this class
- go to return types and arg types
- log classes for seeing which classes, which packages, reviewed, by whom, when, how many times
- no interleaving instance things and static things
- make comment field append-only; when focus is moved, it is moved to end of comment: previous comments are a label with the javadoc
- toggle to showing only methods etc. in a subset of buckets so you can see safe and swing but don't bother with unsafe, for example.

- toggle instance methods to show only safe, only unsafe, or both.
- in tool, public final scalars both static and non-static are turned on by default.
- create output which can be diff'd to existing version using simple text comparators.
- automatically detect if a subclass overrides a method and the suppression in the subclass and superclass are inconsistent.
- Include new choice for disallowed methods: "override" rather than "suppress". Override will prevent the finding of the method even if the method is part of the superclass rather than being just part of the current class. The method that does the override will be automatically generated, and will throw the "no such method" exception. A warning should automatically be presented if "suppress" is selected for a class for which the method is also implemented in superclasses.

Appendix 4: Installation Procedure for building an E Language Machine

INTRODUCTION

This document describes how to build an E Language Machine (ELM). ELM is the world's first capability secure computing platform with a graphical user interface. It is invulnerable to traditional computer viruses and trojan horses, yet is as easy to use as a conventional desktop. However, since ELM is currently only a rudimentary prototype, it is not as fully featured as a modern desktop.

The basic parts of an ELM are:

- Linux core operating system
- Java Virtual machine
- E Language Interpreter
- CapDesk point and click capability secure distributed file manager

The basic idea of an ELM is that the Linux core OS launches a Java Virtual Machine, which launches an E Interpreter, which launches a CapDesk. The CapDesk file manager then seals off the underlying components of this Trusted Computing Base (TCB) in such a fashion that neither Linux nor Java nor E can directly launch any additional applications: the only applications that can be launched from a CapDesk are capability confined applications, or ***caplets***. Two example caplets are included with this distribution of ELM: a simple though effective text editor (CapEdit), and an experimental non-production-quality Web browser (DarpaBrowser), and a rudimentary Web server.

INSTALLATION OVERVIEW

The approach taken with this installation, to keep the process as simple as possible, is as follows:

- Perform a standard Linux installation, including a KDE desktop. The KDE desktop is used for bootstrapping: KDE supplies convenient, point-and-click user friendly tools for completing this installation. Once basic installation is complete, the KDE desktop is toggled off, and CapDesk becomes the default desktop manager. GNOME can be used rather than KDE if that is preferred; however, these installation instructions presume KDE. Similarly, any modern distribution of Linux can be used, but this installation guide specifically assumes RedHat 7.3. If you have received a full ELM build package, rather than just this page of directions, you will find Red Hat 7.3 disks included, along with a single ELM disc. If you have only this page of directions, all elements of this build can be downloaded from the web, from Red Hat, JavaSoft, and Erights.org.

- Install WindowMaker, which will be the window manager for the ELM CapDesk.
- Install the JavaSoft Jave Runtime Environment (JRE) version 1.4. Serious problems with garbage collection have been found when using either JavaSoft JRE 1.3 or the IBM JRE 1.3 (at this time there is no IBM JRE 1.4). These problems still exist with the 1.4 version, but they have been mitigated, and do not degrade performance so rapidly that ELM cannot operate for a reasonable period of time.
- Install the E Programming Language, version 0.8.18 or later.
- Install CapDesk.
- Scan the Linux system for open ports and network services. Kill all network connections that are not driven by CapDesk or one of its caplets. These E-based connections are capability secure and present no cyberattack risks.
- Configure and toggle the system so that, at boot time, WindowMaker with CapDesk is launched rather than KDE
- Reboot, and finish configuring WindowMaker
- Install CapEdit, DarpaBrowser, and CapWebServer
- You now have your own installation of the world's first point and click desktop which is invulnerable to traditional computer viruses and trojan horses.

Step One: Install Linux

By and large, an ELM Linux installation is a standard installation. Special instructions are included for Grub and root passwords, firewalls, additional users beyond root, and desktop/package selection. These instructions assume that the ELM will be a single-user system; for a shared system, contact us at the email address at the bottom of this manual for additional assistance.

1. Configure your computer to boot from cd-rom.
2. Boot from the Red Hat 7.3 disc
3. Select either the Workstation or Laptop setup. Do not select the Server setup: the server option will automatically turn on large numbers of network services that will have to be shut off again later in the ELM construction process.

4. Configure the keyboard, ethernet, etc. as appropriate. Choose to log in graphically if you want to use the X windows security fix listed in these instructions.

5. The Grub password, like the root password, is barely useful for an ELM, and only for giving weak security against direct physical control by the adversary. Frankly, against physical threats these passwords are of limited value: the serious cracker will simply boot from cd-rom or floppy, bypassing these defenses. In general, therefore, we recommend skipping the Grub password. For the same reason, we recommend assigning a simple root password. And go ahead and put that root password on a sticky note on the monitor.

6. Firewall: on the firewall configuration screen, select No Firewall. Firewalls are redundant on an ELM machine, except to the extent to which they interfere with legitimate computing activities.

7. Add User: an ELM does not need or use access control lists, and distinctions between "normal" users and "root", for security. Indeed, on a capability secure desktop, this differentiation of access becomes a pure liability: even though the access controls are superfluous, they can still get irritatingly in the way while trying to get your work done ("oops, I copied this file while under that other user name, and I don't have the authority now").

Having completed this explanation of why non-root users are a bad idea on a single-user ELM machine, we must confess that, with this rudimentary prototype, there is one advantage to having a separate user account: it provides protection against accidentally damaging system files. In a production version ELM, the default file manager windows would be capability confined to operate only in the user areas, and a special action would be needed to bring up a distinctly marked window that browsed and edited system files. Regardless, these instructions assume a user sophisticated enough not to shoot his own foot when given a reasonably friendly graphical user interface, and all instructions here assume only the root account exists.

8. On the Desktop/Package Selection screen, deselect GNOME. Select KDE. Also select "Select Individual Packages", which is at the far bottom of the screen and is easy to miss.

9. On the Individual Package Selection screen, navigate to User Interface/Desktops and select WindowMaker.

10. On the following screen, choose to Install Packages to Satisfy Dependencies for WindowMaker.

11. Complete the installation, reboot, and log in

Step Two: Install WindowMaker, Java, and E

1. Bring up a terminal (you can click on the Konsole icon on the toolbar).
2. Run "wmaker". This will install WindowMaker. Disregard the error messages; these will disappear when KDE is shut off
3. Put in the ELM cd-rom. Mount the cd-rom. Copy the jre file (with the suffix ".bin") to the KDE desktop.
4. Execute the jre bin file from a terminal window. It will play the JavaSoft terms and conditions. Type "yes" at the end to accept the conditions.
5. Accepting the JavaSoft agreement extracts an rpm file from the bin file. Click on the rpm file; this will bring up kpackage.
6. In the kpackage dialog, select the j2re package and click Install. This will unpack the JRE into /usr/java
7. In the /usr/bin directory, place a link to /usr/java/j2re1.4.0_01/bin/java. The exact path will vary depending upon which release of java you are using.
8. Type "java" into a terminal. If your installation has succeeded, you will get the help page for the java virtual machine.
9. Create a folder under /root named "elang".
10. Click on the E tar.gz file on the cd-rom. This will bring up Ark.
11. Extract the E tar.gz file into /root/elang.
12. In /root/elang, copy the eprops-template.txt file to eprops.txt. Edit eprops.text.

    Change the "e.home=" line to

    e.home=/root/elang/

    Change the TraceLog_dir line to

    TraceLog_dir=/root/etrace/
13. Create the folder /root/etrace/
14. Test CapDesk: in a terminal, type "java -jar /root/elang/e.jar /root/elang/scripts/CapDesk.e" In general, you will receive a flurry of warning and error messages which are irrelevant. If you start seeing a series of printouts in the terminal such as "start" and "compiled maker maker", the launch of CapDesk is proceeding successfully. Depending on the performance characteristics of your computer, a window labeled "My CapDesk" will arrive on your desktop in due course. Navigate by clicking and double-clicking on folders, and the Up Directory button.

15. Close the CapDesk window, shutting it down. If you leave CapDesk running during the upcoming security check, it will be much more difficult to ascertain which network services are security risks, since CapDesk itself creates a number of secure network connections.

Step Three: Secure Machine Against Network Services

Even for laptop and workstation installations, RedHat by default turns on several network services. And any variation in the Linux version being used could cause other services to run as well. They all need to be shut off.

1. Make sure CapDesk is shut down.
2. In a terminal, run "lsof -i". This will give you a list of all the ports that are open to attack. Your list may be different if you have varied even slightly from the directions and version numbers herein. But the following items typically need to be removed or modified:
3. Remove or rename /sbin/portmap
4. Remove or rename /usr/sbin/sendmail
5. Somehow configure the X server startup process with the "-nolisten tcp" option. If you are using console login, this argument can be passed through the startx command ("startx -- -nolisten tcp"). If you are using XDM for graphical login, edit the /etc/X11/xdm/Xservers file. Append the nolisten option to the startup command for the :0 X server: ":0 local /usr/X11R6/bin/X -nolisten tcp".
6. Run lsof -i again to confirm that there are no active ports. If possible, run an nmap scan of this machine from another machine, as a double check.

Step Four: Configure WindowMaker/CapDesk Startup

1. At the bottom of the file /root/GNUstep/Library/WindowMaker/autostart, add the lines

   ```
   cd /root/elang
   java -jar e.jar scripts/CapDesk.e &
   ```

   This will cause CapDesk to launch automatically during login to ELM.

2. Remove all the items that appear by default on the WindowMaker popup window that are inappropriate for a capability secure desktop. This is almost the entire list of standard options. We recommend replacing the file /root/GNUstep/Defaults/WMRootMenu with this much shorter version (though you may want to go into Windowmaker and set your theme and style for the desktop before going all the way to this drastic extreme: you can replace this file using CapEdit after you have otherwise completed ELM if you prefer a lighter, brighter desktop than the default WindowMaker)::

   ```
             ("Applications",
   ```

```
("Exit",
("Restart", RESTART),
("Exit", EXIT)
)
)
```

3. Use the KDE Desktop Switcher, or run "switchdesk" from a terminal, to choose WindowMaker as the default window manager.

4. Restart your system. If all goes well, when you log in , you should come up under WindowMaker, and CapDesk should launch automatically.

5. Double-click on the WindowMaker Preferences icon at the top right corner of the screen. Go to "Miscellaneous Ergonomic Preferences". Choose to make the Size Display show at the corner of the screen. Choose to make thePosition Display show at the corner of the screen. Windows drawn by Java seem to have trouble overrunning these realtime popups when they are in the center of the window being adjusted.

6. Right-click on the terminal icon in the top right corner and pick "Settings" off the popup menu. Delete "xterm" from the application path.

7. At this point you have constructed a full ELM workstation. It is possible for users of even modest sophistication to break through the veil of capability security in which Linux and Java have been wrapped, but it is not possible to do so by accident. Since a person with physical access to the system cannot be stopped from running any non-capability-application he wants to run, if that is his dearest intention, this seems like a sensible tradeoff of usability versus security for this rudimentary prototype of a capability secure desktop.

Step Five: Confined Application Installation and Normal Operations

Navigation with CapDesk is typical of point and click file managers. Double-click on a folder to open that folder's contents in the current panel; single click the folder to see its contents in the next panel to the right. Press the Up-folder button on the toolbar to navigate up through the directory tree. Type a folder path in the field at top and press Enter to jump directly to a location. Right-click on a folder and on a file to see the options that appear in the popup menu.

To install CapEdit:

1. Browse in a CapDesk window to /root/elang/caplets/capEdit/

2. Right-click on the file capEdit.caplet. Choose "Install" from the popup menu.

3. Choose a name, an icon, and a default document suffix for CapEdit. You can simply click "Finish Installation", accepting the defaults.

4. There are several ways to bring up CapEdit and edit a file. Double-click on a file with the CapEdit suffix (".txt" by default). Or right-click on any file, choose "Open With" from the popup menu, and select CapEdit. Or, to launch CapEdit with a file, right-click on capEdit.caplet and choose "Run".
5. Once CapEdit is up and running, you can open additional files either with the Open File button on the CapEdit toolbar, or by dragging and dropping files from a CapDesk file manager window.
6. Note: the cut/copy/paste buttons on the bottom of the window (the powerbar) are non-operational. However, control-x/c/v do cut/copy/paste.

To install CapWebServer:

1. Navigate to the capWebServer folder in /root/elang/caplets/capWebServer
2. Right-click on the file capWebServer.caplet. Choose "Install" from the popup.
3. This installation dialog has two tabs. The first is identical to the CapEdit tab. The second tab offers Server authorities. Choose to grant a server port (port 80 is the standard web server port, change it if you like, but you will need to specify any nonstandard port in the URLs for the web browsers thereafter). Choose also to allow the web server to run "independently". It will still be capability confined, but it will run on a separate java vm, which will make CapDesk itself more responsive.
4. To run the web server, right-click on a folder that is configured as the root of a set of page document folders. There is an example root doc folder in the CapWebServer folder. Right-click on this, and Open With...CapWebServer.
5. A dialog box should come up after the separate jvm has launched the server is operational. To terminate the server, click the Terminate button or close the window. In general, you will want to minimize this dialog box. The web site being served can be reached by simply going to http://localhost/ using the DarpaBrowser. Any other web browser can connect and use it as well; keep in mind, however, that this is a rudimentary prototype.

To install DarpaBrowser:

1. Navigate to the DarpaBrowser folder by the CapEdit folder.
2. Right-click on either darpaBrowser.caplet (for the simple demonstration version) or darpaBrowserMemless.caplet (for the testbed version) . Choose to install. Note: you can install both if you prefer, just make sure that they have different pet names (which will happen automatically if you just choose the default pet names).

3. Note that this installation dialog has two tabs. The first tab is identical to the CapEdit tab. The second tab offers web protocol authority. Choose http protocol. Note: file protocol does not work at this time.

4. Right-click on darpaBrowser.caplet and choose Run to launch the browser. If you are connected to the web, type a URL, such as "http://www.combex.com" into the Goto field and hit Enter to begin browsing.

5. The "copy" button on the powerbar is operational, and can be used to copy text to be repasted into CapEdit documents.

6. The DarpaBrowser is not currently able to read local html files; it must access its pages via http (though if you have a web server running on localhost, that can be accessed).

7. You can explore alternate renderers by selecting the Choose Renderer button. By default, the DarpaBrowser starts with its benign renderer; DarpaBrowserMemless starts with capTreeMemless. The textRenderer only works with the demo DarpaBrowser. The benignMemless renderer only works with the testbed DarpaBrowserMemless. The evil renderer, which is by far the most interesting renderer, works with both browsers. This renderer will attack your system in an attempt to take control. It will report on its results as it attempts various breaches. When run unconfined on a Windows or bare Linux system, these attacks are successful. However, here on the CapDesk, they all fail. One of the pages in the sampleRootDoc for the CapWebServer shows the results of these attacks if the malicious renderer is run with standard Windows/Linux authorities.

8. Each of the renderers has strengths and weaknesses. None of them are production quality; they were all designed for research, not daily operations.

   The two benign renderers paint pages quite well, but are extremely fickle about the HTML they accept and consequently many, many pages on the Web cannot be rendered (including the Google home page, for example). A particular problem for the benign renderer is the meta tag <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">. This tag, which is becoming ubiquitous, is misinterpreted by the Java JEditorPane widget's parser to have a "/html" tag embedded in it, with catastophic consequences. The home pages for Combex,

   http://www.combex.com, and for the E platform, http://www.erights.org, have been carefully edited to ensure that they will work with this renderer. So have all the pages in the sampleRootDoc for the CapWebServer.

   The textRenderer will successfully render any html page no matter how badly formed the HTML; however, it is an uninteresting presentation, simply being the source text of the page. CapTreeMemless will render most Web pages, but the output is far from attractive, and long pages (such as the E in a Walnut page) will fail.

To shut down ELM: Right click anywhere on the WindowMaker background. Choose Exit/Exit off the popup menu to shut down.

Congratulations! You have an operational CapDesk system.

Step Six: Maintenance

The CapDesk capability secure desktop is just a rudimentary prototype at this time. While much of the maintenance of a Linux machine can be done using the CapEdit text editor to modify config and startup files, it may occasionally be convenient or necessary to use the tools available from the KDE desktop. When logging in, select kde or failsafe from the Session Type menu on the login dialog. Perform maintenance as required. Upon rebooting, select Session Type default to return to the CapDesk configuration.

IN CASE OF EXTREME DIFFICULTY

Contact Marc Stiegler at marcs@combex.com for further assistance

.

Appendix 5: Powerbox Pattern

The powerbox pattern will be incorporated into the next release of E in a Walnut, which is at this time the definitive work for practical programming in the E language. This is a draft for the powerbox section of the book.

The powerbox pattern collects diverse elements of authority management into a single object. That object, the powerbox, then becomes the arbiter of authority transfers across a complex trust boundary. One of the powerbox's most distinctive features is that it may be used for negotiations of authority. The less trusted subsystem may, during execution, request new authorities, and the powerbox operator may, in response to the request, depending on other context that it alone may have, decide to confer that authority.

The powerbox is particularly useful in situations where the object in the less trusted realm does not always get the same authorities, and when those authorities may change during operation. If the authority grant is always the same and does not change during operations, a simple emaker-style authorization step suffices, and a powerbox is not necessary. If the situation is more complex, however, collecting all the authority management into a single place can make it much easier to review and maintain the extremely security-sensitive authority management code.

Key aspects of the powerbox pattern include:

- A powerbox uses strict guards on all arguments received from the less trusted realm. In the absence of guards, even an integer argument received from untrusted code can play tricks: the first time the integer-like object (that is not really an integer) is asked to "add", it returns the value "123"; but the second time, it returns the value "456", with unpredictable (and therefore insecure) results. An ":integer" guard on the parameter will prevent such a fake integer from crossing into your realm.
- A powerbox enables revocation of all the authorities that have been granted. When you are done using a less trusted subsystem, the authorities granted to it must be revoked. This is true even if the subsystem is executing in your own vat and you nominally have the power to disconnect all references to the subsystem and leave the subsystem for the garbage collector. Even after being severed in this fashion, the subsystem will still exist for an unbound amount of time until the garbage collector reaches it. If the authorities it has received have not been revoked, it can surprise you with its continued operations, and continued use of authority.

  Not all kinds of objects in the Java API can be made directly revokable at this time, because an E revokable forwarder cannot be used in all the places where an actual authority-carrying Java object is required. For example, if the untrusted object may want to create a new image from an url. The natural way of doing this would be to call the Java Icon class constructor with (url) as the argument. But if an E forwarder were handed to this class constructor, the constructor would throw a type exception.

There are different solutions in different situations. In this example, it may be acceptable, to require that the untrusted object read the bits of the image from the url (through a revokable forwarder) and then convert the bits into an icon.

- A new powerbox is created for each subsystem that needs authorities from within your trust realm. If a powerbox is shared across initiations of multiple subsystems, the powerbox may become the channel by which subsystems can illicitly communicate, or the channel by which an obsolete untrusted subsystem can interfere with a new one. When the old untrusted subsystem is discarded, its powers must all be revoked, which necessarily implies that a new subsystem will need a new powerbox.

In the following example, the less trusted object may be granted a Timer, a FileMaker that makes new files, and a url. The object may request a different url in the course of operations, in which case the powerbox will ask the user for authorization on the objects's behalf; the old url is revoked, and the new one substituted, so that the object never has authority for more than one url at a time. The object that operates the powerbox may revoke all authorities at any time, or it may choose to revoke the Timer alone. Finally, the operator of this powerbox may, for reasons external to the powerbox's knowledge, decide to grant an additional authority during operations, an authority whose nature is not known to the powerbox.

```
# The authorized url may change during operations, so it is a var
def makePowerboxController(optTimer,
            optMakeFile,
            var optUrl,
            optMakeUrl,
            makeDialogVow) {

 # In the revokers map, the object being revoked is the key, the revoker
 # is the value. Note it is the actual object, not the forwarder to the
 # object, that is the key.
 var revokers := [].asMap()

 # when a revokable forwarder is made, the revoker is automatically
 # placed in the map of revokers
 def makeRevokableForwarder(object) :near {
  var innerObject := object
  def revoker {
   to revoke() {innerObject := null}
  }
  revokers with= (object, revoker)
  def forwarder extends(innerObject){}
 }

 # makeFileFacet is exposed to less trusted realm; guard the path it receives
 # This simple file facet only supplies two methods that return immutables
 # If the file handed out mutable objects, such as streams, these would have
 # to be wrapped with revokableForwarders as well.
 def makeFileFacet(path :String) :near {
   def theFile := optMakeFile(path)
   def fileFacet {
     to getText() :String   {theFile.getText()}
     to setText(text :String) {theFile.setText(text)}
   }
   makeRevokableForwarder(fileFacet)
 }

 # makeFile is actually handed to the less trusted object
 # It is either null or a revokable forwarder for makeFileFacet
 # In other words, makeFile is a revokable maker of revokable file facets
 def makeFile
 if (optMakeFile == null) {
   bind makeFile := null
 } else {
   bind makeFile := makeRevokableForwarder(makeFileFacet)
 }

 def makeRevokableUrlFacet(optUrl) :near {
   if (optUrl == null) {
     null
   } else {
     def urlFacet {
       to getBytes() :pbc {optUrl.getBytes()}
     }
     makeRevokableForwarder(urlFacet)
   }
 }

 # Return a vow for a new url
 # Use dialog with user to determine if new url should be granted
 # Vow resolves to null if anything goes wrong
 def makeRevokableUrlVow := {
   def makeUrlVow(requestedUrl :String, why :String) :vow {
     def urlDialogVow :=
      makeDialogVow("Url Request",
           `<html>
Confined Object requesting url for this reason:<p>$why
</html>`,
           requestedUrl,
           ["Grant", "Refuse"])
     when (urlDialogVow) -> done(dialog) {
       if (dialog.getButton() == "Grant") {
         optUrl := optMakeUrl(dialog.getText())
         makeRevokableUrlFacet(optUrl)
       } else {null}
     } catch prob {null}
   }
```

```
    makeRevokableForwarder(makeUrlVow)
  }


  var caps := [].asMap()
  caps with= ("TIMER", makeRevokableForwarder(timer))
  caps with= ("FILEMAKER", makeFile)
  caps with= ("URL", makeRevokableUrlFacet(optUrl))

  def powerbox {

    # any of these capabilities may be null, i.e., all are optional
    # in a powerbox, strictly at the whim of the powerbox operator
    # who created the powerboxcontroller and the powerbox
    to optCap(key :String) :any {caps.get(key, null)}

    # When the less trusted object requests a new url, the
    # old url is immediately revoked, and the promise for the
    # new url is substituted
    # If the powerbox has revokedAll, any attempt to requestUrl
    # will throw an exception back to the untrusted object
    # The "why" parameter is the less trusted object's justification
    # for needing the url
    to requestUrl(requestedUrl :String, why :String):vow {
      if (optUrl != null) {revokers[optUrl].revoke()}
      revokableUrlVow := makeRevokableUrlVow(requestedUrl, why)
      caps with= ("URL", revokableUrlVow)
      revokableUrlVow
    }
  }

  def powerboxController {
    to revokeAll() {
      for each in revokers {each.revoke()}
    }
    to revokeTimer() {revokers[timer].revoke()}
    # Confer an additional capability during execution
    to conferCap(key, cap) {
      caps with= (key, makeRevokableForwarder(cap))
    }
    to getPowerbox() :any {powerbox}
  }
}

# now, show how to create a powerbox and hand it to an untrusted object

def makeUntrustedObject(powerbox) :any {
  def timer := powerbox.optCap("TIMER")
  def urlVow := powerbox.requestUrl("http://www.skyhunter.com")
  def untrustedObject {
    #...
  }
}

def powerboxOperator() {
  def makeDialogVow {#...construct dialog vow
        }
  # use real objects, not nulls, in typical operation, though nulls are valid arguments
  def controller := makePowerboxController(null,
                    null,
                    null,
                    null,
                    makeDialogVow)
  def powerbox := controller.getPowerbox()
  def untrusted := makeUntrustedObject(powerbox)
  # later, confer an additional authority
  def special
  controller.conferCap("SPECIAL", special)
  # when done with the untrusted object, revoke its powers
  controller.revokeAll()
}
```

Appendix 6: Granma's Rules of POLA

How difficult is it to operate a capability secure desktop? As demonstrated by CapDesk, all the ordinary techniques, from file dialog boxes to drag/drop metaphors, work pretty much the same as under Windows/KDE/Mac. There are no passwords or certificates or funny little security dialog boxes. So a capability secure desktop is no more difficult to operate than any other desktop.

Ah...but how difficult is it to operate a capability secure desktop ***securely***? Can real people really follow the rules that will protect them from Melissa viruses and Sub7Server Trojan horses? Because even with capabilities, despite the absence of passwords and certificates, we are still depending on normal human beings to make authority-granting decisions. This appears most clearly during the installation of a new application, at which time the application can be endowed with default authority (so a Web browser, for example, would sensibly ask for, and often sensible receive, http-protocol read/write authority).

Anyone who has ever installed a firewall or set up a Unix access control list can be forgiven for being skeptical that authority-granting can be made easy enough for the normal person. Yet capabilities implementing the Principle of Least Authority represent a paradigm shift from IPCHAINS as great as the shift that took us from the TECO line editor to MacWrite. No real human being could use TECO; five year old children can use MacWrite.

Herewith, then, are Granma's Rules Of POLA. There are six of them. They are simple and easy to follow for anyone who has seen a CapDesk in operation. They do not fulfill every subtle desire one might have for secure operations in the deepest heart of the NSA. And perhaps the list is not complete--the list has not had the important experience of being attacked by a thousand crackers yet.. But these rules fulfill Granma's needs, allow her to do everything she wants to do without any annoyance from her security system, and the rules have at least passed the scrutiny of the E-lang capability security community reasonably unscathed. You can read the **entire E-lang thread about these rules** in the pipermail archive. To give the context in which these rules were devised, which explains the threats against which these rules are successful, part of the original email about these rules is excerpted below.

Will every person on earth follow every one of these rules perfectly every day? Of course not. But there is a good chance that you, personally, can follow these rules very consistently. So at least you personally will be safe. And even if someone, somewhere, lets a virus activate on their system, the chances that one of the people to whom that future Love Bug mails itself will allow themselves to become infected as well is small. Epidemiologically, rates of transmission would fall low enough that new viruses get no traction and cannot spread. At least, that is our belief. Take a look at the rules, and judge for yourself.

GRANMA'S RULES

1. If an application, during installation, proposes for itself a name or an icon that looks a lot like the name or the icon of something else, give it a new name and new icon.
2. If an application asks for a bunch of different authorities, just say no.
3. If an application asks for read or edit authority on anything outside the Desktop folder, just say no.
4. If an application asks for edit authority on a bunch of stuff, just say no.
5. If an application asks for wide-ranging access to the Web, rather than access to one or two specific sites, only say yes if you plan to use the application as a Web browser.
6. If an application asks for read authority on a bunch of stuff, and also asks for a connection to the Web, just say no.

Notes: Rule 3 is only required by a CapDesk-style implementation of a capability secure desktop. A totally full-powered design, based on a capability secure OS in addition to a capability secure language, would not need this rule. Rule 4 was added in the course of the E-lang discussion reviewing the original list; the rest of the list is essentially the same as the original. Rule 6 is actually not needed to meet Granma's security goals (since she has no confidential data she is worried about having stolen, as described below), but is an easy enough rule to follow, we included it anyway.

EXCERPT FROM ORIGINAL EMAIL, GRANMA'S RULES OF POLA

As I have toured the countryside in the last month giving presentation about the capability secure desktop (see the screenshots under the Technology section at **http://www.combex.com,** they are pretty cool), there's a particular point I keep making that needs to be backfilled with real data. The particular point is, "The typical home user only has to follow six or seven simple rules to be safe from viruses and trojan horses." It would really be good to know what those rules are.

Herewith, then, is a proposed list of Granma's Rules of POLA. The rules are at the bottom of this email; first we must set the stage by describing the threat model which Granma faces, and her interests in the face of the threat.

Granma does not have security needs like a guy working in a compartment at the NSA. She really doesn't have any terribly confidential information: if someone breaks in and steals all the email she has exchanged with her adorable thirteen-year-old grandson Bobbie, it will not make for good blackmail material, and doesn't enable insider trading. Everyone who knows her thinks she is a cool octogenarian; no one is explicitly targeting her, unique in all the world, for a customized attack.

All Granma wants to do with her computer is browse the web for new cookie recipes, send email to her grandson, create and print clever Valentine's Day cards of her own devising, and play Nancy Drew Virtual Reality Team with her granddaughter. This requires that she be able to download and try card-creation applications (drawing packages, word processors, etc.) and the same for mystery games. She needs to not fear opening attachments sent with her grandson's name on it...she has heard, though she doesn't exactly understand how, people can send her email with Bobbie's name on it but with malicious contents.

She is terrified of having her computer taken over by some thirteen year old who is not as adorable as Bobbie, and having her computer used for nefarious purposes (she doesn't know that the FBI might come knockin' on her door some day if someone used her computer in a DDOS attack, but if she did, she would recognize that that is a reason why bad kids shouldn't be allowed to control her machine).

Granma is also terrified of someone breaking in and stealing her money. She uses the computer to tell the Social Security Admin where to deposit her checks, and she has been thinking about getting a digital cash account using Hansa Dollars or e-gold rather than those blasted credit cards, but she won't put real money on her computer until she thinks it would be safer to have money on her computer than it would be to throw the money into the intersection of 5th and Vine.

Bobbie, her adorable grandson, not only loves Granma's choco-chip cookies, but is also wanted in fifteen states by the FBI for computer cracking. He knows just how dangerous it is out there, and wants to make sure Granma can't be attacked by some creep with no more ethics or scruples than...uh...himself. When he sees CapDesk as an alternative, he immediately loads up his own computer and Granma's computer so that they can both be safe.

Here are the rules he gives Granma as he completes installation, and shows her how to drive around:

- If an application, during installation, proposes for itself a name or an icon that looks a lot like the name or the icon of something else Granma already has, give it a new name and a new icon. Don't be shy Granma, it's your computer and your application!
- If an application asks for a bunch of different authorities, just say no. No legitimate application needs many authorities. (well, except for things like development environments, which Granma doesn't need to worry about).

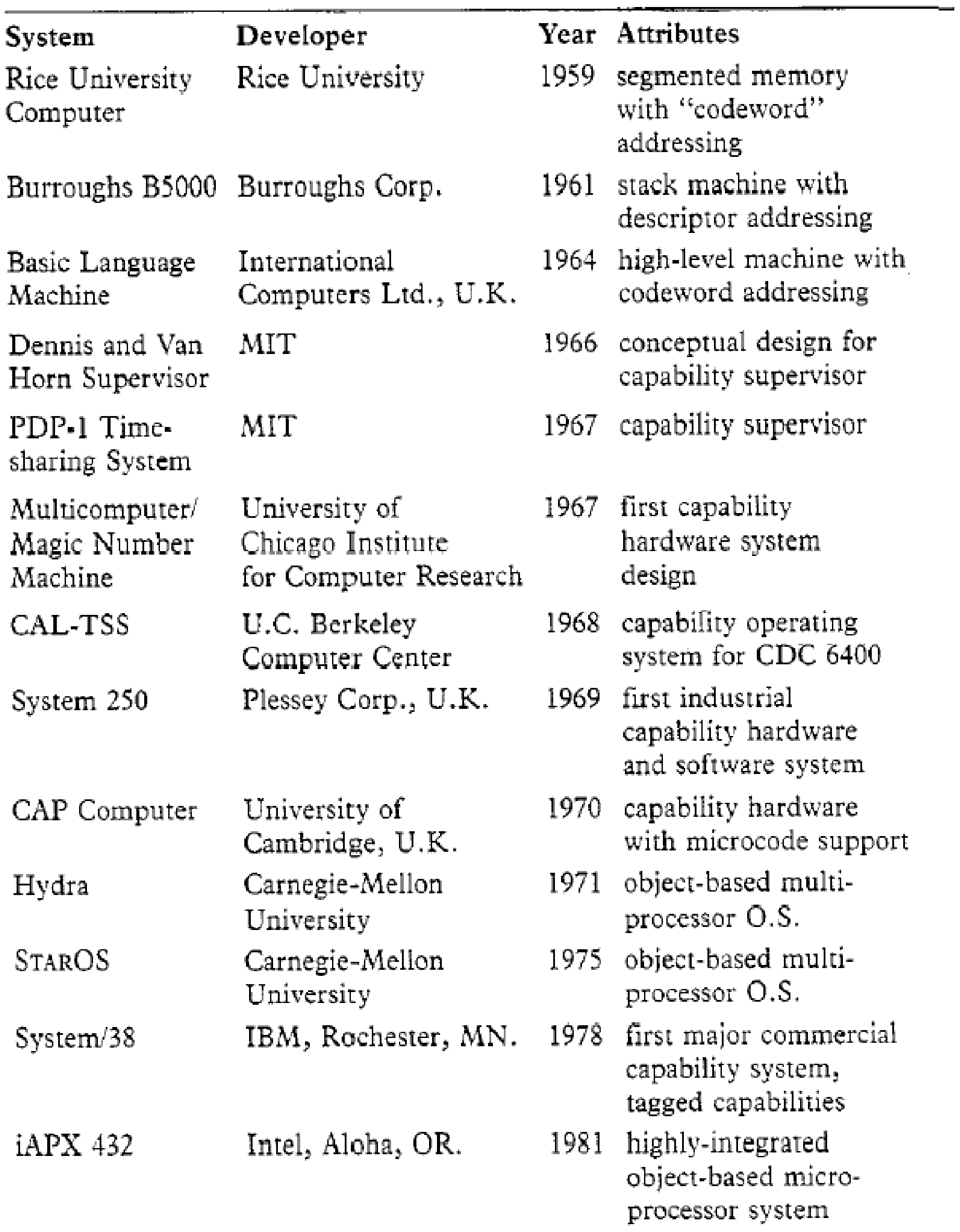
- If an application asks for read or edit authority outside the Desktop folder, just say no. (the current draft layout of stuff in a CapDesk world is, ~/Desktop/MyDocuments contains docs, ~/Desktop has stuff you're currently working on, ~/caplets contains applications, an ~/capData contains info for and about those applications. Proposals for rearranging folders are welcome). Granma, you shouldn't go mucking around outside ~/Desktop either :-) (a real installer, unlike the current CapDesk installer, would copy the caplet executable into the ~/caplet directory for the user as part of the installation).
- If an application asks for read authority on a bunch of stuff, and also asks for a connection to the Web, just say no. (granma doesn't quite need this one, but it is a good rule anyway).
- If an application asks for wide-ranging access to the Web, like a of the http protocol, only say yes if she plans to use it as a Web browser, or if Bobbie says it is ok. A Nancy Drew shared reality team game should only need an Internet connection to one place at a time.

So, there is my draft list, with comments in parentheses that are extraneous to Granma. There's only five of them, room for a forty percent increase before there are more rules than I've been telling people :-)

What clever and terrible attacks can folks think of that will beat these simple rules? In what ways is my specification of Granma's needs and interests incorrect, that requires more flexibility than allowed by these rules?

Appendix 7: History of Capabilities Since Levy

The history of capabilities was first chronicled by Henry Levy in 1984.



System	Developer	Year	Attributes
Rice University Computer	Rice University	1959	segmented memory with "codeword" addressing
Burroughs B5000	Burroughs Corp.	1961	stack machine with descriptor addressing
Basic Language Machine	International Computers Ltd., U.K.	1964	high-level machine with codeword addressing
Dennis and Van Horn Supervisor	MIT	1966	conceptual design for capability supervisor
PDP-1 Time-sharing System	MIT	1967	capability supervisor
Multicomputer/ Magic Number Machine	University of Chicago Institute for Computer Research	1967	first capability hardware system design
CAL-TSS	U.C. Berkeley Computer Center	1968	capability operating system for CDC 6400
System 250	Plessey Corp., U.K.	1969	first industrial capability hardware and software system
CAP Computer	University of Cambridge, U.K.	1970	capability hardware with microcode support
Hydra	Carnegie-Mellon University	1971	object-based multi-processor O.S.
StarOS	Carnegie-Mellon University	1975	object-based multi-processor O.S.
System/38	IBM, Rochester, MN.	1978	first major commercial capability system, tagged capabilities
iAPX 432	Intel, Aloha, OR.	1981	highly-integrated object-based micro-processor system

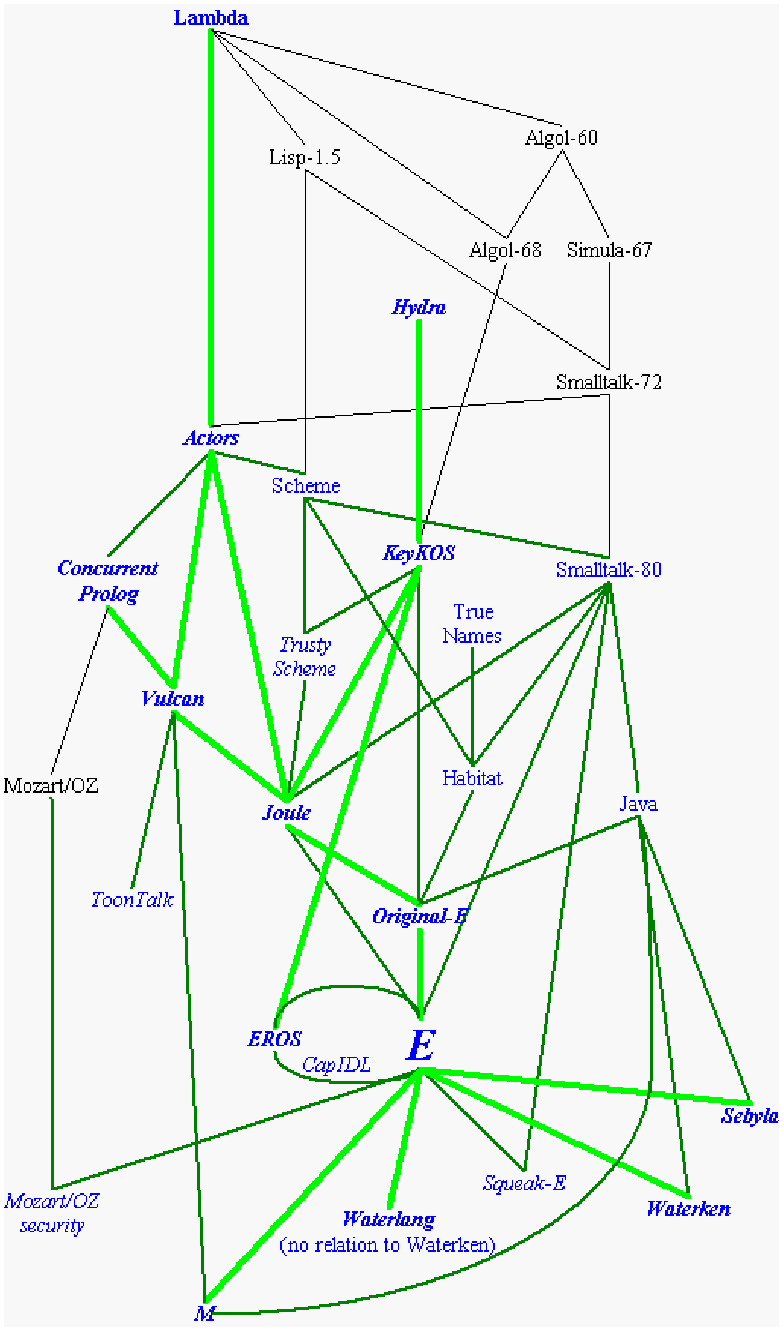
History of Major Descriptor and Capability Systems according to Henry Levy's famous 1984 survey book "Capability-Based Computer Systems" [Levy84].

Not surprisingly, work on capability systems continued since the publication of this book. The major capability milestones absent from Levy's table are

System	Developer	Year	Attributes
Actors	MIT AI Lab [Hewitt73]	1973	1st capability language lambda calculus as distributed capabilities distributed capability protocol sketch
LINCS	Lawrence Livermore [Donnelley81]	1981	Full distributed capability protocol
Concurrent Prolog	Weizmann Institute [Shapiro83]	1983	Horn-clause inference as distributed capabilities
KeyKOS	Key Logic [Hardy84]	1984	Orthogonal persistence Modern capability patterns including confinement
Amoeba	Vrije University [Tanenbaum86]	1986	Distributed password-capability OS
Distributed Mach	CMU [Sansom86]	1986	Transparently distributed capability OS
Vulcan	Xerox PARC [Kahn86]	1986	1st unification of Actors and Logic styles of distributed capabilities
i960/Gemini	Intel	1989	integrated, object-based multiprocessing architecture
W7	MIT AI Lab [Rees95]	1995	Scheme as local-sequential-imperative lambda capabilities
Joule	Agorics [Tribble96]	1996	2nd unification of Actors and Logic styles of distributed capabilities
ToonTalk	Animated Programs [Kahn96]	1996	Animated capability programming for children
Original-E	Electric Communities [Morningstar??]	1997	Unification of distributed and local-sequential models of capability computation.
EROS	UPenn, JHU [Shapiro99]	1998	High performance open-source KeyKOS descendant. Formal verification of confinement.
E	Electric Communities, ERights.org, Combex [Miller00, Yee02b]	1998	1st language-based confinement. Guards, Auditors
Waterken	Waterken [Close99]	1999	Web integration: capabilities as URLs Language-based persistent capabilities
E-Speak2.2Beta	HP [Karp01]	1999	Split capabilities. Better scaling of complex policies.
SPKI	Intel, Electric Communities, Microsoft, MIT LCS, Southwest Bell, SSH [Ellison99]	1999	Public-key infrastructure as an off-line capability-like system.
CapDesk, DarpaBrowser	Combex [Yee02a, Wagner02]	2002	Virus invulnerable capability desktop and application launching framework

Major capability systems and milestones absent from [Levy84]. All dates are approximate -- these systems were or are ongoing projects without unambiguous birth dates.

The following diagram shows the influences directly relevant to our own work on E.



E in context. This diagram does not show all the major influences between these systems. It only shows the influences relevant to the creation of E, or of systems derived from E.

The **blue nodes** & **green arcs** are those that have "capability nature", even though most of these systems were not capability secure, and were not conceived of by their inventors as having any relationship to capabilities or security. Nevertheless, these are the systems to study in order to gain insight into the nature of capability computation. *Of the blue nodes, the ones in italics are actual capability systems.* Of the green arcs, the thicker **light-green** ones show the most influential paths. The thicker **light-green** arcs below E lead to systems that were derived from E in their original conception. The others are retrofits of E concepts into existing systems.

The loop: E and EROS have influenced each other. Both directly, and through our joint descendant, CapIDL.