A Security Analysis of the Combex DarpaBrowser Architecture

David Wagner
Dean Tribble
March 4, 2002

1 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 80 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.

2 DarpaBrowser Project

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

2.1 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:

2.2 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.

2.3 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 2 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.

2.4 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.

3 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.

4 Their Approach

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

4.1 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 ahold 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:

1. Does the E implementation correctly enforce capability discipline?
2. 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.

4.2 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.

4.3 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.

4.3.1 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.

4.3.2 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.

5 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.

5.1 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.

5.2 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

[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.

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 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.

5.3 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

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.

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 4 unsuppressed `hashCode()` methods, about 20 unsuppressed `equals()` methods, and about 13 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.

5.4 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.

5.5 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 memoizes 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 memoization 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.

5.6 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.

6 DarpaBrowser Implementation

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

6.1 Architecture

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

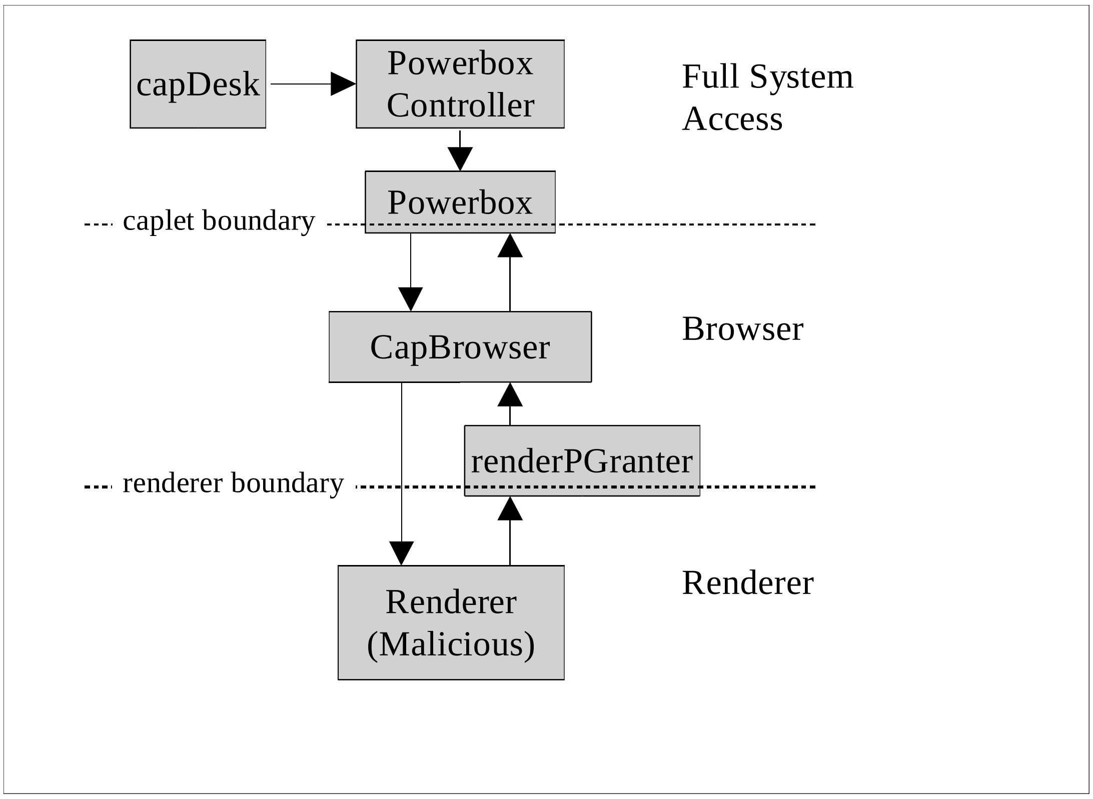

Figure 1 DarpaBrowser Overview

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`.

6.2 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.)

6.3 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.

6.4 `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.

6.5 `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.

6.6 `Powerbox`

The `Powerbox` further limits the capabilities granted to the `CapBrowser`. It allows the `CapBrowser` to call a restricted set of 19 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.

6.7 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.

6.8 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.

6.9 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.

6.10 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.

7 Risks

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

7.1 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.

7.2 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.

7.3 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.

7.4 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.

8 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.