How Emily Tamed the Caml

Marc Stiegler, Mark Miller
Advanced Architecture Program
HP Laboratories Palo Alto
HPL-2006-116
August 11, 2006*

security,
programming,
performance

How does one make a program breach resistant? One promising approach is to apply the Principle of Least Authority at object granularity. The E language has previously demonstrated that object-capability languages turn many of the security requirements for software into emergent properties of traditional object-oriented design and modularity enforcement.

Emily is a subset of OCaml that uses a design rule verifier to enforce object-capability principles. It demonstrates how memory-safe languages can be transformed into breach-resistant object-capability systems with little loss of either expressivity or performance.

How Emily Tamed the Caml

Marc Stiegler and Mark Miller

Hewlett-Packard Laboratories

marc.d.stiegler@hp.com, erights@hp.com

Abstract

How does one make a program breach resistant?One promising approach is to apply the Principle of Least Authority at object granularity. The E language has previously demonstrated that object-capability languages turn many of the security requirements for software into emergent properties of traditional object-oriented design and modularity enforcement.

Emily is a subset of OCaml that uses a design rule verifier to enforce object-capability principles. It demonstrates how memory-safe languages can be transformed into breach-resistant object-capability systems with little loss of either expressivity or performance.

Problem Statement

As amply demonstrated by breaches using ill-formed data in JPEG, ASN.1, Metafile, RAR, and MPEG formats [Cert04a, Cert04b, Cert05a, Cert05b, Cert06], the computer cracking community has embraced all forms of data input as vectors of attack. These attacks using multimedia data input formats are merely natural extensions of the techniques employed heretofore. Ross Anderson developed a simple model of the defender's problem [Anderson01] that suggested economic advantages were afforded the attacker by current software development techniques. His simple model predicted that the problem would get worse, not better. Current statistics summarized in *Figure 1* demonstrate the problem is indeed getting worse, even for widely inspected platforms [Bressers05].

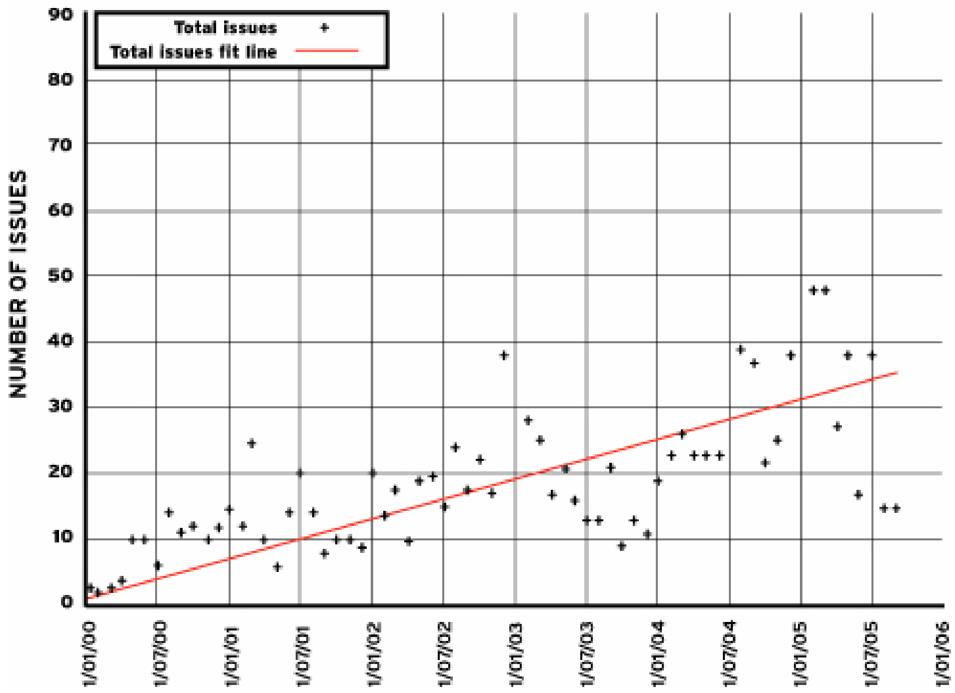


Figure 1. Recent history of vulnerabilities identified by Red Hat. This upward trend conforms to the predictions of the simple model developed by Ross Anderson, which suggested that the attacker has a dramatic economic advantage over the defender that increases as systems grow larger.

Beginnings of Solution: Principle of Least Authority (POLA)

One approach for reshaping the terrain to favor the defender is to apply the Principle of Least Authority (POLA) at finer granularities, minimizing the harm that can be done by any given breach. Currently, much work is focused on implementing various forms of confinement at the application level, reducing the harm that can come from a full breach of an individual application [JWS, Stiegler04a, Howard04, Balfanz00].

Even if these efforts were to achieve full POLA confinement at the application level, they would still be inadequate for some important purposes. Some applications must have enough authority to be dangerous, just in order to do their jobs. These applications already bear much of the brunt of current attacks. Full application-level POLA-based confinement of these apps, while a necessary step, will still disappoint. Such confinement is a first step, not a full solution.

Consider the email client. An email client must, in order to do its job, have both the authority to read the address book and the authority to send email in the name of the user. These are the only two authorities needed to implement a self replicating virus such as the early Melissa virus [Garber99].

Consider a malicious JPEG image embedded in an email. The JPEG image, when read by the rendering engine (see *Figure 2*), will achieve a full breach of the renderer. Because the renderer, as a part of the application, has all the application's authority, the attacker acquires both of the authorities needed to replicate itself. Confinement at the application level is insufficient.

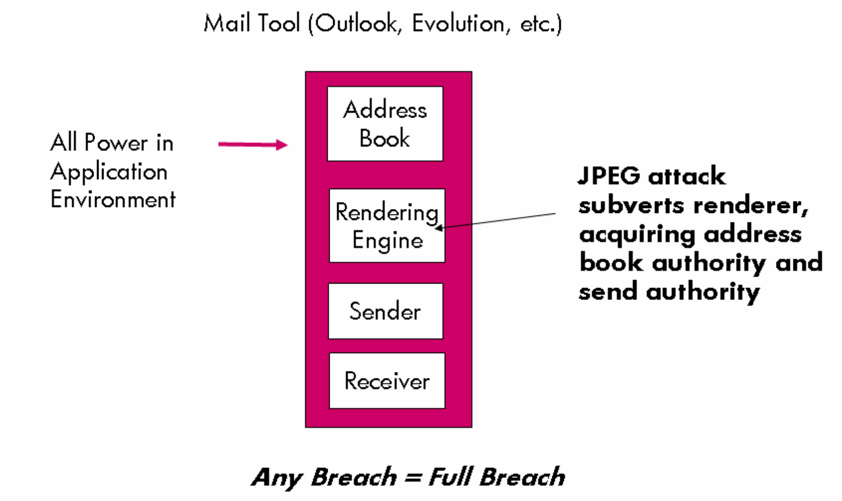

Mail Tool (Outlook, Evolution, etc.)
All Power in Application Environment
Address Book
Rendering Engine
Sender
Receiver
JPEG attack subverts renderer, acquiring address book authority and send authority
Any Breach = Full Breach

Figure 2. Architecture of conventional email client.A full breach of any module yields full access to all authorities held by the program as a whole. A malicious JPEG image, upon breaching the rendering engine, directly acquires both of the authorities needed for self replication (address book reading authority, and mail sending authority).

A More Extensive Solution

A straightforward extension of the idea of implementing POLA, first at the user-account level as partially done by conventional operating systems, and then at the application level with more modern approaches, is to continue implementing POLA at finer granularities all the way down to the individual software object. In *Figure 3*, we depict such a finer-grain (though not object-level) breakdown of the email client. In this version of the email client, the “main” module of the application (where execution begins) hands out to each module just enough authority to do the job needed by that particular module. So the address book manager has the authority to read the address book, but it does not have the authority to send or receive mail. It does not have these authorities simply because it does not need them: the address book manager is POLA-confined.

Now consider the rendering engine. The JPEG image, when displayed, still achieves full breach of the renderer. But the renderer has neither the authority to read the address book, nor the authority to send mail. The self-replicating JPEG image attack fails.

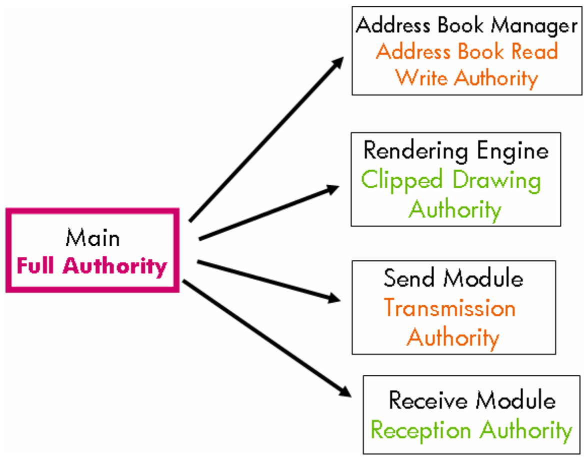


Figure 3. JPEG attack thwarted by applying POLA at the module level. Only the "main" module has enough authority to enable a self-replicating virus; no other module does. If the Main is small and difficult to breach, the attacker must successfully breach both the Send Module and the Address Book to replicate itself

In fact, a virus needs to achieve two independent breaches of the system to replicate: it must breach both the send module and the address book manager. Limiting the size of modules and number of modules that hold those authorities reduces risk of introducing a vulnerability. At the same time, requiring multiple breaches increases the difficulty of exploiting a vulnerability.

POLA down to the object level changes the assumptions implicit in the Anderson model of the defender/attacker relationship:

- Breaches are not all equal. Most breaches are weak. When POLA is applied all the way to object granularity, many objects carry no authority at all; a full breach of such an object does the attacker no good.
- Risk from most modified code is bounded. With traditional development strategies, modified lines of code represent new, additional full-breach risks.
- Risk from most new code is bounded. Experience to date suggests that new code, to add new features, often needs much less authority than the early core parts of the application that had to be developed before the application could work at all.

The simplicity of OCaml makes the transformation into Emily relatively straightforward. Consequently, construction of Emily presents a microcosm of the issues in transforming memory safe languages into object-capability languages. Furthermore, because OCaml is a high performance language (roughly comparable to C++), it makes a good platform upon which to explore the impact of object-capability rules, not only on expressivity, but also on performance.

Transforming OCaml into Emily

Emily may be thought of as OCaml with a set of verifier-enforced, POLA-oriented design rules. All Emily programs are OCaml programs, *i.e.*, all Emily programs compile correctly through an OCaml compiler. However, OCaml programs will usually flag an Emily verification failure.

To transform OCaml into Emily, the following rules are enforced:

- Memory Safety: Proper confinement requires strict memory safety. If a line of code can write someplace other than an explicitly referenced location, confinement is violated. OCaml is almost memory safe; indeed, it is so close to being memory safe that most OCaml programmers would be surprised to learn that it is not. To achieve Emily's true memory safety, the verifier ensures that the Marshalling module is not used, and that various undocumented, hidden-but-present memory-violating functions (such as

Array.unsafe_set) are disallowed.

- External calls to arbitrary executable binaries must be shut off. To enforce this, the keyword "external" is disallowed in the body of the program. It is only available in the *powerbox*, as explained below.
- Static mutables must be disallowed. Static mutables represent an uncontrollable channel through which information (and possibly authority) may be conveyed.
- More generally, ambient authority, *i.e.*, static availability of authority-bearing functions and objects, must be prohibited. For an example of ambient authority, consider the Ocaml Pervasive function

  open_out : string -> out_channel

  This is an immensely powerful authority, the authority to transform any file path available to the program into an editable channel. This function is pervasively available throughout OCaml programs even though very few functions in those programs actually need such power. A security reviewer of an OCaml program cannot be confident this power is not abused without examining every line of source for every module in the system. An attacker can seek out poorly-thought-out usage of such authority as an opportunity. The open_out function, and others like it, are denied by default in Emily.
- Providing the reference to an object must grant the permission to use the object. If a special object that does have the open_out function creates an out_channel, and hands it to another object, the recipient must be able to use the out_channel. This is different from the Java sandbox model, wherein being granted an explicit reference still does not enable you to use the file, the Security Manager will still do its coarse-grained access check.
- Not only must authority-bearing objects be removed from the global scope, but all the libraries associated with the language must be *tamed*. Taming attempts to ensure that the libraries remain easy to use while simultaneously ensuring that the programmers (and security reviewers) are not surprised by the amount of authority conveyed when a reference to one object is handed to another object.

  Going to Java for an example, if an object needed a read stream on a particular file, it would be intuitively natural to hand over a read stream manufactured from the File object. However, inspection of the java.io classes shows that, given such a read stream, the receiving object can cast down into a FileReadStream, from which it can getFile(), from which it can getParent(), thus walking the graph and acquiring full authority over the user's entire file system. Taming of the Java IO system requires suppressing the getFile and getParent methods in order to avoid surprising leakage of authority. Since these methods are seldom used in practical programming, their absence is not a significant inconvenience, though their presence would present a serious hazard.

For a taste of taming, the table below contains the list of keywords and Pervasive library functions (which are globally available in OCaml) that are rejected by the Emily verifier. All these identifiers convey authority. Many of them convey authorities that seem innocuous, but which could be dangerous given a sufficiently rigorous threat model.

For example, consider the grant of the most mundane of authorities, stdin and stdout. Given this pair of authorities, an application can present any data it desires, no matter how misleading, to a user who invokes the program from a command line. For example, the application could present exactly the same sequence of characters that a command line gpg tool might present, to trick the user into surrendering his passphrase. This would be a command-line based "phishing attack", analogous in form and intent to the phishing attacks against browsers that are today so prevalent.

Table 1. The thirty keywords and Pervasive functions in Ocaml that bear authority. These tokens are disallowed by the Emily verifier.

stdin	print_float	prerr_float	open_out	at_exit
stdout	print_endline	prerr_endline	open_out_bin	valid_float_lexem
stderr	print_newline	prerr_newline	open_out_gen	unsafe_really_input

print_char print_string print_int	prerr_char prerr_string prerr_int	read_line read_int read_float	open_in open_in_bin open_in_gen	do_at_exit external exception

The correct strategy with all these authorities, then, is to shut them all off, and then, if for a family of applications specific authorities are not a risk, turn them back on via the powerbox, described next.

One caveat must be noted. In all current POLA-oriented language efforts, at object granularity, two authorities are allowed by default: the authority to consume RAM and CPU cycles. Abuse of these authorities can lead to application lockup via resource exhaustion attack. While this can be a problem in some threat models, the danger is mitigated by the simple fact that if a program locks up, the user can shut it down and restart. No persistent authorities are endangered by such denial of service attacks.

The Powerbox Pattern

The powerbox pattern was invented during the development of CapDesk and the DarpaBrowser; the experiences with building both a good (secure and flexible) powerbox and a poor (rigid yet breachable) powerbox led to a better understanding of the concept [Wagner02, Stiegler04b].

In essence, the powerbox is a composition of objects that grants, revokes, negotiates, and in general manages, the authorities granted to another object. Typically the object to which a powerbox is assigned lies on the far side of a trust boundary, and the authorities for the differently-trusted object vary dynamically, at least on a session basis, or even during the session. The powerbox itself must have extensive authority. As a minimum it must hold all the authority that any of the applications it might service might need. The powerbox then *dynamically* doles out these authorities based on POLA, to the actual instantiated application. Sometimes, the powerbox even negotiates with the user on the application's behalf for additional authorities if the application finds itself unable to fulfill its obligations without more powers.

Powerboxes should be small and carefully reviewed, since they hold so much power. Ideally, a single powerbox can be built for a large family of applications, thus amortizing the cost of a security review over a larger value base, and enabling a user who trusts a single powerbox to run diverse applications from less trusted sources, as long as those applications are confined by the trusted powerbox.

From the perspective of the operating system, the powerbox is the "main" of the program. This powerbox has all the ambient authority in the user's environment, *i.e.*, it has just as much authority as the normal application launched on a traditional operating system. Conceptually, we can think of an application launch as being broken into a two-step process: first the powerbox is launched, and after the powerbox makes an initial determination of authorities to grant this application for this invocation, the powerbox invokes the "CapMain" (a "main" with capped authorities) to actually kick off application execution.

To access all the user authorities, the powerbox is not constrained by the taming regimen of the Emily verifier. It can invoke externals and call authority-bearing parts of the stdlib. The powerbox is written in OCaml, not Emily itself. It is responsible for encapsulating dangerous authorities with pola-disciplined wrappers, and consequently is a part of the taming system.

Emily Verification/Compilation/Linking Process

The Emily compiler takes as input a powerbox folder, an application folder, and an output executable name. The Emily compiler performs the four steps illustrated in *Figure 4*:

The first step is a quick scan of the application code to ensure that the disallowed keywords and Pervasive functions are not used. The second step is a full parse of the code that ensures no static mutables are present (at the present time, this step uses the OCaml documentation generator, with a special "doclet" as the visitor).

Third, a special compilation is performed using the OCaml compiler. The compilation is run against the safe (non-authority-bearing) version of the standard OCaml library, and against the safe part of the powerbox (typically the safe part of the powerbox is the set of interfaces to data types that the powerbox may share with the application; in the example below, the powerbox shares an interface describing a file object, though the code for actually instantiating a file object is held privately in the unsafe part of the powerbox). This compilation will fail if the application attempts to directly use any of the authority-bearing elements of the standard library or the powerbox. A successful compilation at this stage of the process ensures that the application is properly confined.

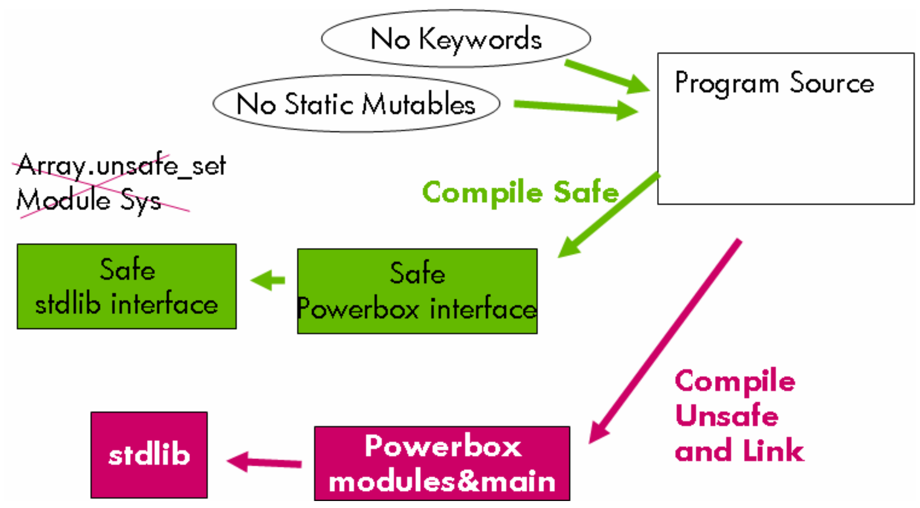


Figure 4. General Strategy for Emily verification/compile/link. After a quick check for keywords and static mutables, the source is compiled against safe versions of the stdlib and powerbox (examples of elements missing from the safe interface include the Sys module and the Array function unsafe_set). If the program passes this verification, it is recompiled and linked with the full stdlib and powebox.

Finally the OCaml compiler is called a second time, this time with the full (unsafe) standard library and the full powerbox. This version of the compilation is linked together into the final program. While this may sound like a tedious process, it is all automated.

A Simple Example: Sash

To demonstrate both the principles upon which Emily is based, and Emily itself, a simple powerbox was built for a small, but broad, family of applications. The powerbox is “sash”, a powerbox for implementing safe bash commands.

Many bash shell commands require surprisingly extensive authority considering the narrow purposes for which those commands are intended. Consider the “cp” command:

```
cp file1.txt file2.txt
```

The cp command is given two file names. Its purpose is to read the first one and copy the contents to the second. This seems like it should only require read authority on the first file, and write authority on the second. However, because the cp command starts out with only character strings describing the paths to the two files, the cp command must actually have at its disposal full read authority over every file the user might specify for the first file, and full write authority over every file the user might specify for the second file – in other words, the cp command must have full read/write authority over the user’s entire file space to achieve its intended goal. A buggy or malicious cp command can do immense harm, even if confined by POLA, because the least authority that cp needs is vast.

The sashcp command looks slightly different:

```
sashcp =file1.txt +file2.txt
```

In sashcp, the input and output files are prefixed with characters (“=“ and “+” respectively) that tell the sash powerbox how to transform the character strings into authorities for the cp application (the “=” and “+” signs are used because they are among the few simple symbols that are not pre-processed by the bash shell for its own purposes; for a better syntax for pola-enforcing shell commands, possible only when replacing the bash shell itself, see Plash [Seaborn05]). Seeing the equals sign, sash will manufacture a read-only file object to hand to the command; seeing the plus sign causes construction of a read/write file object. The sashcp command gets a more sensible interpretation of POLA than the standard cp command: sashcp gets the read and write authorities it needs for this particular execution of the command, dynamically identified and constructed at time of invocation.

Managing this fine-grained authority grant adds little burden to the user. The two additional characters, in addition to denoting the desired authorities, provide a useful mnemonic for the user. How often have programmers destroyed a file by reversing the order of the source and destination files in cp?

Similarly, while the ls command requires read authority over the entire user space (and is typically given full read/write authority, an enormous POLA violation), the sashls command fulfills all the user’s goals with the much smaller, dynamically granted authority over just the directory it is expected to list:

```
sashls =directory1
```

As long as the user executes only bash commands that were compiled and linked with a trustworthy powerbox, he could in principle acquire his implementations of sash-based bash commands from malicious sources, and yet have his risks be quite limited.

We will now look at the sash powebox, the sashcp command, and a small benchmarking progam, sashdeck, in more detail.

Sash Powerbox

The main body of the powerbox can be seen in Example 1.

```
(*** Sash Powerbox ***)
open SashInterface;;

let auths = List.map (fun arg ->
      let argUnprefixed = String.sub arg 1 (String.length arg - 1) in
      match arg.[0] with
              '=' ->  Readable (MleFileMaker.makeReadable argUnprefixed)
            | '+' ->  Editable (MleFileMaker.makeEditable argUnprefixed)
            | '^' -> (match argUnprefixed with
                  | "time" -> Time Unix.time
                  | "stdout" -> Stdout stdout
                  | _ -> raise (Invalid_argument "bad(^) request") )
            | _ -> Str arg
      ) (List.tl (Array.to_list Sys.argv)) in

let commandName = Sys.argv.(0) in

let userOut message =
      print_string ("Command " ^ commandName ^ " said: \n> ");
      let currentLineCharCount = ref 0 in
      let newline() =
            output_string stdout "\n> ";
            currentLineCharCount := 0
      in
      String.iter (fun next ->
            let nextInt = int_of_char next in
            if (nextInt >= 32 && nextInt <= 126) then (
                  output_char stdout next;
                  currentLineCharCount := !currentLineCharCount + 1;
```

```
                        if (!currentLineCharCount > 80) then newline()
                  ) else if (next = '\n') then
                        newline()
                  ) message;
            output_string stdout "\n"
      in
      exit (CapMain.start stdin userOut (auths))
```

Example 1. OCaml source for the sash powerbox.

Most of the operations of this powerbox are straightforward. It creates an authority list the same length as the array of incoming arguments. For each argument, it checks the first character. If the character is an equals sign, the powerbox creates a readonly file object. If the character is a plus, it creates a read/write file object. If the character is a caret, the powerbox recognizes that a special authority is being requested; at this time, only two authorities are recognized: a read authority on the system clock denoted by “^time”, and a direct authority on stdout denoted by “^stdout”. If the first character does not match “=”, “+”, or “^”, the argument is placed directly into the array as a string.

The file objects used here are not part of the OCaml standard library. In OCaml, the functions for file manipulation are spread across a number of modules and functions. Unsurprisingly, they are ill-suited for an object-capability environment.

The file objects used here were constructed and included in the system as part of the powerbox. The file object meets a standard interface specification: a file is an OCaml record with functions for getting an input channel, getting an output channel, get/set bytes, testing to see if the file is a directory, *etc*., as shown in *Example 2*. The Sash file object is quite similar to (though different from) the E file object, which has proven in practice to supply simple, intuitive, pola-disciplined interaction with the file system.

```
      type readable = {
           isDir : unit -> bool;
           exists : unit -> bool;
           subRdFiles : unit ->  readable list;
           subRdFile : string -> readable;
           inChannel : unit -> in_channel;
           getBytes : unit -> string;
           fullPath : unit -> string;
      }

      type editable = {
           ro : readable;
           subEdFiles : unit -> editable list;
           subEdFile : string -> editable;
           outChannel : unit -> out_channel;
           setBytes : string -> unit;
           mkDir : unit -> unit;
           createNewFile : unit -> unit;
           delete : unit -> unit;
      }
```

Example 2. The MleFile interface: an OCaml record with a set of functions. Any Emily module can manufacture objects with these signatures, but only the powerbox can create objects with these signatures that can actually read or write a file.

The next block of powerbox code creates the userOut function. The userOut function accepts a string as input, and outputs a forgery-resistant version of the string to stdout.

Earlier, we had given an example of an attack using stdin and stdout, in which the attacker formats the output to present itself to the user as a gpg command processor, and uses stdin to read the passphrase, in a phishing-style attack. The forgery resistant userOut strips out non-visible characters (preventing overwrite of the screen using escape sequences), postfixes every newline with a “>”, ensures that newlines occur often enough to prevent games with line wrapping, and prefixes the entire message with the name of the command the user invoked to launch the application, as shown in *Example 3*.

```
$ sashls =staticsTests
Command sashls said:
> Contents of dir:
>     simpleStatic.ml
>     nestedStatic.ml
>
$
```

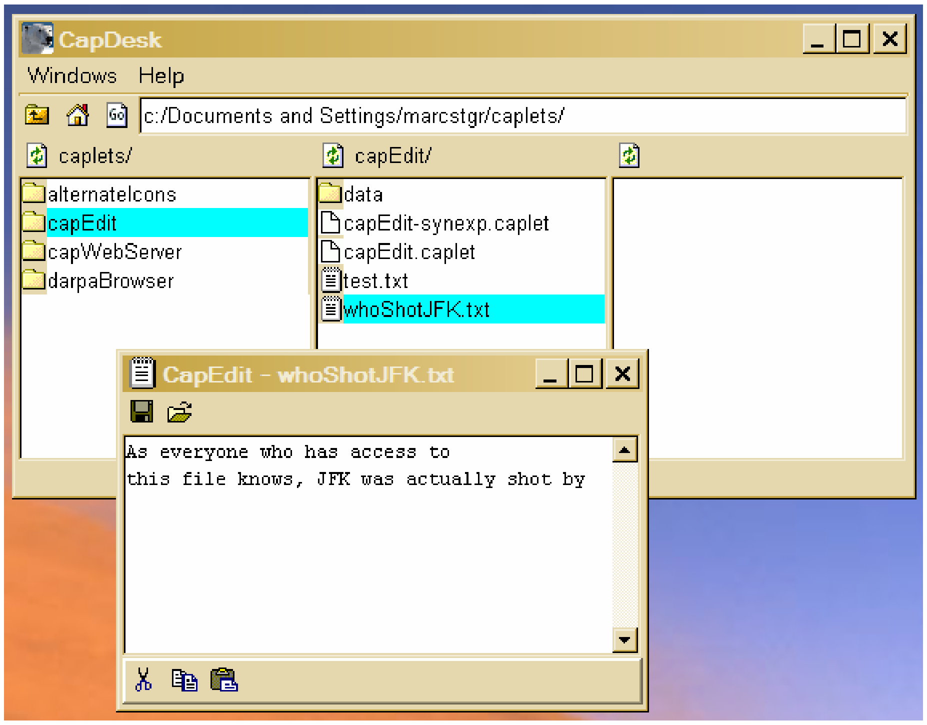
Example 3. Example forgery-resistant output from sashls

Hence the application cannot pretend to be any application except the one that the user actually launched. By using the name of the command given by the user, the powerbox implements a small but complete petname system [Stiegler05a], a solution for some global namespace problems.

The final step in the powerbox is to invoke the application itself. From the shape of the function call, we can see the following features of the powerbox/application interface:

- The application must have a module called CapMain. This module must have a function “start”, which receives as parameters an input channel, an output function, and a list of authorities.
- This particular powerbox always grants its application an *endowment* of authority simply for being launched. The endowment includes the stdin channel, so the application can read keystrokes directly from the input channel specified by the user (typically the user’s keystrokes in the terminal where the application was launched). The endowment also includes the userOut function, so the application can always output data to the channel specified by the user (but in a forgery-resistant fashion).

  This pair of endowments is analogous, in the command shell environment, to the default endowments granted by a CapDesk powerbox to a gui-based application, namely the ability to receive input directed to the application by the user (stdin in one case, typing and clicking in the application’s windows in the other), and the ability to output for the user (sending forgery-resistant, petnamed character strings in one case, opening forgery-resistant, petnamed windows in the other, see *Figure 5*).
- The application is then also granted exactly the needed file and time authorities deemed appropriate by the user for this particular invocation of the application.



```
As everyone who has access to
this file knows, JFK was actually shot by
```

Figure 5. CapDesk opening the CapEdit application on a sensitive file. As with Sash applications,CapDesk applications are launched using forgery-resistant techniques to ensure the applications cannot engage in window spoofing. The leftmost areas of CapEdit's title bar, containing the petname (CapEdit) and the pet icon (image of a notepad), cannot be edited or modified by the application. Consequently it is difficult for CapEdit to pretend, for example, to be the Quicken application and ask for the Quicken password. The application does, however, have append authority on its title bar. Hence it can present additional information, in this case the name of the file being edited. The CapDesk powerbox achieves this fine degree of control over the application's authority using the same POLA-oriented techniques as the Sash powerbox.

The Sashcp Example

Since the powerbox is already transforming the file names into file objects for its applications, the sashcp application is quite small, as shown in *Example 4*.

```
open SashInterface
let start userIn userOut authlist =
    match authlist with
    | Readable fromFile :: Editable outFile :: []  ->
            outFile.MleFile.setBytes(fromFile.MleFile.getBytes()); 0
    | _ -> userOut "To use cp, an input file and output file are required"; 1
```

Example 4. Emily code for sashcp.

If the user handed over a read authority and an edit authority, it makes a conventional copy, first file to second. Any other user input results in output of a help message.

The Sashdeck Benchmark, Emergent Security Properties

The Sashdeck program, given read authority on a clock and a number of runs to compute, shuffles a deck of cards. If a run of 4000 is requested, the program manufactures 4000 decks and shuffles each deck 4000 times.

Sashdeck is just large enough to enable us to start seeing the security properties of POLA-oriented programming emerge. Like all sash-based applications, sashdeck has a CapMain module, which receives from the user, in addition to stdin and userOut, the authority to read the clock, and a string that is converted into an integer count of the number of times to run the benchmark. However, CapMain does not create and shuffle decks itself. Rather, it invokes the Deck module, which in turn invokes the RandGen pseudorandom number generator module as shown in *Figure 6*.

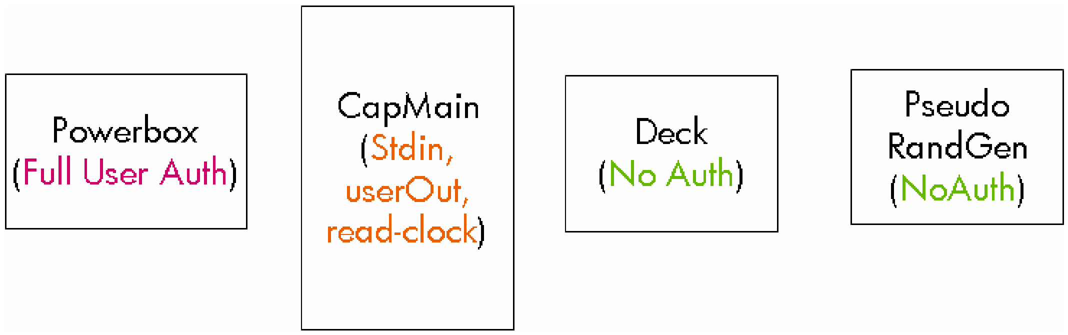


Figure 6. Modules in the sashdeck application. The CapMain receives a small amount of authority from the powerbox. The Deck needs no authority, and does not receive any. The random number generator, constructed by the deck, cannot receive any authority because the deck that manufactures it has none to give. The object relationships can be straightforwardly rearranged for different goals. For example, suppose the CapMain desired to use its clock-reading authority to randomize the pseudorandom number seed. One could reconfigure the hookup so that CapMain generates the seed, constructs a new RandGen, and hands the resulting RandGen to the deck. Both the RandGen and the deck would remain without authority.

CapMain itself starts the clock before repeated invocations of the Deck. All I/O is performed by CapMain itself. Hence the Deck does not need the clock, or the stdin, or the userOut. As one would expect when writing insecure but modular code, the Deck does not expect or receive any arguments that would convey such authorities. Hence it does not have any such authorities, *i.e.*, the Deck is strictly confined, it has no authority. Furthermore, since the Deck has no authorities, it necessarily is unable to grant any authorities to the random number generator. There are several consequences:

- A security review of sashdeck could probably conclude, by inspecting the Deck interface, that because the Deck receives no authority, there is no significant breach that the Deck could perform, even if the Deck were fully subverted. Consequently neither the Deck nor the random number generator needs to be reviewed. This dramatically reduces the number of lines of code that the reviewers must inspect, increasing the cost effectiveness of reviewing.
- Authorities tend to attenuate rapidly as the graph of objects is decomposed into smaller elements. Earlier experiences with POLA-oriented languages suggest that this effect grows stronger as the size of the program increases [Stiegler02]. From the cracker's perspective, the useful attack surface grows much more slowly than linearly as the number of lines of code in the program grows.
- By following simple, traditional object-oriented modularity rules, when using a POLA-oriented language, an emergent property is defense in depth. In this example, a full breach of the random number generator must somehow achieve a breach of the Deck, which must then achieve a breach of the CapMain, before any authority at all can be claimed.

The Sashdeck Benchmark, Performance Characteristics

Because Emily code is pure OCaml code, and because computationally intensive code blocks typically use little or no authority, Ocaml and Emily programs may have identical performance characteristics. Sashdeck reflects this: the code developed by the authors in Emily for the actual card shuffling is identical to what the authors would have written in OCaml. We would expect this to be true for many traditional benchmark programs, particularly for benchmarks focused on pure computational performance.

It is not the case that the code is always identical. It is possible to imagine, for example, a program for which array bounds checking was a critical performance impediment (though running sashdeck with bounds checking turned off, even though array indexing is a major part of the program, produced less than a 2% performance improvement). In OCaml, the bounds checks can be shut off. In Emily, they cannot be shut off.

The object-capability paradigm can also impact performance when more sophisticated access control policies are desired, at which point tradeoffs may arise in performance versus access management. In an object-capability system, sophisticated policies are enforced using intermediation [Stiegler05b]: rather than giving object A a direct reference to object B, we give object A a reference to a forwarder to B, and implement the desired policy in the forwarder. For many kinds of authority-bearing objects, the cost of the forwarding operation is minor. For example, a forwarder for the MleFile object used by sash would be quite cheap compared to the costs necessarily associated with the subsequent file operations. But there are circumstances where the cost of the forwarding can be greater than the cost of the operation being protected. A striking example is the mint [Miller00]. For a mint that manufactures money, and transfers that money among different accounts, the operation protected by the forwarder is typically a simple addition or subtraction in the number representing the balance of an account; the forwarding is surely more expensive than the actual mathematical operation.

Of course, when using the mint as an example, we have leaped from a situation in which the security goals of the program may be soft and tradable for performance, to a situation in which the security goals are clearly paramount. A bank that did not use object-capability forwarders would probably instead have to use access control lists and an extensive client-authentication protocol for access control. A comparison of the performance characteristics of the forwarder to such a client-authentication system is beyond the scope of this paper.

Enabling intermediation at all may have a moderate impact on performance. The authors briefly compared 3 different ways of creating “objects” in OCaml: the use of modules, the use of classes, and the use of records filled with functions. Modules do not naturally support intermediation, but modules have the highest performance. Using classes had a small but measurable impact on performance. Function-filled records had a smaller impact than classes, but the effect was still measurable. Function-filled records naturally support intermediation, which is why the MleFile object is implemented as a function-filled record.

Programming Style Changes Required

Programming in Emily rather than OCaml does have some impact on programming style and programmer behavior. Some impacts come from the lack of the usual ambient authorities, some come from the opportunity afforded to achieve higher levels of reliability and robustness made possible by more resolute enforcement of POLA.

It would seem at first glance that the biggest impact would be the requirement to pass so much authority around as explicit arguments all the time. Upon hearing that authorities must be passed as explicit references, the student first being introduced to object-capability languages could be forgiven for groaning at the thought of having to pass long lists of authorities explicitly throughout his program.

In practice, this problem is less severe than it might seem. There are several reasons. One is that, as shown in the sashdeck example, many objects need little or no authority, so there are few or no extra parameters required. Another reason is that, often when a function does need some authority, the function needs its caller to specify which authority should be used for this invocation anyway, in which case there is once again no additional parameter. For example, to have a function write to a file, in vanilla OCaml one would either need to tell the function the path to the file (in which case the function needs to use open_out to get the file) or would need to hand the function an out_channel to the file. Emily enforces the second implementation style: since the function cannot invoke open_out, you must pass the channel. Many people would consider the second design to be cleaner and more modular even in the absence of POLA enforcement. Indeed, experience suggests this is often true: POLA-oriented design can frequently be thought of as “modular design taken seriously”.

Having said this is not as serious as it might be, there are circumstances in which the requirement to pass the additional arguments can be aggravating. One specific item has been identified in the E work for which the desirability of an ambient authority seems to outweigh the risks. This item is the trace log for debugging. Passing the trace log to every object that might need debugging is a major distraction from the focus on solving the programming problem. Powerbox authors may want to include in the powerbox an ambient authority, available throughout the application without explicit passing, for writing to the log.

Another difference in POLA-oriented programming is that POLA enforcement makes currying attractive more often. A common pattern is to take a generic function, apply it to the authorities required for a particular operation, and then pass the curried result as the function to use. In this fashion the authority is not directly exposed, and does not have to be explicitly passed; it makes for code that is both simpler and more modular. This effect was first noticed in [Wagner02].

Finally, as noted earlier, if more sophisticated access policies may be desired for some element of the system, the function-filled record is the better way to manufacture objects than either through the direct use of modules or classes.

Issues For Emily Users

The biggest single issue with the current version of Emily is that the taming of the libraries is incomplete. A comprehensive pass was taken at removing all authorities from the safe version of the standard library. However, while a proper wrapper for the file system was created, there are some important authorities that are not now wrapped in a POLA-disciplined fashion. These elements of the system must be wrapped by the individual powerbox author to the extent needed for his application area. The most important of these elements are:

- Networking: The powerbox author needs to wrap the networking machinery for accessing sockets, creating servers, *etc*. For many applications this does not have to be complex, but it does have to be done. Fortunately the in_channel and out_channel primitives in OCaml are (approximately) POLA-disciplined, and just as they are used directly in the sash File interface, so too can they be directly used in the networking realm.
- Threads: The threading module is not included in the safe standard library. The traditional approach to threading is not only error prone, it can give rise to security breaches beyond those from which POLA can offer protection, notably Time Of Check to Time Of Use breaches [Abbott76]. Also, threading enables a small violation of POLA: it confers a source of nondeterminism that can be used to listen to covert channels. We hope to augment Emily in the future with a promise pipelining model of concurrency [Miller06].
- Graphical User Interfaces: The OCaml standard library includes an interface to TK. An interface to GTK is available. Neither of these operate according to POLA-oriented guidelines, and neither is tamed. Taming a gui toolkit is a significant undertaking. Taming just enough TK or GTK to enable the construction of simple dialog boxes should be straightforward.

A related important issue is that Emily and its verifier/compiler have not been subjected to a security review. Consequently, it is quite possible, even probable, that there are ambient authority leaks. Nonetheless, even in the absence of such a review, programs that follow the Emily discipline should be more resistant to breach than programs written in non-POLA-oriented languages. However, until such a review is completed, serious risk remains when using libraries from untrusted (possibly malicious) third parties. The earlier example, of allowing world-renowned crackers to write your card deck shuffle modules, is not yet recommended.

There are a number of limitations and weaknesses in the details of Emily’s implementation. These include:

- For the moment, the analyzer that verifies the absence of authority bearing key words and Pervasive functions is overly simple and fails safe. As a consequence, any use of the forbidden words as distinct tokens – even in a comment or a literal string – will cause a verification failure.
- The analyzer that verifies the absence of static mutable state is overly simple and fails safe by catching all static variables, even if they are not mutable. Hence, a module that wanted to declare a public constant:

```
let pi = 3.14
```

would have to instead declare a function that returns the constant:

```
let pi() = 3.14
```

- Strings in OCaml are mutable. In general, this mutability is unused. The consequence is that, in the programmer's mental model, strings can often be thought of as being immutable. This disparity between mental model and actuality creates a hazard. While mutable strings cannot bear authority, they can become uncontrollable overt data channels between objects that have no visible relationship on the reference graph, a violation of POLA that is problematic in the context of some threat models. The next release of Emily will probably shut off the functions that enable mutation of strings. The consequence is that powerbox authors should anticipate this, and include tamed versions of the in_channel input functions: the standard library functions that return data from streams by mutating existing strings must be wrapped with tamed functions that instead either return new strings, or mutate character buffers or arrays (arrays will be left mutable since experience suggests that programmers rarely forget that arrays can be modified).
- In OCaml, expressions return the last evaluation by default. In the context of POLA-oriented programming, another way of saying this is, "OCaml functions leak authority by default". Early work with the E programming language suggested that this hazard was so severe it could not be avoided even by seasoned functional-style programmers. E was modified so that functions and methods return null by default.

  OCaml and Emily are different from E in a number of ways that seem to mitigate this problem. Because of the strong data typing, interface definitions always require explicit typing, returns of surprising types will often generate compiler warnings and errors, and the type of every element is always on the programmer's mind. While none of these differences is a certain cure for the hazard, experience to date suggests that the risk may be adequately mitigated by these differences. Further research, in larger programming efforts, will be required to characterize the actual risk and enable assessment of appropriate countermeasures.
- Emily programs are not allowed to create new exception types. In OCaml, new exception types can be created that bear arbitrary authority. Technically, authority-bearing exception types do not violate the principles of POLA: a careful programmer could consistently catch all exceptions and rethrow only those that he deemed appropriate. In practice, however, exceptions may transfer authority between objects that have no visible connection on the reference graph, and pose a considerable hazard. Until a more refined exception verification mechanism is developed, powerbox authors will have to create exception types appropriate to the application family targeted by the powerbox. Emily programmers can in any event use all builtin exception types from the standard library as well as any types offered by the powerbox. Since many of the builtin exceptions are parameterized with strings, one can for many purposes mimic the creation of a new excepton type simply by discriminating among parameterizations.
- There is a source of nondeterminism available via the catching of the out-of-memory exception. Eliminating this may require a modification to the OCaml compiler and/or runtime support. Within the context of many threat models, this ambient source of nondeterminism does not create a significant vulnerability.

While the following is not a significant handicap for the types of applications for which Emily is currently proposed, there is one more weakness in Emily that is inherited from OCaml. POLA-oriented languages support a rich diversity of securely cooperative patterns among objects that do not and cannot trust one another except within rigorously constrained POLA boundaries [Stiegler04b]. As noted earlier, these patterns of secure cooperation depend heavily on the ability to manufacture objects that perform intermediation [Stiegler05b]. OCaml, with its pure and rigorous static type checking, makes implementation of these intermediaries difficult or even impossible. In E, a general purpose maker of membranes can be written in 24 lines of code [Miller06]. In OCaml and Emily, such membranes must be hand-crafted for each type of object, the crafting is only possible if the objects have been designed to support intermediation, and the programming for each custom membrane requires creation of revocable forwarder code for the transitive closure of all types reachable from the wrapped object. Unless the OCaml community embraces some mechanism ensuring simple and rich intermediation, development of such facilities seems an inevitable direction of evolution for Emily to pursue independently.

Conclusions

With POLA-oriented languages like Emily, traditional object-oriented modular architecture yields a rapid attenuation of authority for individual objects. This attenuation decreases the attack surface, thus achieving higher levels of breach resistance. It is possible that the emergent property of defense-in-depth in POLA-oriented languages can modify the attacker/defender relationship enough to give the defender an enduring advantage.

The Emily compiler, as a member of the POLA-oriented language family, delivers these benefits in combination with OCaml-comparable levels of performance. Emily is effectively a thin layer of design rule verification laid on top of OCaml. Consequently, the OCaml tool chain, documentation, and other associated materials, can all be used directly for Emily development. Because OCaml is a mature development system, it is possible, even with this first draft implementation of Emily, to engage in significant projects without fear that limitations and bugs in the language will put the project in peril. The main libraries missing from Emily at this time are POLA-disciplined wrappers for networking, threading, and graphical user interface; users of Emily currently must manually wrap the OCaml libraries to an extent sufficient for their application family. An important direction of further research is enhanced machinery to support object intermediation, to easily support forms of secure cooperation such as the membrane pattern.

References

[Abbott76] Abbott, R. P., Chin, J. S., Donnelley, J. E., Konigsford, W. L., Tokubo, S., and Webb, D. A., “Security Analysis and Enhancements of Computer Operating Systems,” NBSIR 76–1041, Institute for Computer Sciences and Technology, National Bureau of Standards (Apr. 1976).

[Anderson01] Ross Anderson, “Why Information Security is Hard: An Economic Perspective,” 2001 http://www.acsac.org/2001/papers/110.pdf

[Balfanz00] D. Balfanz and D.R. Simon. “Windowbox: A simple security model for the connected desktop.” In *Proceedings of the 4th USENIX Windows Systems Symposium*, August 2000.

[Bressers05] Josh Bressers, “The Sisyphus Security Dilemma,” 2005 http://www.redhat.com/magazine/013nov05/features/sisyphus/.

[Cert04a] Microsoft Windows JPEG component buffer overflow, http://www.us-cert.gov/cas/techalerts/TA04-260A.html

[Cert04b] Multiple Vulnerabilities in Microsoft ASN.1 Library, http://www.us-cert.gov/cas/techalerts/TA04-041A.html

[Cert05a] Symantec RAR decompression library contains multiple heap overflows, http://www.kb.cert.org/vuls/id/305272

[Cert05b] Microsoft Windows Metafile Handling Buffer Overflow, http://www.us-cert.gov/cas/techalerts/TA05-362Apr.html

[Cert06] Apple QuickTime MPEG-4 Movie Buffer Overflow, http://www.kb.cert.org/vuls/id/587937

[Garber99] Lee Garber, “Melissa Virus Creates a New Type of Threat” *IEEE Computer*, 32(6), pp. 16-19, 1999.

[Howard04] Michael Howard “Browsing the Web and Reading E-mail Safely as an Administrator,” *MSDN Library*, Microsoft, 2004, msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure11152004.asp

[JWS] Java Web Start White Paper, http://java.sun.com/developer/technicalArticles/WebServices/JWS_2/JWS_White_Paper.pdf

[Miller00] Mark S. Miller, Chip Morningstar, and Bill Frantz. “Capability-based Financial Instruments.” *Proc. Financial Cryptography 2000*, pages 349-378, 2000. www.erights.org/elib/capability/ode/.

[Miller06] Mark S. Miller. “Robust Composition: Towards a Unied Approach to Access Control and Concurrency Control." PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. www.erights.org/talks/thesis/.

[Seaborn05] Mark Seaborn. “Plash: The Principle of Least Authority Shell,” 2005. plash.beasts.org/.

[Stiegler02] Marc Stiegler and Mark S. Miller. “A Capability Based Client: The DarpaBrowser.” Technical Report Focused Research Topic 5 / BAA-00-06-SNK, Combex, Inc., June 2002. www.combex.com/papers/darpa-report/.

[Stiegler04a] Marc Stiegler, Alan H. Karp, Ka-Ping Yee, and Mark S. Miller. “Polaris: Virus Safe Computing for Windows XP.” Technical Report HPL-2004-221, Hewlett Packard Laboratories, 2004. www.hpl.hp.com/techreports/2004/HPL-2004-221.html.

[Stiegler04b] Marc Stiegler. “The E Language in a Walnut,” 2004. www.skyhunter.com/marcs/ewalnut.html.

[Stiegler05a] Marc Stiegler. “An Introduction to Petname Systems.” In *Advances in Financial Cryptography Volume 2*. Ian Grigg, 2005. www.financialcryptography.com/mt/archives/000499.html.

[Stiegler05b] Marc Stiegler, “A PictureBook of Secure Cooperation,” 2005, www.skyhunter.com/marcs/SecurityPictureBook.ppt.

[Wagner02] David Wagner and E. Dean Tribble. “A Security Analysis of the Combex DarpaBrowser Architecture,” March 2002. www.combex.com/papers/darpa-review/.