Objects as Capabilities in Squeak

Lex Spoon

1. Overview

In order to widely share Squeak pages across the Internet, it will be necessary to engineer a safe display environment--a sandbox--for such pages. Specifically, once a page (graph of objects) is loaded and a process instantiated to administer the page's active aspects, it is desirable to restrict that process both from invoking harmful operations, and from corrupting the greater Squeak image that the page has been loaded into. This document describes a project to implement such restrictions without stopping normal morphic interactions.

A fundamental perspective in this project is to view each object reference as a capability. That is, an object reference provides a limited list of operations on a specific resource-- the operations are the object's defined methods, and the resource is whatever resource the object refers to. Under this perspective, security policies may be implemented by carefully analyzing the initial set of objects provided to a process, and the mechanisms by which more objects may be maintained and modified. If a process begins with a set of objects that are considered safe, and those objects provide no way to obtain an object that is unsafe, then that process has been successfully restricted.

Overall, this project does not completely implement a sandbox for viewing Squeak pages. However, it does provide a good basis. This document describes the mechanisms that have been put in place, and it describes some design approaches for working with these mechanisms.

2. The ObjectInspector Capability

ObjectInspector is a powerful capability that enables many of the mechanisms described below. Briefly, ObjectInspector allows inspecting objects, even when those objects don't define methods such as #class or #instVarAt:. The full list of messages that ObjectInspector defines is:

1. classOf:
2. instVarOf:at:
3. instVarOf:at:put:
4. elementOf:at:
5. elementOf:at:put:
6. replaceIn:from:to:startingAt:
7. digitOf:at:
8. digitOf:at:put:

ObjectInspector reuses existing primitives in its implementation. For example, #classOf: uses the same primitive as the regular #class method. For this reuse to work, it is necessary to make small changes to the implementation of those primitives in the interpreter.

3. Modifications to the Language

As is, Squeak has several mechanisms that let a process break most security policies, even if that process is given an extremely innocuous set of initial object references. This section describes how these mechanisms are adjusted in order to make security enforcement practical.

3.1. Literals

By default, literals refer to shared instances. If a method operates on a literal such as ’hello, world’, then every invocation of that method will refer to the same literal. The security issue is that a restricted process may modify a literal, and those modifications can influence an unrestricted process. At the least, outside processes can be made to act in strange ways -- for instance, browser windows might be labelled "Happy Hippos" instead of "System Browser". More seriously, array literals can have their elements replaced by arbitrary objects, ultimately leading to arbitrary behavior when an unrestricted thread sends a message to one of those objects.

To prevent these possibilities, this project makes all literals immutable. `Symbol`, `Character`, `Float`, and `ScaledDecimal` are modified to be immutable, while `Array`, and `String`, `LargePositiveInteger`, and `LargeNegativeInteger` each are given a read-only variant: `ReadOnlyArray`, `ReadOnlyString`, and so on. (Note that a `ReadOnlyString` is not identical to a symbol: when sent #copy, a `ReadOnlyString` returns an equal String, while a `Symbol` returns the receiver).

One complication is the initialization of read-only literals: how does an immutable object get its first value? In most cases, ObjectInspector is used to initialize the object. The sole exception is `ScaledDecimal`, where the only initialization method is modified to fail if called on an already-initialized object.

3.2. Dynamic Variables

Global, pool, and class variables might collectively be called "pool" variables. In standard Squeak, all pool variables are statically bound with a single, system-wide binding, causing a difficulty for security: modifications to the variable or to the object it references will be seen and acted on by other processes in the image. Unless care is taken for each such variable, a restricted process will often be able to trick an unrestricted process into doing something on its behalf.

This project gives each process its own set of pool variable bindings. Those bindings may be shared with other threads if desired, but most importantly, it is possible to give a process completely unique bindings. When such a restricted process accesses

"World", it will be accessing a different variable binding than an unrestricted process which accesses "World". If a restricted process executes "World := nil", then that restricted process will be only causing itself difficulties; the rest of the system will be see the same World as if the unrestricted process were not even running.

Incidentally, methods that are marked as privileged, are still be able to access variables through a static binding. If a variable access is preceded by an exclamation mark, then the standard system binding will be used. For example, "!World := nil" can be used to destroy the world even if the current process has a restricted island installed.

The implementation of dynamic variables works as follows. First, each process is given a current runtime context, called an "island" in the prototype. Second, the compiler is modified so that pool variable references are implemented as message sends to the current island. Finally, to aid performance, a simple cache is added at each variable access in a method; with this cache, a full lookup of a variable binding can often be avoided.

3.3. Privileged Methods

There are several structures that Squeak normally allows methods to define, but which allow breaking most restrictive security policies:

- declaring a method as primitive
- statically binding to a pool variable
- accessing thisContext

In this project, the compiler has been modified to disallow each of these mechanisms by default. System programmers may override this default by marking individual methods as "privileged"; privileged methods may use the full Smalltalk language. Whenever code is loaded from untrusted sources, it should be loaded into unprivileged methods.

In order to implement these restrictions, a list of privileged selectors is added to class `ClassDescription`. Processes with direct access to classes are able to add and

remove selectors to these lists. The compiler is modified to check this list after it has processed a method's header, and depending on whether the method's selector is in the list or not, it will or will not allow the method to contain privileged operations.

3.4. Restricting Blocks

Blocks are an important part of Smalltalk code, but current Squeak blocks are more powerful than they need to be. For example, access to a block allows a programmer to modify the bytecodes of the compiled method the block was defined in.

This sandbox scheme modifies the compiler to generate `RestrictedBlock`'s instead of regular `BlockContext`'s. A `RestrictedBlock` allows safe messages like #value and #value:, but does not allow messages like #home:startpc:args. `ObjectInspector` may be used to extract the underlying `BlockContext` from a `RestrictedBlock` as follows: (realBlock := ObjectInspector realBlockFor: aRestrictedBlock).

3.5. Making Classes Safe

Direct access to classes allows a number of attacks. The class hierarchy might be made cyclical. The method dictionary might be replaced by an integer. More deviously, new methods might be installed into a class which do not obey the usual security restrictions. Class objects must be restricted in some way; on the other hand, it is quite common for Smalltalk code to access classes, and existing code should be allowed to work as far as is reasonable.

The general approach in this project is to intercept all accesses to classes, and to only allow safe operations to be applied to them. In particular, the following two access routes are modified for restricted processes:

1. Global variables referring to classes now refer to restricted proxies to classes
2. The #class method returns a restricted class when invoked in a restricted process

Deciding how a restricted class should behave has been "interesting". The current implementation allows the following methods to be executed on a restricted class:

- A small hand-chosen list of methods including #basicNew and #name
- Methods that aren’t marked "privileged" and that don’t access class instance variables

The general goal is to allow both instantiation methods and utility methods. An additional nicety is that the above requirements may be checked quite quickly.

In addition to deciding what methods are reasonable, it is also "interesting" to decide how those methods should be executed. In particular, many class methods return "self", but "self" in this case would normally be a direct reference to a class. Allowing such methods to be executed as normal would allow a restricted process to gain direct access to a class. The solution implemented in this project is to implement class methods so that "self" refers to the restricted class proxy, and not the class itself. The method #valueWithReceiver:withArguments: handles most of the work required.

One remaining difficulty is supered message sends. Supered message sends are looked up starting in class other than the default. Normally in Squeak, the interpreter decides which class to use by looking at the last literal defined in the method. One simple implementation would be to add a parameter to valueWithReceiver:withArguments: which overrides whatever value is in the method header. This implementation requires tricky interpreter changes, however, and so this project instead includes some image-level trickery to achieve the same result.

To make supered message sends work, whenever a class proxy is about to invoke a method that includes supered sends, it copies the method and specifies in the new method that a new, specially created class should be used for supered message lookups. This special class only implements #doesNotUnderstand:, and it implements it with the same method that `RestrictedClass` does. Finally, the special class contains an instance variable referring to the class where the real lookup should occur, so that #doesNotUnderstand: can find the correct implementation.

Overall, implementing class proxies is one of the trickiest part of this system. The primary problem is that classes are a place where Smalltalk mixes language

implementation and regular user code, and so trickery is required to allow access one of these aspects but not the other.

3.6. Moving Primitives from Base Classes

The current system defines a number of primitives in base classes like Object, which shouldn't be available to untrusted code. Such primitives should be moved to external capabilities.

An incomplete list of such methods is the following:

- someObject and nextObject. These allow iterating through all objects in the system and thus clearly break confinment. They are moved to class `SystemDictionary` and #allObjectsDo: is modified to use the new locations.
- instVarAt: and instVarAt:put:. These can remain except for proxies, but in proxies they must definitely be removed. These methods are made available in class `ObjectInspector`.
- become:. First, the existing implementation can cause image crashes, and second, it is difficult to allow become: between objects in a sandbox but still to prevent become: between a sandboxed object and a proxy. Well-written Smalltalk rarely needs this facility, however, so leaving it out shouldn't be too much of a burden.

4. Restricted Access to Common Resources

This project modifies several common resources so that a restricted process may be given limited access to them. The usual technique is to remove power from primitives directly implemented on an object, and instead to add a global variable with that power. In a restricted process, the global can be replaced and thus security-critical requests can be intercepted.

4.1. Characters and Symbols

The Squeak implementations of `Character`'s and `Symbol`'s rely on class variables to ensure the uniqueness of equal instances. For example, there should be only one character in the system with ASCII value 65. These class variables must be shared across an entire Squeak image to achieve their purpose, but they must also be protected from corruption by untrusted code. For each class, these dual goals are achieved in a different way.

For `Character`'s, the state in the class variables is moved to a globally accessible object named StandardCharacterRepository. This object can be easily checked to be safe against malicious access, because it is so simple.

For `Symbol`'s, the shared state is protected in a different way, largely due to lack of the implementor's time. First, a mutex is created which is held during all accesses to the shared state. Second, all methods which access the mutex or the other shared variables are reviewed.

4.2. Mouse Cursors

Currently, class `Cursor` directly implements methods #beCursor and #beCursorWithMask: as primitives. Thus any code which accesses a `Cursor` may immediately install it system-wide.

To address this, the ability to install a cursor is removed from cursors themselves and located in a global object named CursorInstaller. In unrestricted processes, CursorInstaller responds to #installCursor: and #installCursor:withMask: with primitives. In restricted processes, CursorInstaller will typically be a proxy. This proxy might or might not immediately install the cursor as requested, depending on the precise security policy that is being implemented.

4.3. Semaphores and Delays

`Semaphore`'s and `Delay`'s both have a complex structure and a simple interface. Both

access processes explicitly, and might allow for mischief if untrusted code can access them directly. Thus it is natural to arrange that all accesses go through a restricted proxy. The proxy classes are restricted and reviewed as normal. To ensure that only restricted proxies may be accessed, the globals Semaphore and Delay are adjusted in restricted processes. Instead of referring to classes, these variables refer to special objects that implement methods like #forSeconds: and #forMutualExclusion to return safe proxies.

4.4. Exceptions

Properly executed exceptions do not cause any security troubles, as they merely rearrange control flow. However, exception objects provide numerous possible ways to subvert the system. Probably, exceptions should have been proxied the way blocks, semaphores, and delays are, but time has been too short to guard exceptions very well.

Nevertheless, all of the most obvious holes with exceptions have been closed. For example, the #initialize: method is removed, as is #receiver. Furthermore, the method to find the handler context when an exception occurs, is moved to a privileged class method and thus is no longer accessible to restricted processes.

5. Guidelines for Implementing Restricted Proxies

Most security-critical objects written in this project are restricted proxies to a more powerful object. These objects allow partial access to an underlying resource. Most likely more of these will be written in the future, and so this section tries to give some suggestions on how to implement them in a reliable way.

There are two important properties that such a restricted proxy must hold. First, a proxy must limit access to the underlying resource as is appropriate for the particular proxy--for example, a proxy on a `FileDirectory` might allow only accessing files in

one particular directory but not in others. This kind of restriction is specific to the resource being defined, and will not be further addressed in this section.

Second, a proxy must carefully guard the object references that it returns from its methods. It must neither transfer a powerful object to the restricted side, nor transfer an arbitrary restricted object to a place where an unrestricted process might act on it. There are two basic elements to this restriction.

First, certain primitives normally inherited from class `Object` must be overridden and disabled for proxy classes. In particular, the following primitives should be disabled:

- instVarAt: and instVarAt:put:, because they allow directly breaking confinement
- at:, basicAt:, at:put:, and basicAt:put:, if the proxy has indexed fields, because they would allow directly breaking confinement

Additionally, #shallowCopy and #clone make revocation much more difficult; thus, they should most likely be overridden to return the receiver instead of returning a true copy.

Note that all non-primitive methods from class `Object` may be safely left accessible. Since such code must consist of message sends between parameters, self, and globals, user code could emulate the code even it it were disabled, and so disabling such methods gives no gain in security.

The second element is that, for each method that is privileged or which accesses instance variables, the inputs and return values of the method must be scrutinized. The return value of a method must not be an object reference which might have come from one sandbox or the other. The argument to any parameter sent to an instance variable or a parameter, must again not be a reference which might have come from one side or the other.

For input parameters, the simplest approach is to always filter them through one of several new safeXFor: methods. These methods simultaneously check the type of an object, and return a newly allocated copy of the object. These methods may also be used to copy return values; while return values don’t need to be type-checked, they do need to be new objects.

An exception to this policy is that references to certain immutable objects may be passed freely from one space to another. In particular, characters, symbols, and small integers may be safely shared. Such objects do not allow further interaction between spaces, and they do not have problems when accessed from multiple threads.

6. Denial of Service

In addition to attacks that delete files or munge the screen, there are denial of service attacks which are irritating if not as damaging. In particular, a loaded project might run continually and hog the CPU, or it might allocate objects endlessly and thus steal memory from the rest of the system.

The problem of hogging CPU is easily addressed by running sandbox-ed threads at a lower priority than threads in the foreground. This problem is not addressed further.

The problem of stealing memory is more difficult. No solution has been implemented in this project, but here are some ideas towards a solution.

In some fashion, the amount of memory that a sandbox may use at one time needs to be limited. Garbage collection makes this more difficult: it isn't reasonable to put a limit on the amount of memory allocated, but instead, the limit should be on the total amount of memory allocated at one time.

The most straightforward solution seems to be to have separate object memories, and to force a sandbox to allocate from a particular memory separate from the main system's memory. Then, a project can only run its own memory out of objects, and can't steal memory from the main system.

There are multiple ways to implement separate memories, but perhaps the simplest is to add a notion of submemories. A submemory is a special kind of object that itself contains other objects. The garbage collector would need to be updated to recurse into submemories, and the allocator needs to take a parameter which is the submemory to use. A primary advantage of the submemories approach is that there is no need to add a special kind of cross-memory oop; regular oops will continue to work as they are. A second advantage is that the mechanism works on all platforms: there is no need to

allocate new memory as Squeak runs. While I am far from an expert on object memories, submemories seem like a reasonable and simple design.