An Analysis of Javascript Security in Mozilla

COMP 527 Final Project Proposal
Fall 2006

Blake Kaplan Gregory Malecha

December 11, 2006

1 Background

One of the most significant technological innovations in the past 10 years has been the emergence of the Internet as the dominant medium for large scale content distribution. Its popularity has, amongst other influences, created a large market for distributing information and services over it. *Dynamic HTML* and *AJAX* are two technologies that allow for the creation of interactive web pages, beyond the simple link-clicking model provided by the early web browsers. It is because of these technologies that web browsers must support large and varied JavaScript *API*s and expose them to web pages to allow those pages to have maximum control over their content.

Mozilla Firefox is a modern open source web browser that is currently growing in popularity. It provides both the rich *API* described above for web sites and a wide platform for extensions based on the same technology, JavaScript. We have chosen to focus on Firefox because of its growing popularity and the open nature of its source code.

2 Problem

The *API*s provided to web pages were often created without consideration of their security ramifications. Therefore, Firefox, as a modern web browsers, must protect themselves against malicious web pages. Firefox has the additional problem that its rich extension *API* must also be protected from these malicious pages. This protection even extends to cases where an extension might want to interact with a such pages. This allows us to classify attacks into two categories:

Cross Site Scripting (XSS) In this type of attack, one web page is able to gain access to another web page's data (such as any cookies it sets). This type of attack might allow a malicious programmer to have access to a user's bank account.

Privilege Escalation Attack In this type of attack, one web page is able to perform actions reserved for the browser *chrome* (user interface). This attack is extremely dangerous, as the web page could read or write files, snoop on other sites, set browser preferences or any number of other attacks.

Mozilla Firefox has a security mechanism in place that stops most attacks. Security in Firefox follows a capabilities based model. This means that property accesses are prefixed with a security check. However, due to performance problems and simple bugs in the code, some of these checks are omitted. While fixing attacks one at a time in a whack-a-mole fashion works to protect users, a more systematic solution is desired. Ideally we would like the security system to fail-safe. However, this can only be guaranteed if developers are not required to manually insert security checks.

The checks themselves are straightforward property-access checks. When a check is performed, the security manager consults the current JavaScript call stack. It then walks down the stack to find the currently executing code. It then finds the principals of the object being checked. Principals are tokens that describe the origin of an object and the related privileges. Object principals are based on the object's lexical scope – and are therefore inherited from its constructor's principals. Having this information, the security manager compares the two principals for equality (or, based on the object, less strict checks, such as ensuring that both principals come from the same origin).

3 Method

We take a systematic approach to solving the problem. We believe that the major reason that the Mozilla security model does not work in all cases is because security checks are neglected in some cases. To address this, we enforce that checks are performed by adding them to the generated bytecode when the JavaScript code is compiled.

Our research falls into three general phases. In the first phase, we modify the interpreter to perform security checks on all property accesses. For the second phase, we create a "safe" version of each of a subset of the modified bytecodes which does not perform the check. Then we modify the bytecode compiler to emit these instructions instead of the unsafe versions in cases where we can prove that the check will pass. In the final phase, we implement a caching mechanism to avoid duplicate checks when the static optimizer can not guarantee the check will pass at compilation time. This helps to optimize around function calls and complicated control flow structures.

3.1 Phase 1: Inserting Security Checks

The first phase of the research requires us to add security checks to all data accessing byte codes. To do this, we modify the interpreter's implementation of the following bytecodes to perform a security check on the appropriate object.

- **name** - the opcode for looking up the value of a given variable.
- **getprop** - the opcode for looking up the value of a property on an object.
- **getxprop** - the opcode for looking up a property when the existence of that property is required.
- **getelem** - the opcode for indexing into an array.
- **getxelem** - the opcode for indexing into an array, when the existence of the property is required.
- **setname** - the opcode for setting the value of a given variable.
- **setprop** - the opcode for setting the value of a property on an object
- **setelem** - the opcode for setting the value of an object in an array.
- **importall** - the opcode for importing all properties from an object.
- **importelem** - the opcode for importing a specific property from an object.

Adding the security checks requires very few, highly localized changes to the interpreter function. These particular changes required changing roughly 33 lines. Most of these changes were simple modifications, primarily adding flags for whether accesses should be performed, either reading or writing. Additionally we made a small change to a macro to actually perform the appropriate security check.

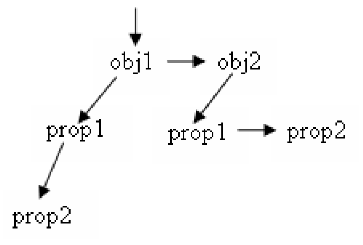

Figure 1: The state of the optimizer is stored in a stack of trees. Each node has points to its sibling and child nodes, as well as the node it was cloned from (not shown).

3.2 Phase 2: Static Optimization

During the second phase of the research we look at the bytecode compiler to implement static optimization of security checks. The dynamic nature of JavaScript makes it very difficult to optimize in general. Low abstraction barriers, such as modification of a callers stack variables without them being explicitly passed and the `eval` construct, which enables very diverse programming paradigms to be used, make many general static optimization techniques, such as constant propagation and useless code elimination, impossible. Therefore, we focus our efforts purely on optimizing the security checks.

3.2.1 Optimizer State

In order to optimize, we keep a tree of the values which have been checked and the access permissions which they have been checked for. Figure 1 shows an instance of the optimization tree. In order to handle nested lexical scopes and code that may or may not be reached, we maintain a stack of these structures for each level. Additionally, each node in the tree contains a clone pointer which points to the element in the previous frame which it was constructed from, or null if the object was newly created for this frame. Insertion operations occur at the top of the stack and have no effect on the rest of the frames in the stack. All removal operations occur at the top of the stack and are propagated back down all entries in the stack by following the clone pointers. Clearing the stack is done by removing the root elements and removes all entries from all frames in the stack.

3.2.2 Optimization Rules

Sequences of statements provide the greatest chance for optimization. Variable and property lookups are the easiest syntactic constructs to handle. Name accesses of local stack variables do not need to be checked at all. Variable lookups and property accesses must be checked only the first time that they are read. When a variable or property is assigned to, properties under it must be checked again, but the property itself does not need to be. Figure 2 shows an example of an optimization.

Calls to other functions might disturb global variables that the current function is using, no guarantees can be made about the state of these variables after a function call. Therefore, after a function call, all non-local values must be checked. This is unfortunate due to the relative frequency of function calls in most JavaScript programs, but this can not be avoided. We implement this by destroying all values in the optimization tree for all blocks.

Conditional expressions can be optimized by optimizing the test condition and then optimizing each of the consequence and alternative branches and computing a greatest common subset to use after the branch. We approximate this by pushing a frame on the optimization tree for the consequence clause and running the optimization. When finished with the consequence, we pop the frame and push a new one for the alternative clause. This technique fails to take advantage of all

```
function f(x) {
  var a;
  a.b;
  a.b.c;
  a.b = x;
  a.b.c;
}
```

```
00000:  getvar 0
00003:  pop
00004:  getvar 0          Look up 'a'
00007:  getprop "b"       We haven't looked 'b' up
                          yet
00010:  pop
00011:  getvar 0          Look up 'a'
00014:  safegetprop "b"   We looked 'b' up before
00017:  getprop "c"       We haven't looked 'c' up
                          yet
00020:  pop
00021:  getvar 0          Look up 'a'
00024:  getarg 0          Look up the new value
00027:  setprop "b"       Now set 'b', 'c' is no
                          longer valid
00030:  pop
00031:  getvar 0          Look up 'a'
00034:  safegetprop "b"   We know that we have access
                          to 'b'
00037:  getprop "c"       But we don't know about 'c'
00040:  pop
00041:  stop
```

Figure 2: Straight line code is optimized by remembering the previously checked entities and emitting a safe opcode when accessing an objects a second time. After assignment, property accessing on the assigned object must be removed from the "checked" list.

possible optimizations, but ensures that all optimizations performed are correct. Figure 3 gives and example of optimization of a conditional structure.

Looping expressions are slightly more complicated than conditionals because they produce a cycle in the control-flow graph. A good optimization for loops is to unroll the first iteration of the loop and convert it to an `if` followed by a `while` loop. This modification, was difficult to implement under the current infrastructure given our time constraints. Therefore, we take a less aggressive approach and simply push a new, blank optimization frame to use when compiling the loop body. Figure 4 gives an example of the optimization which we perform for loop structures.

All optimization is based on a single root which represents the head of the current scope chain. The `with` construct in JavaScript allows the programmer to add another entity in front of the current scope chain. This requires us to be very careful inside of the block which the `with` contains. Therefore, we treat `with` as a function call and clear the tree. The contents of the `with` block are optimized alone in the blank context, and the frame is popped at the end of the block to reveal a blank context.

The `try-catch` construct provides exception handling facilities for JavaScript. Optimizing property accesses is handled in a method similar to conditional expressions. An frame is pushed on the stack for the `try` block. Then it is popped and a new one pushed for `catch` blocks. `finally` blocks are optimized in the context of base as if they did not occur in any special construct.

3.2.3 Problems with Optimization

The problems with static optimization are apparent when large parts of the tree must be discarded. As was previously noted, we can not optimize around function calls because we have no way of

```
function f() {
  if (x.y) {
    x = 3;
  } else {
    x.y;
  }
  x.y;
}
```

```
00000:  name "x"          We haven't seen 'x'
00003:  getprop "y"       Nor 'y'
00006:  ifeq 22 (16)
00009:  bindname "x"      In the if
00012:  uint16 3
00015:  setname "x"       We haven't set 'x'
00018:  pop
00019:  goto 29 (10)      In the else
00022:  safename "x"      We have seen 'x'
00025:  getprop "y"       And we look up 'y'
00028:  pop
00029:  safename "x"      We saw 'x' in the condition
00032:  getprop "y"       We might not have seen 'y'
00035:  pop
00036:  stop
```

Figure 3: When we first use x and its property y, we have to security check them both. In the else clause, we use x.y, however, when the code paths converge at the end of the if, we must be conservative and security check the final property access.

determining what a function call will do. New code could be downloading as scripts are executing which makes whole program optimization impossible, not just infeasible.

There is also a problem of what invokes a function call. In more traditional languages, function calls are always denoted as such in the concrete syntax of the language (e.g., with parentheses). Due to the inclusion of getters, setters, and watch-points in JavaScript, any property access can potentially be a function call. This makes even the optimizations that we performed too aggressive in reality. Figure 5 shows an example of such a problem. This problem can be solved if we can know when a getter, setter, or watch-point is executed because we can begin ignoring the compilation notes and start performing aggressive security checks. We were unable to implement this due to interface through which getters are handled, but we believe that it is a tractable problem. Due to the fact that Mozilla is currently the only browser which supports getters, setters and watch points, their prevalence on the web is extremely low. Therefore, this approach still constitutes an optimization on the majority of web sites.

3.3 Optimization using Caching

In many places, there is likely to be be insufficient information to optimize away security checks entirely. However, since some of these checks will be duplicated, we will implement a caching mechanism to avoid expensive checks. The idea of a cache is that, even though we might not be able to determine the final property being accessed at compile time, we always know it at runtime. Therefore, by pushing decisions about whether or not to run a security check to runtime, we can more accurately decide when we do not have to repeat a security check.

In order for this caching to work, the optimization cannot be based on property names. a caching optimization based on property names would fall prey to the same problems described in the section on static optimization. Instead, the cache can be keyed off of the address of the object being returned. This means that if we have some object o and property a and b refer to the same object, then when the code first accesses $o.a$ a security check will be performed to ensure that the code has access to the object stored there. However, when the code then accesses $o.b$, we already know that access to that object is allowed, so the cost is reduced to a single lookup in a hash table.

It is important to note that this optimization works, even when getters, setters, and watchpoints

```
function f() {
  i = 0;
  while (i++ < 100)
    x[i] = i;
  return x;
}
```

```
00000:  bindname "i"       Initialize 'i'
00003:  zero
00004:  pop
00005:  nameinc "i"        Loop condition
00008:  uint16 100
00011:  lt
00012:  ifeq 29 (17)       In the loop
00015:  name "x"           We have not seen 'x'
00018:  name "i"           We might not have seen 'i'
00021:  safename "i"       We definitely have here
00024:  setelem            Set x[i]
00025:  pop
00026:  goto 5 (-21)
00029:  name "x"           We might not have entered
                           the loop
00032:  return             Return 'x'
00033:  stop
```

Figure 4: Because the top of the loop block has two places that control flow can come from, we must use the weakest subtree. In order to avoid unrolling the loop—since this would break some websites—we optimize the contents of the loop by pushing a new frame and invalidating all of the entries.

are involved. Even if a property access ends up calling a getter, all of that getter's operations are necessarily going to be security checked. Therefore, the getter itself cannot return an object that the code does not have access to.

Another important property of this optimization is that it easily copes with destructive assignments. Because the property check on reads is based on the object that currently resides in the given slot, when code writes to that stack slot subsequent checks will simply see the new value there and optimize appropriately.

3.3.1 Implementation of Caching

The hash table that we used was one that the engine already provided. It is a simple hash table that uses buckets for conflict resolution. The keys into our hash tables were the objects that we were looking up, so we used the object addresses as the keys. Using these keys gave about 50% hit/miss ratio over the course of Firefox startup.

Originally, when we envisioned this optimization, we thought that we would have a global hash table. However, because of strict multi-threading guarantees that SpiderMonkey makes, we were unable to use a mutating hash table. Also, using a global hash table would have meant that we would have to keep track of what the principals of the current stack frame were. Our solution was to instead make a new hash table for each stack frame. If two stack frames shared the same principals, then we would share the hash table between them. This approach had the advantage that we did not have to worry about either of the two solutions.

The last advantage that this had is that stack frames are relatively short lived. Because of this fact, we did not invest too heavily in deciding when to evict cache entries. We were also able to make this design decision because we were only using 8 bytes per entry, which, on a system with 1GB of *RAM* is not a lot per entry. However, one optimization that we did make was to evict entries when objects were garbage collected.

```
var obj = {_c : 0};
obj.__defineGetter__('c',
                     function () {
                       return _c++ == 0
                              ? window
                              : frame.contentWindow;
                     });
obj.c.location;
obj.c.location;
```

Figure 5: We create an object with a private field '_c' that keeps track of how many times the c property of the object has been obtained. The object then defines a getter for the 'c' property that returns a safe window on the first call, and a dangerous window on every other call. Our code incorrectly optimizes the second obj.c.location to not do a security check when the dangerous window is returned, leading to a potential security hole.

3.3.2 Problems with Caching

The only problem that we were unable to solve about this approach dealt with properties that are marked as "allAccess." When the current code comes across these properties, it remembers that the access check succeeded. However, allAccess properties are based on the accessor's class and what property name is being accessed. This means if a site sets a random property to be an object that is allAccess through one path, and another site (from another origin) accesses the allAccess property, first through the "correct" path, and then through the other path, our patch will allow the second access, even though it should be denied. We could fix this by treating allAccess properties specially and not caching those results in our hash table, however, we decided not to implement that within our time constraints.

4 Results

4.1 In Practice

We have implemented all of the topics described here. The patch can be found online [1]. This patch does not introduce any known new regressions, though we have not tested the browser extremely heavily. We have tested two known security bugs in current Firefox builds:

`https://bugzilla.mozilla.org/show_bug.cgi?id=355214` Proxy Auto Configuration is a method by which the browser downloads an external JavaScript file to dictate how to connect to the Internet. This bug exploits a problem by which the configuration script can access an object from outside of the sandbox that it runs in and can use it to run code with elevated privileges. This bug is FIXED by our changes, as we are able to intercept any uses of the leaked object and throw appropriate security errors. This is, in essence a breach of the capabilities-based security model; however, as our fix tightens the model to check for security violations even once the model has been breached, we expect to fix other potential bugs in the same way.

`https://bugzilla.mozilla.org/show_bug.cgi?id=344495` When a site attempts to use the *top.location* property to break out of a frame, it is possible for the top frame to trick the browser into running code in the first frame's security context. While testing this bug is currently blocked by the problem with *location.href*. While our fixes do FIX this bug, we are not entirely sure why.

[1] `http://sys.cs.rice.edu/course/comp527/FirefoxSecurityUpgrade?action=AttachFile&do=get&target=final-patch.txt`

We expect that we have also fixed other, as yet undiscovered, bugs. Our fix is an example of "defense in depth" where we fix a class of bugs with one change. In particular, bugs that rely on obtaining a reference to an object from the wrong scope are now out of luck, as we will security check all of their uses of their illegally obtained objects and throw security exceptions.

4.2 Performance Results

The importance that the Firefox development team places on speed pervades this study. It is the motivation for both the static optimization and the caching. The final performance results are both good and bad.

```
function LoopTest() {
  var a = {b: {}, c: 5};
  for(var i = 0; i < 100000; i++) {
    a.b; a.b;
    a.c; a.c;
  }
}
```

Figure 6: The loop test is the test with the most pronounced difference between the original and the optimized version with the checks. We believe this to be due to the overhead of the function call in an otherwise highly optimized interpreter.

The additional security checks do make Firefox slower, but the result is only statistically significant on the loop test whose code is given in Figure 6. This test represents a tight loop with several, property accesses. These tests likely do exist on some pages, but we don't believe that they pervade the Internet to the degree that it would cause a major problem.

The DHTML test suite which Mozilla uses as part of the benchmarks for Firefox show considerably more promising results. These tests measure the speed of Firefox on common DHTML operations such as swapping images and moving and resizing text. The results are summarized in Figure 7. Out of all 17 tests, only three showed any statistically significant variation between the original program without any security checks and the final with both static optimization and caching. In two of these cases, the difference was less than 3ms. The third test is the `layers2` test which actually involves a tight loop which changes the clipping of 200 DOM elements. None of our optimizations address this because each DOM element is unique. It is possible to perform additional optimization with DOM elements, but the current implementation makes checking if an object is a DOM element very expensive.

5 Alternative Methods

While determining how best to address the problem, we considered two alternatives to the above solution.

The first design is based on the operating system hardware abstraction. The design concept calls for constructing an isolated JavaScript heaps for each web page. In order to allow web pages to interact with sub-frames, and content in windows which they opened, a small bridging interface would be implemented. It is unclear of the performance ramifications of this design. Accessing content in other webpages would require multiple object dereferencing operations, one to lookup the object in the other heap, and the other to actually perform the fetch of the property. Additionally, values passed to cross-heap function calls would have to be boxed into the callee's heap and values returned

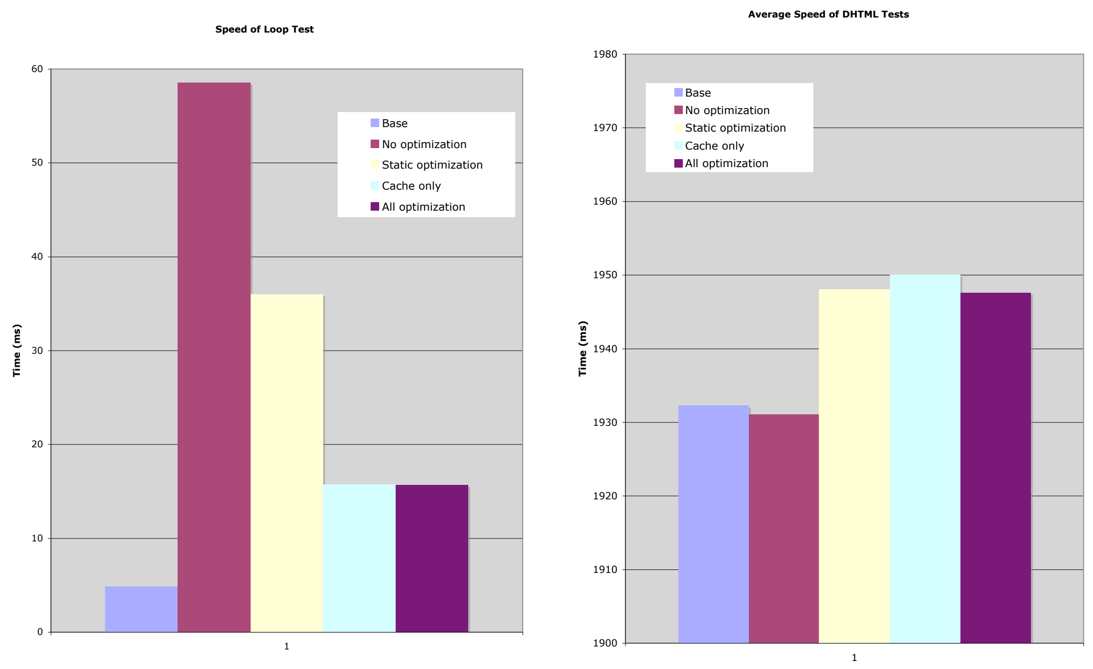

Figure 7: While the loop test is statistically significant, most of the DHTML test are not due to variation in layout time.

would have to be boxed into the caller's heap. Although this would likely constitute considerable overhead on cross-heap functions, intra-heap accesses would never require security checks.

Ultimately, we determined that such a design was not practical to implement in the limited amount of time that we had. Additionally, such a design would require a major reorganization of the JavaScript engine, which goes against the spirit of the project.

The second design is derived from the ACL model which is popular in many file systems. The idea would be to maintain a token in each window which would determine what operations a function could execute on any given object. In order to guarantee anything about the system, these checks would have to be embedded into all object and property access operations which would likely constitute a considerable performance penalty which would be much more difficult to optimize away.

This design was rejected for the same reasons that the previous design was rejected for. Namely, the translation of the capabilities based model to an ACL model would require a considerable amount of re-architecture and the implementation would not be feasible within the short period of time given.

6 Conclusions

This study shows that aggressive security checks improve the security of the Firefox codebase. The bugs that our checks fix are complicated hacks in which privileged functions are tricked into calling unprivileged functions with privileged parameters. Although there is a trade-off between the improved security and the execution speed of JavaScript, it seems safer to err on the side of security than focusing too much on speed. Although we fix numerous bugs, our fix is not a panacea for the security problems. Problems with the method for discovering the principals of the current script, especially when getters and setters are executed, can still be a problem.

The results suggest that though optimization can mitigate the performance penalty paid by the additional security checks, it is unable to completely remove them. The benefits of static optimization were significant for the loop test, but it failed to yield much benefit for the DHTML results. It is possible that implementing more aggressive compile-time optimization could benefit the performance, the fact that compile time is counted against speed makes more complex procedures less feasible. Caching seems like the more feasible alternative. Essentially, it focuses on trying to decrease the cost of security checks in general. This suggests that the separate heap solution proposed earlier could yield considerable benefits since only operations on external objects are penalized and most operations seem to be internal to the page.

This study sheds light on programming language security in general. The bugs that were fixed involve overriding conventions which are relied on by the ECMAScript specification, or using "magic" methods such as `call` and `eval`. We believe that a more static language would itself solve many of the issues that arise in the area of security. Removing strange calling conventions, such as on the `call` method which uses its first parameter as the `this` object would allow more uniform treatment of functions which would reduce both the complexity of the code and the interface which needs to be secured.