2010-07-23

Using Lightweight Formal Methods for JavaScript Security

Using Lightweight Formal Methods for JavaScript Security

May 3, 2010

Mark Reynolds

Summary. The goal of this investigation was to apply lightweight formal methods to the study of the security of the JavaScript language. Previous work [1] has shown that lightweight formal methods present a new approach to the study of security in the context of the Java Virtual Machine (JVM). Since there is a (somewhat) formal specification for the JVM, but there is nothing approaching a formal specification for JavaScript security, the current work has attempted to codify best current practices in the form of a security model for JavaScript. Such a model is a necessary component in analyzing browser actions for vulnerabilities, but it is not sufficient. It is also necessary to capture actual browser event traces and incorporate these into the model. The combination of the static model template representing the security rules, together with the state data derived from the browser actions, creates a complete model that can be analyzed. The work described herein has demonstrated that it is (a) possible to construct a model for JavaScript security that captures important properties of current best practices within browsers; and (b) that an event translator can be written that captures the dynamic properties of browser site traversal in such a way that model analysis is possible, and will yield important information about the satisfaction or refutation of the static rules. This paper will first describe the nature of the problem, namely the scope of security vulnerabilities when using JavaScript. It will then describe the security model and event translator, and present the results obtained for actual analysis of real-world websites. Finally, a description of future work will be given.

Background. The JavaScript language is a website scripting language that has achieved extremely high penetration on the Internet. Unfortunately, this popularity, combined with some intrinsic aspects of JavaScript that will be discussed below, have made JavaScript a particularly enticing target for malware authors. According to some statistics [2], JavaScript-based web attacks, particularly Cross Site Scripting (XSS) and Cross Site Request Forgery (CSRF), account for the majority of all web-based malware. Statistical data on malicious incidents in the past four years [3] indicate that XSS alone may account for as much as seventy percent of all malware. Malicious JavaScript can do damage in and of itself, or it can act as a delivery system for a malicious payload that is executed on the client's (victim's) machine. JavaScript is a purely interpreted language. Although it is named after Java, and bears some syntactic resemblance to Java, JavaScript is in fact quite distant from Java at the semantic level. The Java programming language has a published specification for the language [4] and also for the virtual machine on which Java class files actual run [5]. Thus, Java can be said to have a formal security model as expressed by the constraints that a conforming Java class file must satisfy to be executable. The same cannot be said for JavaScript. JavaScript has a loose collection of rules that conforming browsers should obey, but these rules are not standardized, and variations in the strictness of different implementations can be readily observed by viewing the same website in different browsers. Part of the work described in this report, therefore, has involved the creation of a formal (but currently partial) security model for JavaScript. The goal in creating this model has been to capture best current practices and

create a model that is sufficiently restrictive that aberrant behavior associated with malware can be identified, while still permitting legitimate website traversals to be properly executed. As with the prior Java work, this JavaScript security model has been expressed in terms of constraints, which have then been encoded as an Alloy model template. When a properly instrumented browser visits a website, event data is collected, and a translator run over the event log. The output of the translator is a set of Alloy relation initializers. When these initializers are combined with the Alloy model template that expresses the security constraints, a complete Alloy model is obtained. The Alloy analyzer can then be run against the model to look for counterexamples. Counterexamples are constraint violations, and constraint violations are potential vulnerabilities. One goal in developing the constraint model is to reduce or eliminate false positives (a counterexample appearing when a non-malicious website is traversed) and also false negatives (a counterexample is not found in the presence of malicious behavior). To some extent, this has proven to be a heuristic and iterative process; the evolution of the constraints studied will be described below. Developing a complete security model for JavaScript is a large task, so the focus of the current work has been to demonstrate the conceptual soundness of the approach on a small but meaningful set of constraints. The goal of the current work has been to show that non-trivial XSS attacks associated with the violation of the "same origin policy" can be detected. It has also been a goal to provide an extensible model and translator framework, so that new constraints and new event types can be handled as the work matures.

Prior work on JavaScript security (and web-facing application security in general) has primarily focused on classification-based approaches [6, 7]. Such approaches attempt to classify attack vectors, and then define policies which a "safe" script must obey, such as the same origin policy [8]. The same origin states, informally, that a safe script may only read properties and documents that have the same origin (host) as the originating document. (A more comprehensive definition of the same origin policy will be given below when the model template is described.) At the present time the number of such JavaScript policies is very small, and their development to date has been ad-hoc in nature. A formal specification for the document object model (DOM) of a webpage does exist [9], but this specification does not fully address the dynamics of client-server interaction, and is not a constraint-based security specification. There does not seem to be prior work on the application of formal methods to JavaScript security.

Security Model. In order to understand the proposed JavaScript security model, it is first necessary to briefly review the constraints being modeled, and also consider the ways in which malicious code can bypass those constraints. The primary JavaScript security policy is the "same origin policy". When a web page is loaded into a browser it may contain embedded JavaScript, typically delimited by <SCRIPT></SCRIPT> blocks within the HTML. Any script that occurs in this HTML context is known as a top level script. All top level scripts will have been loaded from a particular web site, namely the one associated with the containing HTML. Thus, top level scripts are associated with a unique origin. Top level scripts can cause other, subsidiary objects, including other scripts, to be loaded. In fact, as will be described below, there are several different ways in which a top level script can instantiate a subordinate script. The same origin policy

states that subordinate scripts (and other subordinate objects, such as graphics, Flash objects, etc) should only be loaded from the "same" origin as the origin of the top level scripts. A naïve interpretation of this policy would assert that all the origins for all scripts should match exactly when expressed as IP addresses. In fact, a simple examination of network traffic between a browser and almost any script-enabled website shows that this interpretation is much too narrow. In order to capture best current practices, it is necessary to extend this policy to allow not only a tree of IP addresses (such as all the hosts within a subnet), but to allow several such trees, that is, a forest of IP addresses. When actual model and the live data are presented, below, this will be made more explicit.

What is the purpose of the same origin policy? Specifically, what is the risk to the user if the same origin policy is not properly enforced? The risk occurs due to the nature of HTML documents. HTML documents have a flat structure, in which control elements, formatting and content are intermixed. When data is provided by a web client, typically in an HTML form or query, that data can be used by server-side scripts to dynamically generated HTML containing the response. This type of "reflection" can create a vulnerability if the input is not properly sanitized, e.g. if HTML control characters are presented as part of the input. The malware author can make use of this failure to properly handle input to create seemingly innocuous links that actually point to unrelated websites. They can cause the user to navigate to one of these malicious links automatically, without any human intervention, because the attacker has placed code that executes within the security context of the browser, and so is trusted. If the same origin policy is properly enforced, however, such malicious redirection should be blocked, because active content (scripts, for example) that come from outside the origin of the top level script will not be loaded or executed. Thus, even if input sanitization is not handled properly, the attacker will not be able install code that autonomously redirects the user. For a more complete description of cross site scripting attacks and the ways that they can leverage weaknesses in the same origin policy, see [7].

A JavaScript script can arrange to load other scripts in several different ways. In particular, the host address part for the script can be specified by a fully qualified domain name (FQDN), by a relative path, by an explicit IP address, or even by an IP address specified as a 32-bit number. (For the purposes of the current work, only IPv4 addresses are considered.) At the implementation level name resolution must take place and ultimately remote content fetched from one or more particular IP addresses. One might therefore model the set of IP addresses as a tree, for example, in which the root node corresponds to the top level of the remote host hierarchy (e.g. cnn.com), and subordinate nodes correspond to children of the root (e.g. images.cnn.com). Because name resolution of a FQDN may produce several answers, however, a tree structure is not general enough to encompass many real world situations. For example, looking up "cnn.com" produces six DNS answers, all within the same /16 subnet but apparently forming three constituent /24 subnets. For these reasons it was decided to model an organizational unit representing a remote (possibly virtual) host as a forest. A forest is modeled as a set of trees. A tree, in turn, is modeled as a collection of nodes, of two disjoint types: an address node and a host node. An address node is a collection of subordinate address and host nodes, much

like a directory. An address node is never a leaf in the tree. By contrast, a host node is always a leaf in the tree (like a file). The Alloy signatures for these model items are:

```
// ipv4 address

sig ipv4 {
      octet1: Int,
      octet2: Int,
      octet3: Int,
      octet4: Int
}

// nodes

abstract sig Node { }

// addr nodes and host nodes
// addr hode is a non-leaf, host node is a leaf

sig AddrNode extends Node {
      masklen:   Int,
      hostlen:   Int,
      netmask:   ipv4
}

sig HostNode extends Node {
      hostlen2:  Int,
      hostip:    ipv4
}

fact {
    Node = AddrNode + HostNode
}
```

The final fact expresses the assertion that address nodes (directories) are disjoint from host nodes (files), and that any node must be one of the two types. An address node contains the netmask for its (sub)domain, as well as information on the length of the network and host parts. Of course, the model view contains deliberate redundancies. A host node contains a prefix length (host part) and an actual IP address. The signature for a tree element (which is based on the Alloy filesystem model), and a forest element, are as follows:

```
// tree of nodes

sig Tree {
    reachable: set Node,
    root:      AddrNode & reachable,
    parent:    (reachable - root) -> one (AddrNode & reachable),
    contents:  AddrNode -> set Node
}{
   reachable in root.*contents
   parent = ~contents
}
```

```
// forest of trees

sig Forest {
    trees:     set Tree
}
```

The tree signature contains four relations, a reachability relation (which expresses the set of nodes that can be reached by some traversal starting from the root), a root node relation (which must always be a reachable address node, since it is always a collection object even if there is only a single host in the collection), a parent relation (which is a map from a reachable node other than the root to exactly one reachable address node), and a contents relation (which is a map from an address node (collection) to the set of nodes that it contains). The two subsequent facts assert that all reachable nodes are in the reflexive transitive closure of the contents relation applied to the root node [`reachable in root.*contents`], and also that the parent relation is the inverse of the contents relation [`parent = ~contents`].

Given this model, one can then make assertions that should be true and verifiable in a valid realization of this model. Many of these assertions are not security assertions, they are statements about the model as a whole. For example, the Alloy fact

```
fact {
  all a: AddrNode | a.masklen + a.hostlen = 32
}
```

says that for all collection nodes, the lengths of the network part and the host part must sum to 32 bits. A less trivial assertion is the more specific prefix fact, which asserts that subordinate nodes must have more specific prefixes than their parents:

```
fact {
  all h: Node, t: Tree | (h in t.reachable) =>
     gte[h.hostlen, ((t.parent).h).hostlen]
}
```

More assertions of this form must be added in order to complete the “organizational unit” part of the model (for example, facts stating that the network and host parts must be non-negative). In order to add the security facts that can then be checked by the model one must express the statement that a host, given as an IP address, either exactly matches the IP address given in at least one host node, or is contained in at least one address node’s subnet. However, in order to state this type of inclusion predicate, there is a technical challenge that must be addressed. This challenge relates to the way that integers are treated in Alloy. In fact, integers are treated as sets, and thus the range of values that an integer may hold is related to the size of the search space. Thus, if one attempts to validate a model with a qualifier of the form “for 8 int” this implies that integers are limited to 8 bits. As a result, using 32 bit integers is not practical, since the search space become to large. This is one of motivations for expressing an IP address as a set of four octets, rather than as a single 32-bit integer. It is not merely a reflection of the fact that IP addresses are conventionally written in terms of four 8-bit integers; it is a reflection of a practical computations limits for the model. This representation as a set of four octets

leads to additional complications, however. When performing prefix arithmetic it is not necessarily the case that a prefix will always be a multiple of 8. In fact, prefixes such as /12, /18 and /23 are common for subnets. This means that checking whether an IP address is contained within a forest of IP addresses representing the allowed set according to the same origin policy may involve the use of bitwise operators. The current version of Alloy (4.1.10) does support bit shift operators; however, it does not support bitwise AND, OR or NOT operators, which significantly complicates the construction of an inclusion predicate. This issue will be explored more fully in the section on Alloy extensions, below.

When a JavaScript script is loaded as part of a web page, all the functions that are defined as part of that script become part of the namespace of JavaScript functions accessible from within that webpage. However, this is not the only means by which JavaScript functions can become part of the scope of a top level script. In fact, there are three additional methods by which a script can create or load new script functions. First, a script can pull in additional script content by explicitly naming an external script through a "src=" tag. From the standpoint of the model such a sourced script is considered as a child of the script that sources it. Second, a script can use elements of the Document Object Model (DOM) to dynamically write new HTML, which can include new script code. This is typically done using the document.write() function, although other DHTML techniques can also be used. Third, new JavaScript objects can be loaded through a text file that describes those objects in JSON (JavaScript Object Notation). Such objects can change method (function) behavior through redirection. From the standpoint of the Alloy model it is desirable to allow for the possibility of different behaviors for each of the four script sources. While this makes the model more complicated, it does conform to best current practices for web technologies such as AJAX (Asynchronous JavaScript and XML). AJAX applications deliberately attempt to weaken the single origin policy in order to improve data sharing between the data/script components of an AJAX dashboard.

For this reason it was decided to model a JavaScript script object as a container which can contain up to four different types of child script: a pure script (which will necessarily be a top level script defined in the origin HTML), sourced (indirect) scripts, dynamic scripts, and JSON scripts. This still preserves some aspects of a tree structure, but now there is not a single "contents" relation, there are four such relations. We define a script node and its subnodes in Alloy as

```
// script nodes

abstract sig ScriptNode { references: Forest }

sig PureScriptNode, IndirectScriptNode, DynamicScriptNode,
  JSONScriptNode extends ScriptNode
```

and we require that the four nodes types are disjoint:

```
fact {
    ScriptNode = PureScriptNode + IndirectScriptNode +
```

```
          DynamicScriptNode + JSONScriptNode
}
```

The tree of script nodes still has a reachability relation, and it also has a root relation which represents the top level script. We require that the root node be pure, namely that it be present within the top level HTML code of a loaded page. Any one of the four subnodes will have a unique parent, but there will be four content relations based on the subtype. This leads to the following Alloy signature:

```
// tree of script nodes

sig ScriptTree {
    reachable: set ScriptNode,
    root:      PureScriptNode & reachable,
    parent:    (reachable - root) -> one (ScriptNode & reachable),
    purecontents:  ScriptNode -> set PureScriptNode
    indircontents: ScriptNode -> set IndirectScriptNode
    dyncontents:   ScriptNode -> set DynamicScriptNode
    jsoncontents:  ScriptNode -> set JSONScriptNode

}{
   reachable in root.*purecontents + root.*indircontents +
           root.*dyncontents + jsoncontents
}
```

Note that we can no longer assert that the parent relation is the inverse of the content relation, as was the case with the network address part of the model. Note also that for full generality, we use a (network address) forest relation for each type of ScriptNode, rather than imposing some a priori structure in the signature definition.

Based on this model we can now write security assertions that can be verified for top level scripts and their tree of children. To do so, however, requires that we have defined a new Alloy primitive that corresponds to the IP address subset relationship described earlier. Before we talk about how this might be implemented concretely, let us write an example of a checkable assertion, assuming that this new primitive is called “ipin”:

```
pred sameorigin_strict {
  all F: ScriptNode, A: F.root.references | F.reachable.*(…content).references ipin A
}
```

In this notation the ellipsis …content is used to denote the set union of all four content types. In words, this predicate asserts that for all nodes that are reachable from the root script node, by any of the four means, any IP address accessed by one or more of those nodes is contained in the forest of IP addresses that are accessed by the root node. This statement is a restrictive form of the same origin policy in that it does not permit variant behaviors by the four subtypes (and would thus prohibit a great deal of AJAX code). For the immediate purposes of verifying that the model actually captures this more restrictive behavior, we will start with this assertion and build translator code from JavaScript event logs that can verify or refute this assertion concretely.

How can the "ipin" relation be implemented? In Alloy one can reference internal functions exported by Alloy's Java API from Alloy code. Thus, two things will be required to implement this new relation: (1) Java primitives that perform the appropriate address calculations, and (2) an Alloy file that defines the binding between the Alloy-visible namespace of relations and the Java primitives. For example, integer operations are defined in util/integer.als. These functions reference Java exported functions. The extensions needed by Alloy to support the creation of an "ipin" relation are the subject of the next section.

Alloy Extensions. In order to perform the necessary IP address calculations in Alloy, several modifications needed to be made to Alloy itself. The first goal was to implement bitwise AND (bitand) and bitwise OR (bitor) binary operators, and also a bitwise NOT (bitnot) unary operator. These operators were successfully implemented and tested, in a manner that will now be described. Existing integer operations in Alloy are implemented in the file models/util/integer.als. This file defines a set of functions that indirectly call functions exposed by Alloy's Java API. For example, the definition of the rem function is as follows:

```
fun rem [n1, n2 : Int] : Int { n1 fun/rem n2 }
```

Alloy supports file inclusion in a standard manner, where directory names act like namespace qualifiers. The special name "fun", however, is not an actual directory. Any function that is named within the "fun" namespace is actually an Alloy function implemented in Java. Thus it is straightforward to add the three indicated functions to this file as:

```
fun bitand [n1, n2 : Int] : Int { n1 fun/bitand n2 }
fun bitor [n1, n2 : Int] : Int { n1 fun/bitor n2 }
fun bitnot [n1 : Int] : Int { fun/bitnot n1 }
```

In order to provide support for these operations within Alloy, several steps were taken. The Alloy parser is not directly implemented in Java; instead it is implemented using the CUP compiler, which takes a CUP specification as input, and produces parser code (in Java) as output. Thus, the first step was to modify the parser/Alloy.cup file to provide tokens INTBITAND, INTBITOR and INTBITNOT, and then declare that they are exported via the "fun" namespace. The next step was to modify the abstract syntax tree handling code to recognize these tokens. This involved modifying the files ast/Expr.java, ast/ExprBinary.java and ast/ExprUnary.java. Nodes of type "Expr" were created. The final step was to actually implement the operations. Alloy ultimately converts all expressions into a Boolean satisfiability problem and then uses a SAT solver to derive a satisfying instance or counterexample. The conversion between the AST representation and the SAT representation is handled by the translator portion of Alloy. When a node of type ExprUnary or ExprBinary is encountered by the translator, the operator it represents is extracted and then applied to the operand(s). Thus for a REM node the code is:

```
case REM: I = cint(a); return i.modulo(cint(b));
```

where “I” is of type IntExpression, “a” is the left subnode of the node, and “b” is the right subnode of the node. Thus for the two new binary operators the following translations are added:

```
case BITAND: I = cint(a); return i.bitand(cint(b));
case BITOR: I = cint(a); return i.bitor(cint(b));
```

while for the new unary operator the translation becomes:

```
case BITNOT: return cset(x.sub).bitnot(), x);
```

In this latter case “x” is the node, and “x.sub” is its sole subnode (the target of the operator). Once the simple methods bitand(), bitor() and bitnot() were added to the IntExpression and Func classes the work of exporting the corresponding operations was complete. In order to avoid conflicting with Alloy’s built-in handling of the tokens “&”, “|” and “~”, no attempt was made to provide overloaded operators. Thus to invoke any of these three new operations, they must be written as a function call, e.g. “bitand[a, b]” rather than “a bitand b” or “a & b”.

Unfortunately, an additional difficulty arose when attempt to extend these operations to provide the required “ipin” predicate. The difficulty arises due to the way that Alloy represents integers, as has already been described. Given the existing presentation for IP addresses using the “ipv4” signature, shown in the previous section, three approaches presented themselves. First, one could perform all operations by bit-shifting address components that do not fall onto an 8-bit boundary onto an address component that does, performing the corresponding bit-wise AND, OR or NOT, and then shifting back. The second approach would have been to provide an alternate method for handling integers in Alloy that would workaround its limitations on treating them as sets. In this approach, a new internal (Java) data type would be created to represent 32-bit quantities. In this approach an address, address prefix or integer would be bound to an Alloy variable by means of a new unary operator called “intern”. This would eliminate the need to manipulate integers or addresses using the Alloy “Int” data type. With this new operator it would be possible to write Alloy code such as

```
A = intern[128.89.81.219]
B = intern[0xFF FF F8 00]
C = bitand[A, B]
```

In addition to requiring Java code that would expose “intern” through the “fun” namespace, support would be required from the Alloy parser to suppress interpretation of literals within the scope of an intern[] evaluation. Although this second approach would require more changes to the Java implementation of Alloy, it would appear to address the performance issue associated with integer manipulations.

The third approach would be to restrict address calculations to those that did fall on 8-bit boundaries. While this approach only supports a subset of possible real-world data, it had the advantage of being significantly easier to implement than either of the other two approaches. For the purposes of the current work, the third approach was chosen as a compromise of development time versus functionality. The results presented in the data analysis section, below, reflect this choice of implementation. In future extensions of this work, the limitations imposed by this choice will be removed. Using this reduced formulation the "ipin" predicate was defined as

```
pred ipin[F: Forest, H: HostNode]  {
        some A in F.Tree | ipinaddr[H, A.reachable]
}
```

which expresses the idea that for some rooted tree A in the forest of allowable addresses F, the newly fetched address (as represented by H, which is of type HostNode) satisfies the address inclusion predicate "ipinaddr" with respect to at least one node reachable from A. In order to understand the function of the "ipinaddr" predicate, consider an example (again, assuming only octet-based address and prefix alignment). The host address 128.33.0.20 is contained within itself; it is also contained within 128.33.0.0/24, 128.33.0.0/16 and 128.0.0.0/8. Therefore we can express this "contained within relation" as the concatenation of four clauses. For the direct equality case the clause is:

```
H.hostip.octet1 == Ax.netmask.octet1 &&
H.hostip.octet2 == Ax.netmask.octet2 &&
H.hostip.octet3 == Ax.netmask.octet3 &&
H.hostip.octet4 == Ax.netmask.octet4 &&
Ax.masklen == 32
```

(where Ax represents a member of the set A.reachable). For the /24 case the clause is

```
H.hostip.octet1 == Ax.netmask.octet1 &&
H.hostip.octet2 == Ax.netmask.octet2 &&
H.hostip.octet3 == Ax.netmask.octet3 &&
Ax.masklen == 24
```

with similar formulations for the /16 and /8 cases. The "ipinaddr" predicate then can be written as

```
pred ipinaddr[H: HostNode, A: set Node] {
        some Ax in A | (clause1 || clause2 || clause3 || clause4 )
}
```

Browser Event Translator. The Alloy security model described above is only a model template. It contains signatures and relations, but no initializers for those relations. As in the case of the earlier work on Java, live data (JVM bytecodes or website traversals) must be parsed and converted into relation initializers in order to create a complete model that

can be submitted to the Alloy analyzer. Collecting the data needed by this translator required an instrumented browser, which is a web browser that had been specially modified to collect script execution data with sufficient fidelity that it could be used to initialize the two forests: the forest of trees of acceptable IPs, and the forest of trees of traversed IPs. The Firefox browser was chosen because it is open source, supports JavaScript debugging, and is widely used. It is necessary to capture both synchronous function execution and also asynchronous execution of event handlers (such as the onMouseOver() handler and other built-in JavaScript handlers). Fortunately, it is straightforward to do this in Firefox. The default internal call hook, the function _callHook(), was modified to collect function and event trace information. This function is called with the execution context of the browser (a C++ object of type JSContext), and the current stack frame (a C++ object of type JSStackFrame, which contains sufficient information that stack traversal through higher level frames is possible). It was demonstrated that function and event information could be collected (two examples are shown below) that provided sufficient information so that relation initializers could be written to check the stated security constraints.

Consider the JavaScript function:

```
alert("This is an alert");
```

Execution of this function displays a dialog box with the indicated text and an "OK" button. The OK button is automatically bound to an internal callback function which dismisses the dialog box and takes no further action. Thus, in calling this function within the context of another JavaScript function, say "x()", it should be possible to catch both the synchronous execution of "alert", the asynchronous execution of the callback handler, and also to determine that alert() was called from within x(). These goals have been realized by appropriate manipulation of the calling context and call stack objects. The resulting trace (edited for readability) is shown below:

```
_callHook():
        function:       alert
        nargs:          1
        arg0type:       JSString
        arg0:           "This is an alert"

        event:          onClick
        eventType:      Button <class>
        eventText:      OK
        coords:         235,644

        function:       <anonymous>
        nargs:          1
        arg0type:       integer
        arg0:           0
```

Once a website has been displayed by the browser, the event log is given as input to the translator. The event logs have been found to contain five types of data: (a) DNS names for IP addresses; (b) URLs containing DNS names; (c) raw IP addresses; (d) URLs containing raw IP addresses; and (e) numerically encoded IPs. The last category was unexpected, but presents no additional translation problems. For example, a numerically encoded version of 128.33.0.20 would be 2149646356, which is $128*2^{24} + 33*2^{16} + 0*2^{8} + 20$. When the translator encounters a URL, it separates it out the host part from the remaining parts of the URL. No attempt is made to translate embedded URLs that occur in the URL "command line" at this time. Thus, for example, given a complex URL of form http://www.msn.com/?ocid=hmlogout?ref=http://abcnews.com, the host part would be taken to be www.msn.com and the embedded URL "abcnews.com" would (currently) be ignored.

If a DNS lookup is required, the translator performs it locally, and captures all the answers. Thus, if the lookup returns multiple values in the "answer" part of the returned DNS reply, all such values are used in populating the forest of IP addresses. One difficulty in the translation process is that none of the address generated by capturing script information contains IP address prefix information. Only IP addresses are present. Therefore a unification rule must be provided by the translator to infer prefix length from the data gathered. This can subsequently be verified using Autonomous System information for prefixes as published by the various Regional Internet Registries, e.g. published by RIPE at http://www.ripe.net/projects/ris/rawdata.html. (Currently this step is done manually.) The unification rule, as it is currently formulated, takes the most specific prefix from all DNS answers for a given URL. It does not attempt to unify across DNS answers from multiple URLs. Actually data is presented below.

The output of the translator is a set of Alloy initializers (HostNodes and AddrNodes) for the model. Note that at the present time the event log contains a far larger set of information than is actually needed to create the IP address initializers. This will allow future versions of the translator to capture more information for a more extensive set of initializers and constraints.

Website Testing and Data Analysis. The goal of website testing is to verify that the same origin hypothesis is properly checked by the Alloy model and the initializers created by the translator. This has both positive and negative aspects. A conforming website should yield at least one instance (and no counterexamples); a nonconforming website should yield at least one counterexample (and no instances). For the purposes of conformance testing a fairly complex website is desirable, in particular one that loads a number of scripts and has a fairly dense forest of IP addresses, both within its own domain, and also from external domains. For the purposes of nonconformance testing, it is desirable that, at a minimum, it can be demonstrated that the "ipin" constraint is violated for site traversals that violate the same origin policy. To date model conformance has been verified for several conforming sites. Detection of malicious activity is not complete at this time.

As an example, when http://abcnews.com is loaded, a large number of internal and external URLs are generated by the top level HTML code. The top level JavaScript file (based on network packet captures) is en_US/all.js. This script loads more than twenty subsidiary scripts from a large number of URLs. Captured URLs within the abcnews.com domain include:

abcnews.com	199.181.132.250
abcnews.go.com	198.105.195.49
a.abcnews.com	96.6.46.50; 96.6.46.35
	unified to 96.6.46.0/24
ll.static.abc.go.com	208.111.128.6; 208.111.128.7
	unified to 208.111.128.0/24

Outside the abcnews.com domain the following URLs (among others) were seen:

edge.quantserve.com	64.94.107.0/24
www.google-analytics.com	74.125.113.0/24

Several JavaScript script files are loaded from hosts outside any of the abcnews.com IP address blocks. This serves to confirm several of the speculated properties described in previous sections. First, if the same origin policy had been modeled as a rooted tree, rather than a forest of rooted trees, verification would have failed. Second, if only direct loading (via a "src=" script tag) had been modeled, verification also would have failed. In a number of cases the actual JavaScript that was executed was dynamically generated by combining components from a number of different sites (usually as a result of combining static content with advertising content). Thus direct loading, indirect loading and dynamic content creation were observed. However, it should also be noted that for the sites investigated, dynamic creation of content through the instantiation of JSON files was not observed. Most importantly, Alloy found several model instances and no counterexamples for the indicated benign sites.

In conclusion, this work is the first step in the development of a complete formal security model for JavaScript, which will be expressed as a series of lightweight constraints. To date, the same origin policy has been encoded, and has been shown to yield model instances for valid websites. In addition, an instrumentation framework for the Firefox browser has been developed, and a translator created that converts the logged event output from the browser into Alloy initializers in order to create a complete model.

Future Work. Several things need to be done in order to bring the JavaScript system to the same level of completion as the corresponding Java system. All the steps that are manually done during the translation process need to be incorporated into the translator so that they can be done in a fully automated manner. The Alloy constraints need to be generalized so that they can handle arbitrary prefixes, not just those that are multiples of 8. This step may involve more work on Alloy internals, or it may be realized in terms of bitshift arithmetic. Additional work on detecting violations of the same origin policy needs to be performed so that actual detection of malicious activity is possible directly

from the model. Finally, once it is possible to detect XSS, the set of constraints must be enriched with additional rules that will allow other types of malicious activity, such as CSRF, to also be detected.

References.

1. M. Reynolds, "Lightweight Modeling of Java Virtual Machine Security Constraints", in *Abstract State Mcahines, Alloy, B and Z: Second International Conference, ABZ2010*; M. Frappier, ed.; Springer Verlag, 2010.
2. Web Hacking Incidents Database, http://www.webappsec.org/projects/whid.
3. Common Weaknesses Enumeration, http://cwe.mitre.org
4. J. Gosling et al., *Java™ Language Specification, 3rd ed*, Addison Wesley, 2005.
5. T. Lindholm and F. Yellin, *Java™ Virtual Machine Specification, 2nd ed*., Prentice Hall, 1999.
6. D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook*, Wiley Publishing, 2008.
7. J. Grossman et al, *XSS attacks*, Syngress, 2007.
8. D. Flanagan, *JavaScript: The Definitive Guide 5th Edition*, O'Reilly, 2006.
9. http://www.w3.org/DOM.