A Formal Model of Web Security Showing Malicious Cross Origin Requests and Its Mitigation using CORP

Krishna Chaitanya Telikcherla[1], Akash Agrawall[2] and Venkatesh Choppella[2]
[1]*Microsoft India, Hyderabad, India*
[2]*IIIT Hyderabad, Hyderabad, India*

Keywords: Web Browser, Security, Alloy, Modelling, Formal Methods, Cross Origin Request Attacks, Mitigation.

Abstract: This document describes a web security model to analyse cross origin requests and block them using CORP, a browser security policy proposed for mitigating Cross Origin Request Attacks (CORA) such as CSRF, Click-jacking, Web application timing, etc. *CORP* is configured by website administrators and sent as an HTTP response header to the browser. A browser which is CORP-enabled will interpret the policy and enforce it on all cross-origin HTTP requests originating from other tabs of the browser, thus preventing malicious cross-origin requests. In this document we use Alloy, a finite state model finder, to formalize a web security model to analyse malicious cross-origin attacks and verify that *CORP* can be used to mitigate such attacks.

1 INTRODUCTION

Analyzing the security of the web platform is a daunting task, since it is based on several complicated web specifications which are often written and implemented manually. Over the years, researchers have used formal verification to analyze the security of network protocols. Taking this forward, Akhawe et al. (Akhawe et al., 2010) built a formal model of web security based on an abstraction of the web platform. They used Alloy, a finite state model finder, to build their formal model and showed that their model is useful in identifying well-known as well as new vulnerabilities in web specifications. Following this work, several researchers used Alloy to formalize security aspects of the web. They used it to verify the soundness of both existing security aspects as well as newer proposals. Inspired by the widespread application of Alloy in verifying web specifications and architectures, we have used it to formalize and verify the soundness of CORP (Cross Origin Request Policy) (Telikicherla et al., 2014), proposed by Telikicherla et al. in *2014*.

1.1 An Introduction to Alloy

Alloy (Jackson, 2012) is a light weight, declarative modelling language based on first order relational logic. It is used for describing the structural properties of a model. Alloy Analyzer is a constraint solver which automates the analysis of models written in Alloy. Alloy supports two kinds of analysis - simulation and checking. In simulation, the consistency of an invariant or operation is demonstrated by generating an instance (a state or transition). In checking, a consequence of the specification is tested by attempting to generate a counterexample. Alloy analyzer converts the Alloy specification of a model (having simulation and checking queries) into boolean satisfiability problems (SAT) and uses a SAT solver to solve the satisfiability problem within a user-specified scope.

The Alloy specification consists of signatures, fields, facts, predicates, assertions, functions, commands and scope. Signatures describe the entities that are being reasoned and fields define the relation between signatures. Facts are additional constraints on signatures and fields, which always hold true. Predicates are constraints which are required to hold true only when they are simulated. Assertions are constraints which are intended to follow from the facts of a model. If an assertion does not hold true, the analyzer will produce a counterexample. Functions are expressions that can be reused in different contexts. The run command asks the alloy analyzer to generate an instance of the model and the check command asks the analyzer to verify if an assertion holds good. The scope command limits the size of instances considered, to make model finding feasible (the default scope is 3).

Alloy supports the following set operators: Union (+), intersection (&), difference (−), subset (in), equality (=) and the following logical operators: negation (not), conjunction (and, &&), disjunction (or, ||), implication (implies, =>), alternative (else),

Telikicherla, K., Agrawall, A. and Choppella, V.
A Formal Model of Web Security Showing Malicious Cross Origin Requests and Its Mitigation using CORP.
DOI: 10.5220/0006261105160523
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 516-523
ISBN: 978-989-758-209-7

double implication (<=>). The join (.) of two relations is obtained by taking every combination of a tuple from the first relation and a tuple from the second relation, with the condition that the last element of the first tuple matches with the first element of the second tuple, and omitting the matching tuples.

An Alloy model is interpreted as a conjunctive logical formula. Constraints enforced by signatures and facts become a part of the formula. Also, a predicate which is being simulated becomes part of the formula.

1.2 Characterstics of the Proposed Model

1.2.1 Simpler Abstraction

Akhawe's Alloy model captures the abstraction of the web platform and can be used as a baseline for our model. However, it gets complicated when it is extended with DOM (Document Object Model) elements and their relations. Since the problem we are solving is related to cross origin interactions, instead of extending Akhawe's model, we have borrowed the basic signatures and captured only the relevant details, thereby building a simpler abstraction of the web platform.

1.2.2 Non-empty Context

In a general browsing scenario, when a user opens a new browser window, there is no initial context (we refer to this as "Empty context"). In such a context, the user initiates the first HTTP request by typing a URL in the address bar. Once the request gets a successful HTTP response, a document is constructed, which is the state of the browser. Subsequent HTTP requests occur in the context of this document, which we refer to as "Non-empty context". Our model assumes that a non-empty context of attacker's website is available and does not model the HTTP transaction which built this context. This assumption makes the model simpler to analyze.

1.2.3 Focus on Malicious X-Origin Calls

Since our model assumes that a non-empty context of attacker's website is available, it can easily capture malicious cross origin HTTP requests sent to a genuine site. In a typical cross site attack scenario (Cro,), a user logs into a genuine website in one tab of a browser and (unintentionally) opens a malicious website in another tab, which generates malicious cross origin HTTP requests to the genuine site. CSRF (CSR, 2016), Web application timing attacks (Web,), Clickjacking (Cli,), Login detection (Log, 2012) etc., come under this category of attacks, which we combinedly call as Cross Origin Request Attacks (CORA). The instances of our model generated by Alloy show a document loaded from an evil server containing a set of elements (non-empty context), which make cross origin HTTP calls to a genuine server. This aligns with the aforementioned attack scenario.

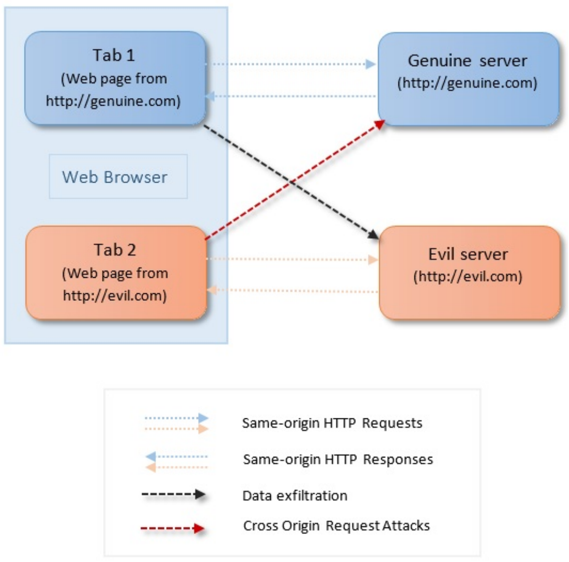


Figure 1: Exfiltration vs Cross Origin Request Attacks.

Note that the model should not be confused with data exfiltration scenario, where a genuine website makes X-Origin calls and sends data to an evil server (e.g., due to an XSS flaw). We mention this word of caution since data exfiltration can be prevented using Content Security Policy (CSP), which is not what we are solving using CORP. Figure 1 depicts the difference between Exfiltration and Cross Origin Request Attacks.

1.2.4 Minimal Scope

To keep the model simple, we have restricted Alloy's instances to one Browser and 2 Servers. This is because, to demonstrate CORA (Cross Origin Request Attack), we need a minimum of two servers (a genuine server and an evil server) and one document loaded from the evil server, which makes HTTP requests to the genuine server. Also, as against the depiction in Figure 1, we are not modelling multiple tabs in the browser. The reason is, CORA is all about a malicious HTTP request originating from an evil webpage, which causes undesirable consequences in the state of a genuine server. So it is sufficient to have a single tab (synonymous to Browser in our model) which loads a document from an evil server.

2 RELATED WORK

Telikicherla et al. proposed a formal model for Cross Origin Request Policy (CORP) (Telikicherla and Choppella, 2013). We have simplified the model further and captured only the elements which are required for initiating cross-origin requests and its mitigation. Akhawe et al. proposed a formal model of web security and used this model to identify the security vulnerabilities (Akhawe et al., 2010). The model explains the details about browser, servers, cookies, HTTP protocol and discusses different scenarios based on various types of potential attackers. We have borrowed the basic signatures and captured the relevant details from it thereby building a simpler abstraction of the web platform.

Ryck et al. modelled CSRF attacks and showed how their proposed method can be used to mitigate the attack using Alloy (De Ryck et al., 2011). Chen et al. proposed App isolation in a single browser to get the security of multiple browser (Chen et al., 2011). They used alloy to model a web browser and verify the proposed method. Cao et al. have used Alloy to model their proposed Configurable Origin Policy (COP) and verify it (Cao et al., 2013). There has also been some work done in validation of web security services by modelling (Gordon and Pucella, 2005; Bhargavan et al., 2006; Clarke et al., 2000; Cremers, 2008; Armando et al., 2005; Carlucci Aiello and Massacci, 2001; Blanchet et al., 2005).

Cross-origin request attacks can be very dangerous such as CSRF (CSR, 2016), login detection (Log, 2012), clickjacking (Cli,), web application timing (Web,). There are certain limitations of current methods in mitigating this class of attacks (Telikicherla et al., 2014). In this paper, we explain a formal model of web security showing malicious cross-origin requests and their mitigation using *CORP*.

3 Pre-CORP AND Post-CORP MODEL

For easier analysis and understanding, we have created two Alloy models: Pre-CORP and Post-CORP. The Pre-CORP model captures the current state of the web platform, where unrestricted cross origin requests are possible. The Post-CORP model is an extension of our Pre-CORP model wherein we add additional signatures, facts and predicates which help in enforcing constraints on cross origin requests.

4 MODELLING CROSS-ORIGIN REQUESTS IN THE WEB PLATFORM (Pre-CORP)

Figure 2 shows the meta model of Pre-CORP model. The core components of this model are: HTTPTransaction, Origin and HTTPEventInitiator. The relations and constraints around these components are explained in this section.

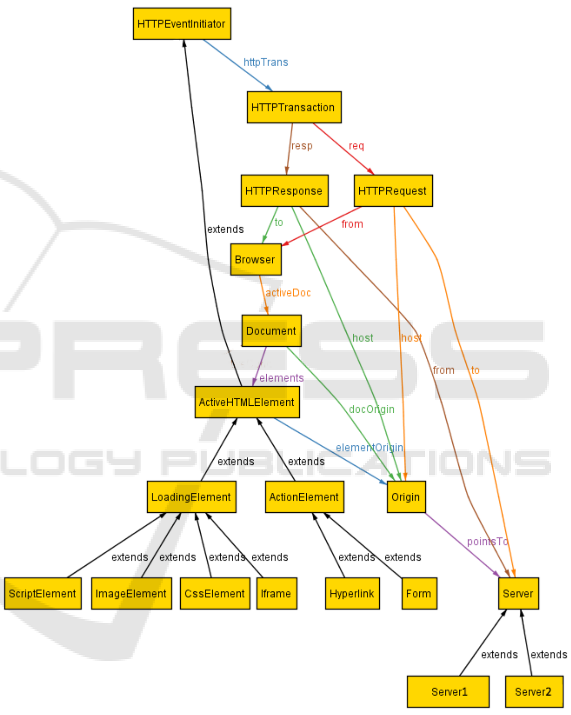


Figure 2: Meta model of Pre-CORP Alloy model.

4.1 HTTP Transactions

The code in Listing 1 shows our abstraction of an HTTP transaction. A HTTPTransaction is a type which consists of exactly one HTTP request and exactly one HTTP response. An HTTP request is initiated from a browser and sent to a server, while an HTTP response is initiated from a server and sent to a browser. Both of them are associated with an origin, HTTP Request is associated with the server to which it is making the request and HTTP Response is associated with the server from where the response is originated.

```
abstract sig HTTPTransaction {
        req : one HTTPRequest ,
        resp : one HTTPResponse
}
sig HTTPRequest {
        from : one Browser ,
        to : one Server ,
        host: Origin
}
sig HTTPResponse {
        from : one Server ,
        to : one Browser ,
        host: Origin
}
```

Listing 1: HTTP Transaction Formalization.

4.2 Origin

The code in Listing 2 shows the basic web model which explains a key concept of the web platform - the origin and its relation with a browser and a server. A browser consists of exactly one active document (i.e., the document which a user interacts with at any point of time). Each document has exactly one origin, which is the origin of the webpage loaded in the browser. Document also contains elements, a set of ActiveHTMLElements, the elements which initiate HTTP Transaction (explained with more clarity in "HTTP Event Initiators" section). Each origin points to exactly one server.

```
sig Origin {
        pointsTo : one Server
}
sig Browser {
        activeDoc : Document
}
abstract sig Server {}
sig Server1, Server2 extends Server {}
sig Document {
        docOrigin : Origin ,
        elements : set ActiveHTMLElement
}
```

Listing 2: Browser Origin Formalization.

4.3 HTTP Event Initiators

The code in Listing 3 shows our classification of components of a web page (document). A HTTPEventInitiator is any component that can trigger an HTTP transaction, and is associated with an origin. ActiveHTMLElement are modelled as HTTP event initiators. We have not included inline JavaScript in our model since cross-origin requests originated from these elements are automatically blocked because of Same Origin Policy (SOP) (Zalewski, Michal, 2011). DOM elements in a document are classified based on their capability to trigger HTTP requests. Those that cannot trigger HTTP calls are called as passive HTML elements (e.g., Div, Span, Textbox etc.), while those that can trigger HTTP calls are called as active HTML elements. For keeping the model concise, only active HTML elements have been included in our specification. ActiveHTMLElements are further classified as LoadingElements - elements that automatically trigger HTTP requests as soon as they get added to the DOM tree (e.g., img, script, iframe etc.) and ActionElements - elements that require an action by humans to trigger HTTP requests (e.g., hyperlinks, forms).

```
abstract sig HTTPEventInitiator {
        httpTrans: lone HTTPTransaction,
        initiatorOrigin: Origin
}
abstract sig ActiveHTMLElement extends
    HTTPEventInitiator {}
abstract sig LoadingElement, ActionElement
    extends ActiveHTMLElement {}
sig ScriptElement, ImageElement, CssElement,
     Iframe extends LoadingElement {}
sig Hyperlink, Form extends ActionElement {}
```

Listing 3: Element Formalization.

4.4 Fact: Transaction Rules

Listing 4 shows the rules which ensure that the model is sane with respect to HTTP transactions which happen on the web. Below is the description of the fact.
For all instances of HTTPTransaction, Browser and Server, the following rules should hold:

Line 3: Request and response must belong to the same origin.
Line 4: An HTTP request's hostname and its destination server's origin must be the same.
Line 5: If a request is sent to a server, response should be received from the same server.
Line 6: If a request is sent from a browser, response should be received by the same browser.

For any two disjoint HTTPTransactions *t1, t2*, the following rules should hold good:

Line 10: Two transactions should not interfere with each other's request.
Line 11: Two transactions should not interfere with each other's response.

Line 13: All HTTPTransactions should be due to some Active HTML element.

```
fact TransactionRules {
        all t:HTTPTransaction, b:Browser, s:
            Server | {
                t.req.host = t.resp.host
                t.req.host = t.req.to.~
                    pointsTo
                s=t.req.to => s = t.resp.
                    from
                b = t.req.from => b = t.resp
                    .to
                t in HTTPEventInitiator.
                    httpTrans
        }
        all disj t1, t2: HTTPTransaction | {
                no ( t1.req & t2.req )
                no ( t1.resp & t2.resp )
        }
        HTTPTransaction in ActiveHTMLElement
            .httpTrans
}
```

Listing 4: Fact: Transaction Rules.

4.5 Fact: NoOrphanElement

Listing 5 shows that every element must have a parent attached with it (except the root element, i.e. HTTPEventInitiator).

```
fact NoOrphanElement {
        no (ActiveHTMLElement - Document.
            elements)
        no (HTTPTransaction -
            HTTPEventInitiator.httpTrans)
        no (HTTPRequest - HTTPTransaction.
            req)
        no (HTTPResponse - HTTPTransaction.
            resp)
        no (Document - Browser.activeDoc)
        no (Server - Origin.pointsTo)
        no Origin - (HTTPRequest.host +
            HTTPResponse.host+Document.
            docOrigin)
}
```

Listing 5: Fact: No Orphan Element.

4.6 Fact: EventInitiatorsInheritParentOrigin

Listing 6 shows that for all instances of HTMLElement, the initiator's parent document's origin must be the same as the element's origin.

```
fact EventInitiatorsInheritParentOrigin{
        all elem:ActiveHTMLElement | elem.
            initiatorOrigin = elem.~elements
            .docOrigin
}
```

Listing 6: Fact: Event Initiators Inherit Parent Origin.

4.7 Fact: Disjointness

Listing 7 ensures that no two disjoint elements interfere with each other's operations.

```
fact Disjointness {
        all disj b1, b2 : Browser | {
                no ( b1.activeDoc & b2.
                    activeDoc )
                no ( b1.activeDoc.docOrigin
                    & b2.activeDoc.docOrigin
                     )
        }
        all disj o1, o2 : Origin | no ( o1.
            pointsTo & o2.pointsTo )
        all disj e1, e2 : ActiveHTMLElement
            | no ( e1.httpTrans & e2.
            httpTrans )
}
```

Listing 7: Fact: Disjointness.

4.8 Predicate: SameOriginTransaction

Listing 8 shows instances of same origin transactions, which are safe. There can be only one server involved since this is a same origin transaction.

```
pred SameOriginTransaction {
        one HTTPTransaction
        one Server
}
```

Listing 8: Predicate: Same Origin Transaction.

4.9 Predicate: CrossOriginTransaction

Listing 9 shows instances of cross origin transactions. There should be 2 servers involved since this a cross origin transaction, one of the servers to which the HTTPEventInitiator belongs and the other server to which the request is made.

```
pred CrossOriginTransaction {
        some t:HTTPTransaction | {
                t.req.from.activeDoc.
                    docOrigin.pointsTo =
                    Server1
                t.req.to = Server2
        }
}
```

Listing 9: Predicate: Cross Origin Transaction.

5 MODELLING RESTRICTIONS INTRODUCED IN CORP (Post-CORP)

We have extended our Pre-CORP model (which depicts unrestricted cross origin HTTP requests) with new signatures and enforcement rules in the form of facts and predicates. The resultant model assists web administrators in configuring certain permissions, which limit Cross Origin Request Attacks (CORA) to a great extent. Figure 3 shows meta model of Post-CORP model.

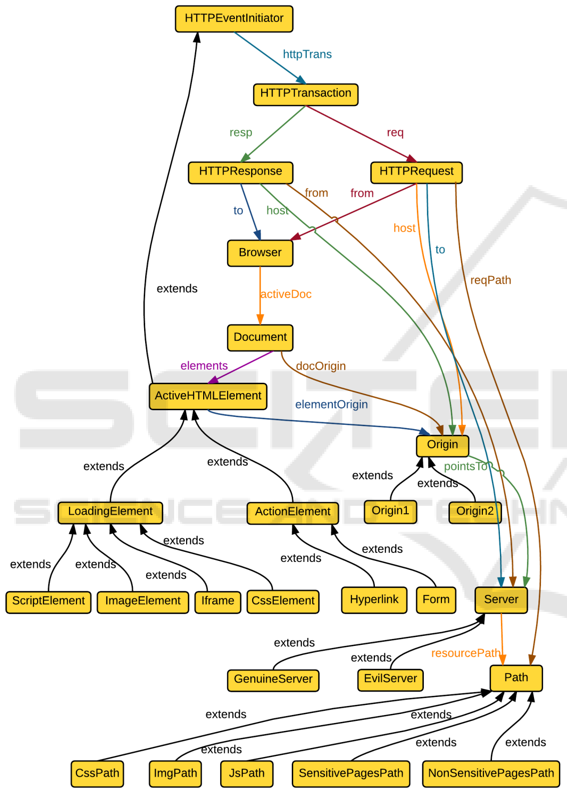


Figure 3: Meta model of Post-CORP Alloy model.

5.1 Key Idea of CORP

The key idea of CORP is to preserve the semantics of HTTP transactions on the web, which help in defending against CORA. Instead of looking at the root cause of CORA as “Confused Deputy Problem”, we borrow concepts from programming languages theory and look at root of these attacks as “Type checking problem”. We draw an analogy between HTTP requests and typing of programming languages, which is the base for our proposal. We observe that HTTP transactions in the current web model are analogous to dynamic, loosely typed languages (e.g., JavaScript), where variables do not have type but they accept values of any type. This is similar to an image tag making a request to a “.jsp” page and changing its state. If we can imagine the web analogous to static, strongly typed languages (e.g., Java), an image tag will only be able to request content of the type “image” and not a “.jsp” page. This prevents several malicious cross origin attacks.

Since it is not possible to infer the type of content before making a request, we propose a new HTTP response header called CORP, which contains a key-value mapping between “types” of content and their “resource paths”. This ensures that a tag (e.g., image) can make a request to a URL which matches with ”resource path” of that type.

5.2 Resource Paths

Every resource on the web is identified by a unique path. As a good engineering practice, web administrators often organize different types of resources (e.g., images, scripts etc) under different directories (e.g., *http://A.com/images*, *http://A.com/js* etc.) on the server hosting the resources. Based on this observation, we have defined the signature *Path* in the Post-CORP model. It has various subtypes as shown in Listing 10. The subtype *NonSensitivePagesPath* refers to pages that do not contain any sensitive content, while the subtype *SensitivePagesPath* refers to pages that contain sensitive content and need utmost protection.

```
abstract sig Path {}
one sig ImgPath, JsPath, CssPath,
    NonSensitivePagesPath,
    SensitivePagesPath extends Path {}
abstract sig Server {
        resourcePath: set Path
}
```

Listing 10: Cross Origin Transaction.

5.2.1 Predicate: maliciousXOriginTransaction

The predicate *maliciousXOriginTransaction* is an enhancement to the predicate *crossOriginTransaction* defined in Section 4.9. The type *Server* has been extended with the subtypes *EvilServer* and *GenuineServer*. Listing 11 shows the contraints for this predicate.

```
sig EvilServer, GenuineServer extends Server
     {}
pred maliciousXOriginTransaction{
  some t:HTTPTransaction|{
    t.req.from.activeDoc.docOrigin.pointsTo=
        EvilServer
    t.req.to=GenuineServer
    t.req.reqPath = SensitivePagesPath
  }
}
run maliciousXOriginTransaction for 3 but
    exactly 1 HTTPTransaction expect 1
```

Listing 11: Predicate: maliciousXOriginTransaction.

5.2.2 Predicate: CORPCompliantTransaction

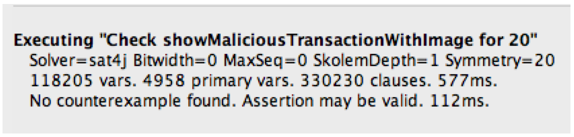
The predicate *corpCompliantTransaction* shown in Listing 12 takes three arguments of type *Origin*, *HTTPEventInitiator* and *Path* respectively. It asks Alloy to produce an instance of the model where the following constraints hold good for at least one HTTP transaction:

Line 3: The request must originate from EvilServer.
Line 4: The request must be made to GenuineServer.
Line 5: The origin of the HTTP event initiator must be the same as the predicate's argument *o*.
Line 6: The HTTP event initiator that triggers an HTTP transaction must be the same as the predicate's argument *ev*.
Line 7: The path to which the HTTP request is made must be the same as the predicate's argument *pt*.

```
pred corpCompliantTransaction[o:Origin, ev:
    HTTPEventInitiator, pt: Path] {
  some t:HTTPTransaction {
    t.req.from.activeDoc.docOrigin.pointsTo=
        EvilServer
    t.req.to=GenuineServer
    t.req.from.activeDoc.docOrigin = o
    httpTrans.t= ev
    t.req.reqPath=  pt
  }
}
pred restrictImagesWithCorp{
  corpCompliantTransaction[origin2,
      ImageElement, ImgPath]
}
pred restrictScriptsWithCorp{
  corpCompliantTransaction[origin2,
      ScriptElement, JsPath]
}
```

Listing 12: Predicate: corpCompliantTransaction.

The Listing 12 also shows three more predicates - *restrictImagesWithCorp*, *restrictScriptsWithCorp* and *restrictJsCodeWithCorp*. Each of them in-turn invoke the predicate *corpCompliantTransaction* with various arguments. For example, the predicate *restrictJsCodeWithCorp* asks Alloy to produce an instance where an HTTP request is made to a *GenuineServer* by a *JavaScriptCode* from a document having an origin *origin2*. It also mandates the request to be made only to the path *NonSensitivePagesPath* on the *GenuineServer*.



```
Executing "Check showMaliciousTransactionWithImage for 20"
  Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
  118205 vars. 4958 primary vars. 330230 clauses. 577ms.
  No counterexample found. Assertion may be valid. 112ms.
```

Figure 4: Checking the post-CORP assertion *showMaliciousTransactionWithImage*

5.2.3 Assert: showMaliciousTransactionWithImage

Listing 13 shows an instance of the model where a cross origin HTTP transaction initiated by a Image tag can be restricted through CORP to non-sensitive pages only. While the instances produced by Alloy show that the predicate is consistent, it is equally important to assert the negation. i.e., To find if there exists a cross origin HTTP transaction triggered by Image tag to the genuine server, where the predicate *restrictImagesWithCorp* is violated?

```
assert showMaliciousTransactionWithImage {
        no t:HTTPTransaction |{
                restrictImagesWithCorp
                t.~httpTrans.initiatorOrigin
                     = origin2
                t.~httpTrans=ImageElement
                t.req.reqPath! = ImgPath

        }
}

check showMaliciousTransactionWithImage for
    20
```

Listing 13: Assert: showMaliciousTransactionWithImage.

The assertion *showMaliciousTransactionWithJsCode* shown in Listing 13 verifies if such a possibility exists. As shown in Figure 4, Alloy fails to produce a counterexample when the assertion *showMaliciousTransactionWithJsCode* is checked.

Thus, with the results shown by the predicate *restrictJsCodeWithCorp* and the assertion *showMaliciousTransactionWithJsCode*, it can be said that the post-CORP model is sound.

6 CONCLUSION AND FUTURE WORK

In this paper, we present a simple model of web platform which comprises of some basic components of web security like origin, HTML elements and HTTP transaction, browser and servers. Our threat model includes modelling of malicious cross-origin requests and showing that it can be mitigated using CORP. This model can be extended to include other web security vulnerabilities and verifying their mitigation measures.

As future work, we plan to model specific cross-origin attacks, such as CSRF, clickjacking, cross-site timing attacks, login detection, and verify that CORP can be used to mitigate them. We also plan to test CORP with complex cross-origin request scenarios, such as Federated Identity Management.

REFERENCES

Clickjacking. https://www.owasp.org/index.php/Clickjacking.

Cross-site scripting (XSS). https://en.m.wikipedia.org/wiki/Cross-site_scripting.

Web Application Timing attack. https://codeseekah.com/2012/04/29/timing-attacks-in-web-applications/.

(2012). I Know What Websites You Are Logged-In To (Login-Detection via CSRF). https://www.whitehatsec.com/blog/i-know-what-websites-you-are-logged-in-\to-login-detection-via-csrf/.

(2016). Cross-site request forgery. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

Akhawe, D., Barth, A., Lam, P. E., Mitchell, J., and Song, D. (2010). Towards a formal foundation of web security. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 290–304. IEEE.

Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., Drielsma, P. H., Héam, P.-C., Kouchnarenko, O., Mantovani, J., et al. (2005). The avispa tool for the automated validation of internet security protocols and applications. In *International Conference on Computer Aided Verification*, pages 281–285. Springer.

Bhargavan, K., Fournet, C., and Gordon, A. D. (2006). Verified reference implementations of ws-security protocols. In *International Workshop on Web Services and Formal Methods*, pages 88–106. Springer.

Blanchet, B., Abadi, M., and Fournet, C. (2005). Automated verification of selected equivalences for security protocols. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 331–340. IEEE.

Cao, Y., Rastogi, V., Li, Z., Chen, Y., and Moshchuk, A. (2013). Redefining web browser principals with a configurable origin policy. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE.

Carlucci Aiello, L. and Massacci, F. (2001). Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic (TOCL)*, 2(4):542–580.

Chen, E. Y., Bau, J., Reis, C., Barth, A., and Jackson, C. (2011). App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 227–238. ACM.

Clarke, E. M., Jha, S., and Marrero, W. (2000). Verifying security protocols with brutus. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):443–487.

Cremers, C. J. (2008). The scyther tool: Verification, falsification, and analysis of security protocols. In *International Conference on Computer Aided Verification*, pages 414–418. Springer.

De Ryck, P., Desmet, L., Joosen, W., and Piessens, F. (2011). Automatic and precise client-side protection against csrf attacks. In *European Symposium on Research in Computer Security*, pages 100–116. Springer.

Gordon, A. D. and Pucella, R. (2005). Validating a web service security abstraction by typing. *Formal Aspects of Computing*, 17(3):277–318.

Jackson, D. (2012). *Software Abstractions: logic, language, and analysis*. MIT press.

Telikicherla, K. C. and Choppella, V. (2013). Alloy model for cross origin request policy (corp). Technical report.

Telikicherla, K. C., Choppella, V., and Bezawada, B. (2014). Corp: A browser policy to mitigate web infiltration attacks. In *International Conference on Information Systems Security*, pages 277–297. Springer.

Zalewski, Michal (2011). Browser Security Handbook. Technical report. https://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy.