Alloy model for Cross Origin Request Policy (CORP)

by

Krishna Chaitanya, Venkatesh Choppella

Report No: IIIT/TR/2013/-1

Centre for Software Engineering Research Lab
International Institute of Information Technology
Hyderabad - 500 032, INDIA
August 2013

Alloy model for Cross Origin Request Policy (CORP)

Krishna Chaitanya Telikicherla
krishnachaitanya.t@research.iiit.ac.in

Venkatesh Choppella
venkatesh.choppella@iiit.ac.in

18th August, 2013

Abstract

This document describes the formal model for Cross Origin Request Policy (CORP), a new browser security policy proposed for enhancing the security of the web platform. CORP aims to defend against several Cross Origin Request Attacks (CORA) such as CSRF, Clickjacking, 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. Alloy [2], a finite state model finder, is used to formalize CORP and verify its soundness.

1 Introduction

Analyzing the security of the web platform is a daunting task, since it 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. [1] 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 wide-spread application of Alloy in verifying web specifications and architectures, we have used it to formalize and verify the soundness of our proposal, CORP.

Below are some of the main characteristics of our model.

1.1 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 intereactions, 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 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.

1.3 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, 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, Web application timing attacks, Clickjacking, Login detection 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.

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.

1.4 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, 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.

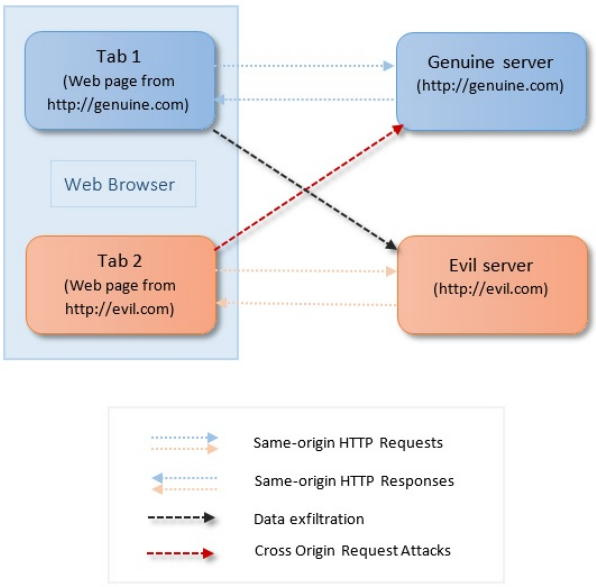

Figure 1: Exfiltration vs Cross Origin Request Attacks

1.5 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.

2 Modelling cross-origin requests in the web platform (Pre-CORP.als)

Figure 2 shows the meta model of our Pre-CORP model.

2.1 HTTP Transactions

The code in listing 1 shows our abstraction of an HTTP transaction. Figure 3 shows Alloy's instance of an HTTP transaction (projected over multiple signatures for simplicity). A HTTPTransaction is a type which consists of exactly one HTTP request and exactly one HTTP response. The types HTTPRequest and HTTPResponse extend HTTPEvent (i.e., they are subsignatures of HTTPEvent). This means, a HTTPEvent can be either a HTTPRequest or a HTTPResponse and its fields (here, the "host" field) will be inherited by both its subsignatures.

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. Furthermore, an

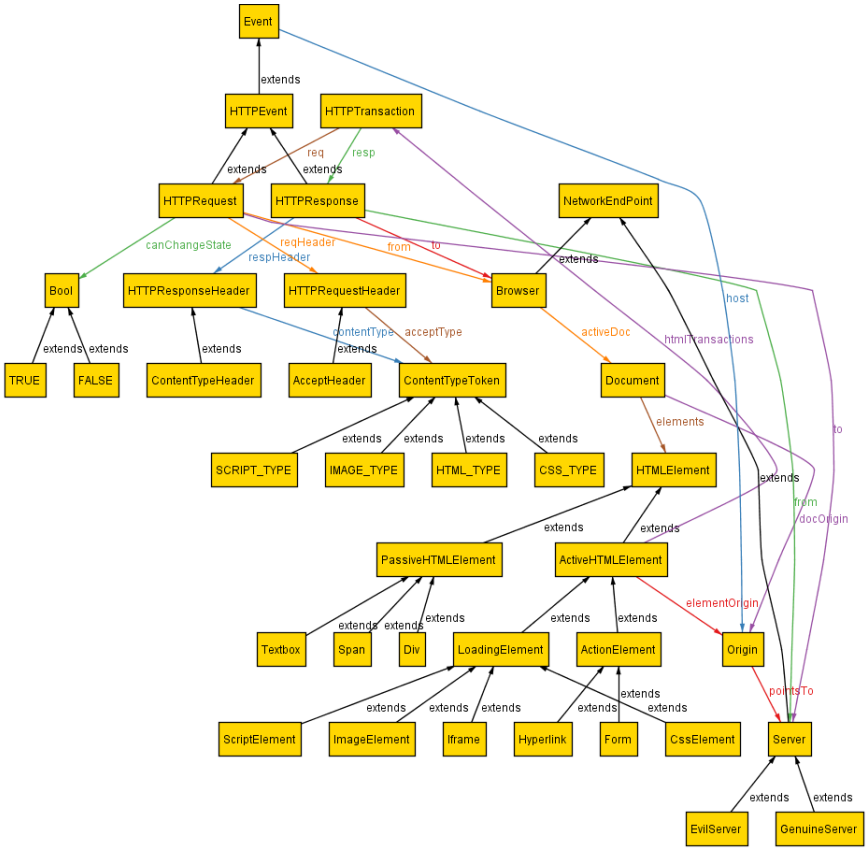

Figure 2: Meta model - Cross origin web requests (Pre-CORP)

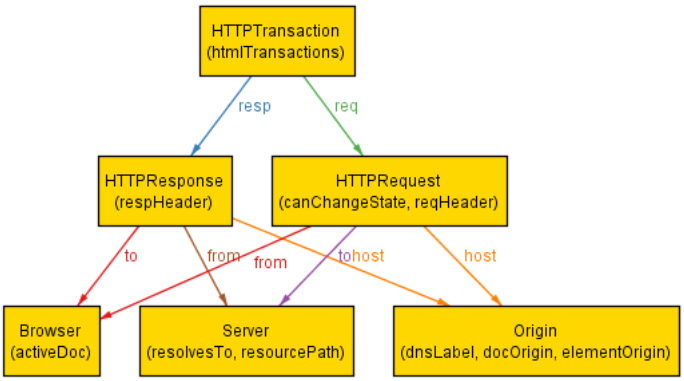

Figure 3: Model of HTTP Transaction

HTTP request has a set of request headers (denoted by the signature "HTTPRequestHeader"), a request path (which is the URI to which a request is made) and a field "canChangeState", which takes a boolean value indicating whether the request has the ability to change state on the server or not. An HTTP response has a set of response headers (denoted by the signature "HTTPResponseHeader"). The "AcceptHeader" extends HTTPRequestHeader while the "ContentTypeHeader" extends HTTPResponseHeader.

```
1  abstract sig Event{}
2  abstract sig HTTPEvent extends Event {
3          host : Origin
4  }
5  abstract sig HTTPTransaction{
6          req:  one HTTPRequest,
7          resp: one HTTPResponse
8  }
9  sig HTTPRequest extends HTTPEvent{
10         from: one Browser,
11         to: one Server,
12         reqHeader: HTTPRequestHeader,
13         canChangeState: one Bool
14 }
15 sig HTTPResponse extends HTTPEvent{
16         from: one Server,
17         to: one Browser,
18         respHeader: HTTPResponseHeader
19 }
```

Listing 1: Basic HTTP Transactions in our model

2.2 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,

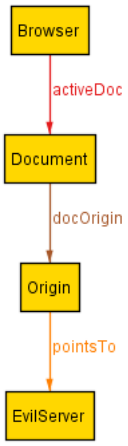

Figure 4: Basic browser model

which is the origin of the webpage loaded in the browser. Each origin points to exactly one server, which can be either a GenuineServer or an EvilServer. Figure 4 depicts this key relation.

```
1  sig Origin{
2          pointsTo: one Server
3  }
4
5  abstract sig NetworkEndPoint{}
6
7  sig Browser extends NetworkEndPoint{
8          activeDoc: Document
9  }
10
11 abstract sig Server extends NetworkEndPoint{}
12
13 sig EvilServer, GenuineServer extends Server{}
14
15 sig Document{
16         docOrigin: Origin,
17         elements: set HTMLElement
18 }
```

Listing 2: Basic browser model

2.3 Elements

The code in listing 3 shows our classification of DOM elements based on their capability to trigger HTTP requests. We classify elements which cannot trigger HTTP calls as PassiveHTMLElements (e.g., Div, Span, Textbox etc.) while those which can trigger HTTP calls as ActiveHTMLElements. Even further, we classify ActiveHTMLElements as LoadingElements - elements which automatically trigger HTTP requests as soon as they get added to the DOM tree

(e.g., img, script, iframe etc) and ActionElements - elements which require an action by humans to trigger HTTP requests (e.g., hyperlinks, forms).

```
1  sig HTMLElement{}
2  sig PassiveHTMLElement extends HTMLElement {}
3  sig  Div, Span, Textbox extends PassiveHTMLElement{}
4  abstract sig ActiveHTMLElement extends HTMLElement {
5          elementOrigin: Origin,
6          htmlTransactions: lone HTTPTransaction
7  }
8  abstract sig LoadingElement, ActionElement extends
       ActiveHTMLElement {}
9  sig ScriptElement, ImageElement, CssElement, Iframe extends
       LoadingElement {}
10 sig Hyperlink, Form extends ActionElement{}
```

Listing 3: DOM elements and transactions

2.4 Type tokens

Every HTTP request has an "accept" header (denoted by "AcceptHeader" in our model) as one of its request headers and every HTTP response has a "content-type" header (denoted by "ContentTypeHeader" in our model) as one of its response headers. These headers have "content-type" (denoted by "Content-TypeToken" in our model) as their value. Through the accept header, a browser sends the types of content which it is expecting from the server. Through the content-type header, the server sends the type of content which it is serving. Understanding how these types vary in request/response plays an important role in the design of CORP. IMAGE, CSS, SCRIPT, HTML are some of the content types which we have considered in our model. Token types can be seen in our metamodel in figure 2

```
1  sig AcceptHeader extends HTTPRequestHeader {acceptType:
       ContentTypeToken}
2  sig ContentTypeHeader extends HTTPResponseHeader {
       contentType: ContentTypeToken}
3  abstract sig ContentTypeToken{}
4  sig IMAGE_TYPE, CSS_TYPE, SCRIPT_TYPE, HTML_TYPE extends
       ContentTypeToken{}
```

Listing 4: Type Tokens

2.5 Fact: TransactionRules

Listing 5 shows the rules which ensure that the model is sane with resepct 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 9: Two transactions should not interfere with each other's request

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

For all HTTPTransactions, the following rule should hold good:

Line 12: All HTTPTransactions should be due to some element.

```
1  fact TransactionRules{
2          all t:HTTPTransaction, b:Browser, s:Server| {
3                  t.req.host = t.resp.host
4                  t.req.host = t.req.to.~pointsTo
5                  s in t.req.to => s in t.resp.from
6                  b in t.req.from => b in t.resp.to
7          }
8          all disj t1,t2: HTTPTransaction | {
9                  no (t1.req & t2.req)
10                 no (t1.resp & t2.resp)
11         }
12         HTTPTransaction in ActiveHTMLElement.
               htmlTransactions
13 }
```

Listing 5: Fact - Transaction Rules

2.6 Fact: ElementsInheritParentOrigin

This fact says that for all instances of HTMLElement, the element's parent document's origin must be the same as the element's origin.

```
1  fact ElementsInheritParentOrigin{
2          all elem:HTMLElement | elem.~elements.docOrigin =
               elem.elementOrigin
3  }
```

Listing 6: Fact - ElementsInheritParentOrigin

2.7 Fact: BrowserSetsAcceptType

When an element makes an HTTP request, browsers automatically associate an accept header which takes a value of the form "type/subtype". e.g., If an < img >makes an HTTP request, browsers typically set accept header to "img/*". Same is the case with other requests.

```
fact BrowserSetsAcceptType{
        all t:HTTPTransaction | let acceptType=t.req.
            reqHeader.acceptType |{
                some ImageElement => acceptType = IMAGE_TYPE
                some CssElement => acceptType = CSS_TYPE
                some ScriptElement => acceptType =
                    SCRIPT_TYPE
                some Iframe => acceptType = HTML_TYPE
                some Hyperlink => acceptType = HTML_TYPE
                some Form => acceptType = HTML_TYPE
        }
}
```

Listing 7: Fact - BrowserSetsAcceptType

2.8 Fact: Disjointness

This fact 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.
            htmlTransactions & e2.htmlTransactions)
}
```

Listing 8: Fact - Disjointness

2.9 Predicate: wrongResponseTypePossible

This predicate produces instances where the accept type and response contentType do not intersect. Figure 5 is one such instance.

```
pred wrongResponseTypePossible {
        some elem:LoadingElement
        {
                no (elem.htmlTransactions.req.reqHeader.
                    acceptType &
                                elem.htmlTransactions.resp.
                                    respHeader.contentType)
        }
}
```

Listing 9: Predicate - wrongResponseTypePossible

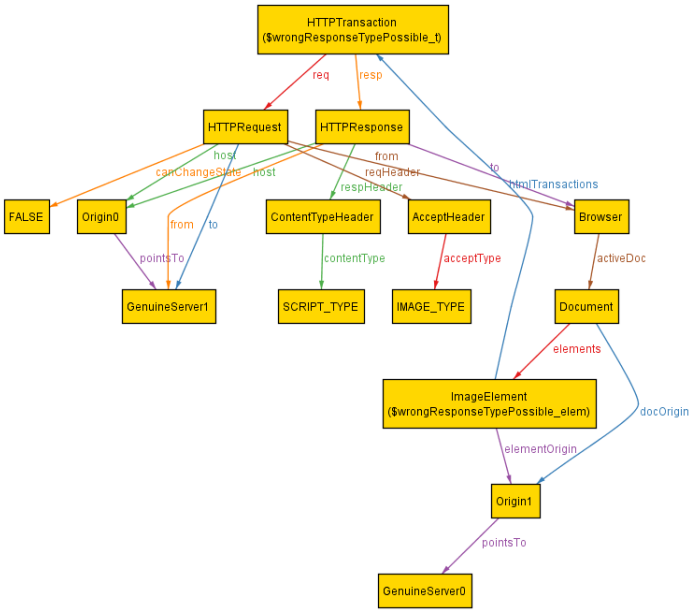

Figure 5: Predicate - wrongResponseTypePossible

2.10 Predicate: ElementCanChangeState

This predicate shows instances where elements (both loading and action elements) can change state on the server. Figure 6 shows an instance where a loading element can change state. Figure 7 shows an instance where an action element can change state.

```
1  pred ElementCanChangeState {
2          some elem:ActiveHTMLElement {
3                  elem.htmlTransactions.req.canChangeState=
                       TRUE
4          }
5  }
6  pred loadingElementCanChangeState {
7          ElementCanChangeState
8          some (ActiveHTMLElement - ActionElement)
9  }
10
11 pred actionElementsCanChangeState{
12         ElementCanChangeState
13         some (ActiveHTMLElement - LoadingElement)
14 }
```

Listing 10: Predicate - ElementCanChangeState

2.11 Predicate: almostSafeXOriginTransaction

This predicate shows instances of cross origin transactions, which are almost safe (does not change state on the server). It is still vulnerable to web application

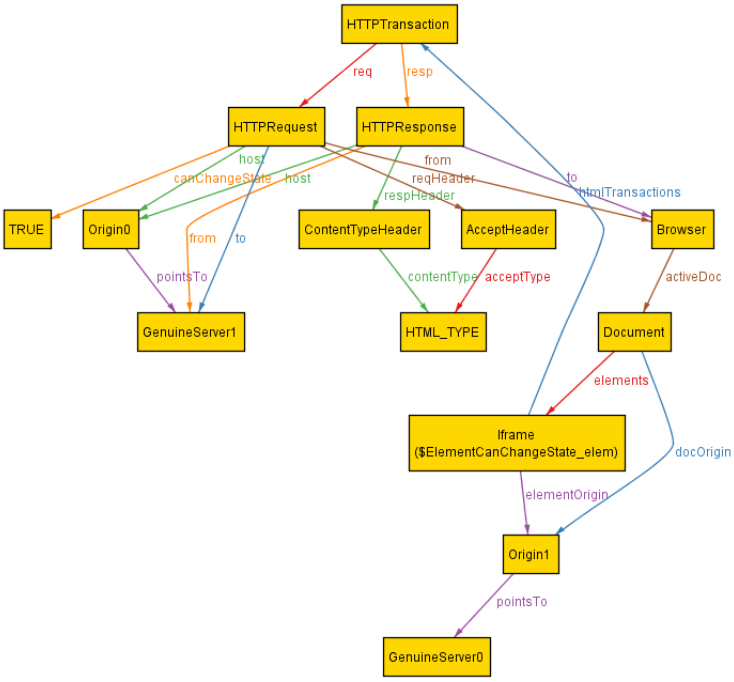

Figure 6: Predicate - loadingElementCanChangeState

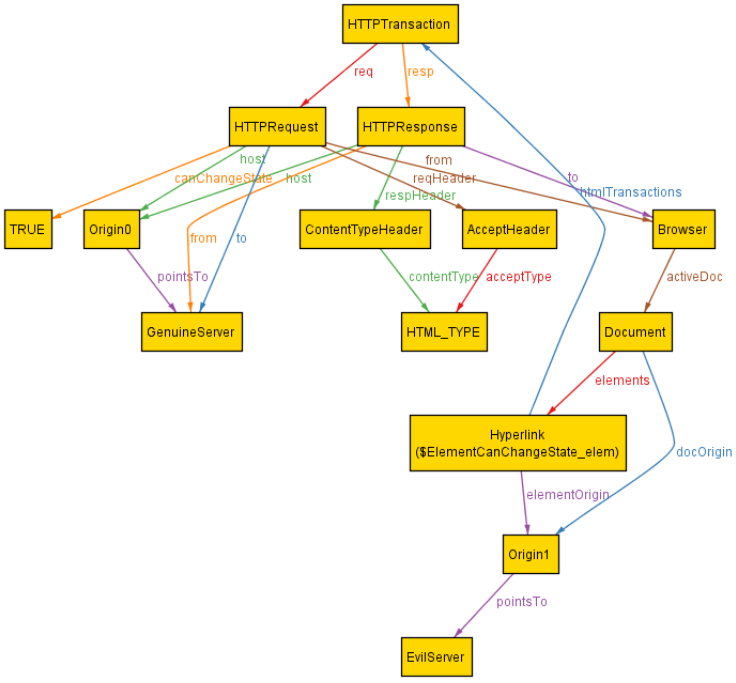

Figure 7: Predicate - actionElementsCanChangeState

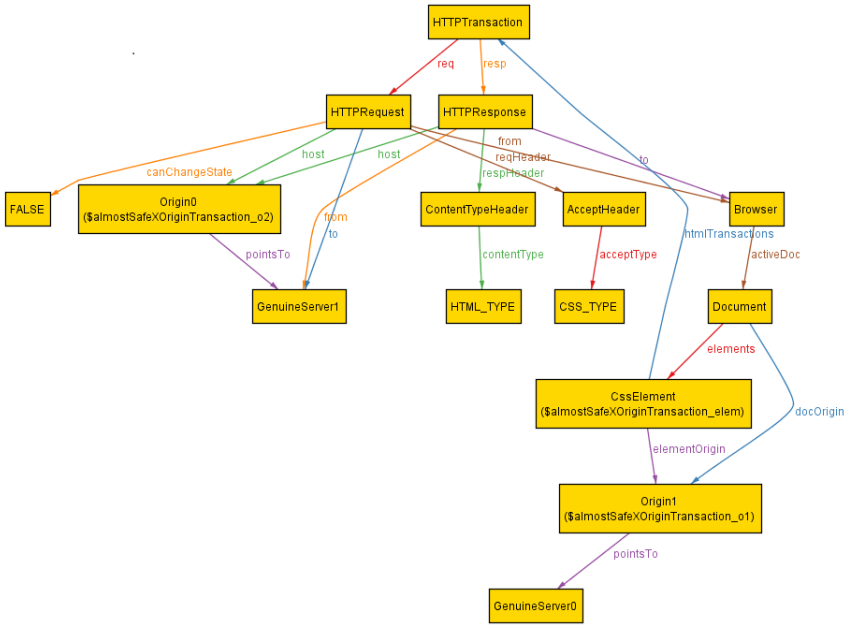

Figure 8: Predicate - almostSafeXOriginTransaction

timing attacks, so we call it almost safe. Listing 11 shows the code for this predicate. It says, for some disjoint origins o1, o2 and some ActiveHTMLElement, show instances where:

Line 3: Element belongs to an origin o1 and

Line 4: Element makes an HTTP transaction to another origin o2.

Line 6: The transaction does not cause a change in state on the server.

Line 7: Only Genuine servers are involved in the transaction.

Figure 8 is an instance of this predicate.

```
1  pred almostSafeXOriginTransaction{
2          some disj o1, o2: Origin,  elem:ActiveHTMLElement|{
3                  elem.~elements.docOrigin=o1
4                  elem.htmlTransactions.req.host=o2
5          }
6          HTTPTransaction.req.canChangeState=FALSE
7          no (Server & EvilServer)
8          #Browser = 1
9          #Server = 2
10         #HTTPTransaction =1
11 }
```

Listing 11: Predicate - almostSafeXOriginTransaction

2.12 Predicate: maliciousXOriginTransaction

This predicate shows instances of cross origin transactions, which are malicious (causes a change in state on the server). It says, for some disjoint origins o1, o2 and some ActiveHTMLElement, show instances where:

Line 3: Element belongs to an origin o1 and

Line 4: Element makes an HTTP transaction to another origin o2.

Line 5: Element's origin points to evil server (in other words, the element belongs to a malicious document).

Line 6: Element triggers an HTTP request to a genuine server.

Line 8: The transaction causes a change in state on the genuine server.

Figure 9 is an instance of this predicate.

```
1  pred maliciousXOriginTransaction{
2          some disj o1, o2: Origin, elem:ActiveHTMLElement|{
3                  elem.~elements.docOrigin=o1
4                  elem.htmlTransactions.req.host=o2
5                  elem.elementOrigin.pointsTo=EvilServer
6                  elem.htmlTransactions.req.to=GenuineServer
7          }
8          HTTPTransaction.req.canChangeState=TRUE
9          #Browser = 1
10         #Server = 2
11         #HTTPTransaction =1
12 }
```

Listing 12: Predicate - maliciousXOriginTransaction

3 Modelling restrictions introduced in CORP (Post-CORP.als)

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 CORA to a great extent. Figure 10 shows meta model of the Post-CORP alloy code.

3.1 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

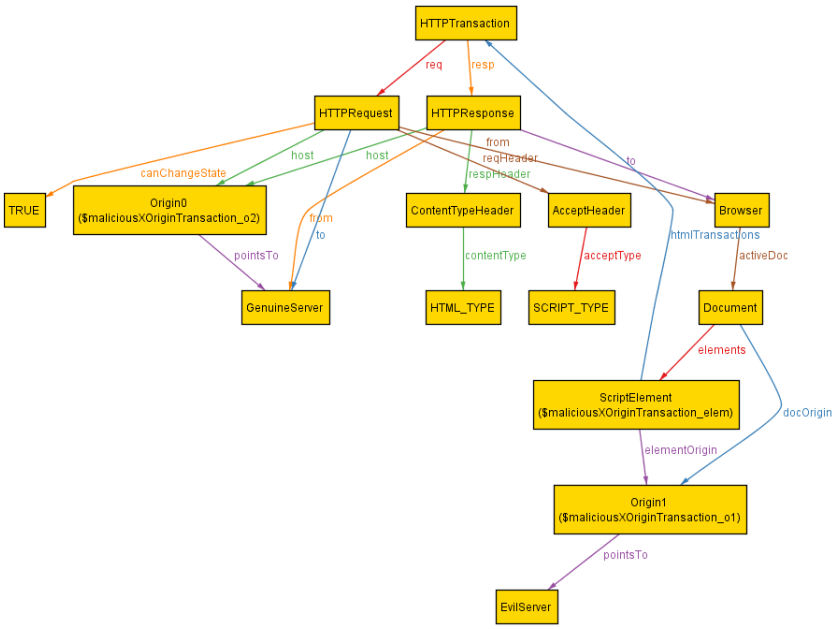

Figure 9: Predicate - maliciousXOriginTransaction

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.

3.2 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) in different directories (e.g., A.com/images, A.com/js etc). Based on this observation, we have defined different paths in our model, which host content based on their type . The code in listing 13 shows the path hierarchy in our model.

```
1 sig HTTPRequest extends HTTPEvent{
2         ...
3         reqPath: one Path
4 }
5 sig Server extends NetworkEndPoint{
6         resourcePath: set Path
7 }
8 abstract sig Path{}
```

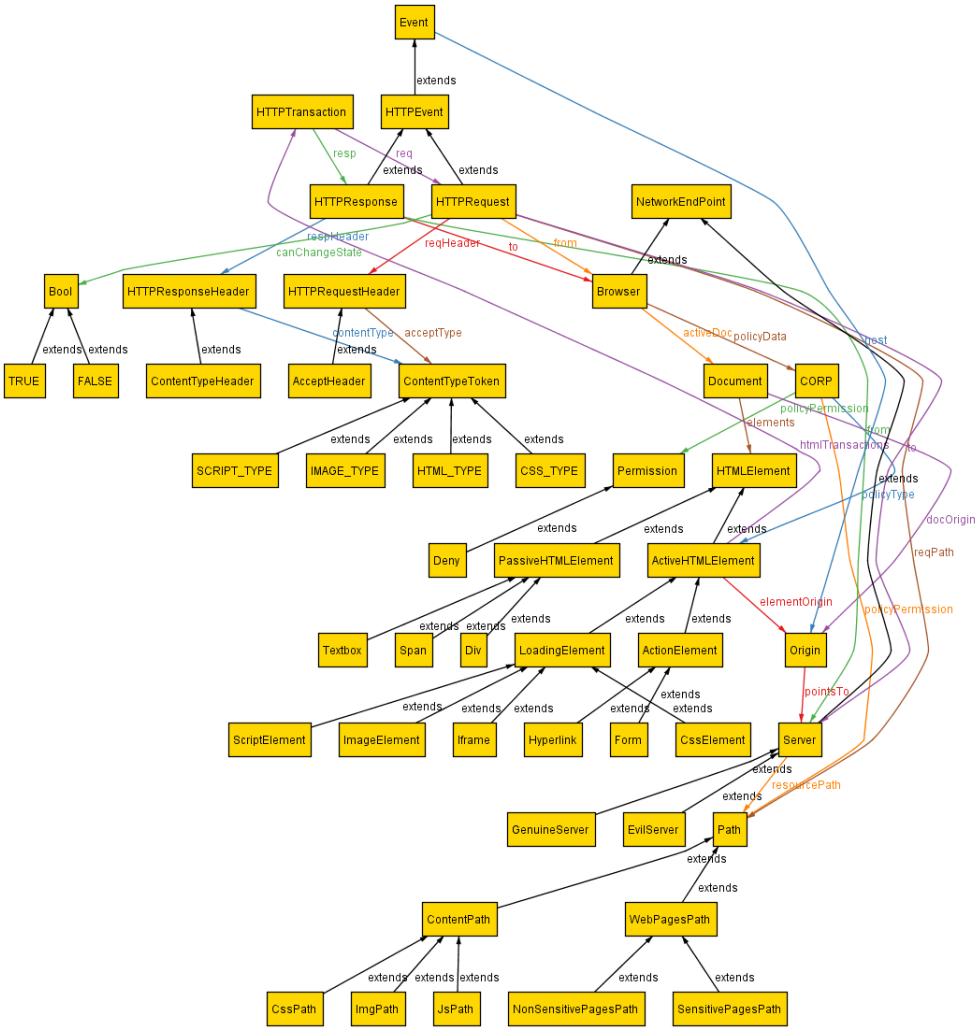

Figure 10: Meta model after adding cross origin restrictions (Post-CORP)

```
9  abstract sig ContentPath, WebPagesPath extends Path{}
10 sig ImgPath, JsPath, CssPath extends ContentPath{}
11 sig NonSensitivePagesPath, SensitivePagesPath extends
       WebPagesPath{}
```

Listing 13: Resource paths

As shown in the code, a HTTPRequest has a request path, which is of the type "Path". Path is in turn extended by ContentPath (which is extended by ImgPath, JsPath, CssPath) and WebPagesPath (which is extended by NonSensitivePagesPath and SensitivePagesPath). The path hierarchy can be seen in our metamodel in Figure 10.

3.3 CORP signature

Every web transaction is accompanied by HTTP headers along with content. Figure 11 shows a snapshot of HTTP request and response headers from Google. When a user navigates to a CORP enabled website, CORP rules are sent to the browser through the response headers of the site. A CORP enabled browser will enforce the rules in CORP on subsequent cross origin requests. To make the model simpler, we assume the aforementioned HTTP transaction has taken place and CORP rules reside in the browser (i.e., Browser.policyData has a set of CORP rules). CORP rules in the response headers are sent as key:value format. Figure 12 may be considered as a sample to understand CORP signature. In our model, we represent the key, value pair as "policyType" and "policyPermission" respectively. CORP.policyType maps to an ActiveHTMLElements while CORP.policyPermission points to a value which is a union of path and permission. CORP and its fields can be seen in our metamodel in Figure 10.

```
1  sig Browser extends NetworkEndPoint{
2          ...
3          policyData: set CORP
4  }
5  abstract sig Permission {}
6  sig Deny extends Permission {}
7  abstract sig CORP{
8          policyType: ActiveHTMLElement,
9          policyPermission: Path + Permission
10 }
```

Listing 14: CORP signature

3.4 Fact: ServerSetsContentType

When an HTTP request is made to a server, the following conditions should hold:

- If the response type is image, then the server should contain an image resource. i.e., some (ImgPath & server.resourcePath)
- The request path should point to server.resouce path. i.e., (some reqPath & server.resourcePath)

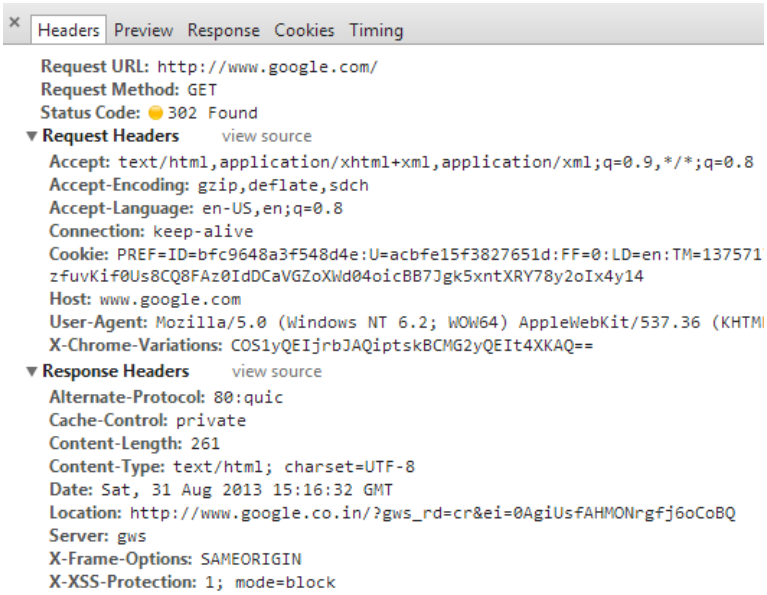

Figure 11: Snapshot of HTTP headers from http://google.com

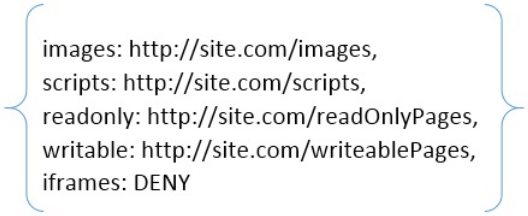

Figure 12: Sample CORP rules

The above conditions can be written as: some (ImgPath & server.resourcePath & reqPath)

```
1  fact ServerSetsContentType{
2          all t:HTTPTransaction | let contentType=t.resp.
               respHeader.contentType, server=t.resp.from,
               reqPath=t.req.reqPath | {
3                  #server.resourcePath>1
4                  contentType= IMAGE_TYPE => some (ImgPath &
                       server.resourcePath & reqPath)
5                  contentType= CSS_TYPE => some (CssPath &
                       server.resourcePath & reqPath)
6                  contentType= SCRIPT_TYPE => some (JsPath &
                       server.resourcePath & reqPath)
7                  contentType= HTML_TYPE => some (WebPagesPath
                        & server.resourcePath & reqPath)
8          }
9  }
```

Listing 15: Fact - ServerSetsContentType

3.5 Fact: StateChangeRules

This fact states that, if an HTTP request can change server state, its path should be only to sensitive pages. Else, it should be only to non-sensitive pages.

```
1  fact StateChangeRules{
2          all t:HTTPTransaction | {
3                  t.req.canChangeState=TRUE => some (t.req.
                       reqPath & SensitivePagesPath)
4                  t.req.canChangeState=FALSE => no
                       SensitivePagesPath
5          }
6  }
```

Listing 16: Fact - StateChangeRules

3.6 Fact: CorpEnforcement

This fact states that, if CORP is configured, the following conditions should hold:

Line 4: No cross origin state changes are allowed

Line 5: Accept-type and content-type must match

Line 6,7: If the policyPermission is NonSensitivePagesPath, then HTTP request can be made only to NonSensitivePagesPath. Similar is the case with rest of the values for policyPermisison.

Line 18, 19: If the policyPermission is Deny, then no HTTP transactions will be allowed for that element type.

```
fact CorpEnforcement{
        some Browser.policyData implies {
                let policyPermission=CORP.policyPermission,
                    elemTrans=ActiveHTMLElement.
                    htmlTransactions| {
                        elemTrans.req.canChangeState=FALSE
                        elemTrans.resp.respHeader.
                            contentType= elemTrans.req.
                            reqHeader.acceptType
                                policyPermission =
                                    NonSensitivePagesPath=> {
                                elemTrans.req.reqPath =
                                    NonSensitivePagesPath
                        }
                        policyPermission = ImgPath=> {
                                elemTrans.req.reqPath =
                                    ImgPath
                        }
                        policyPermission = JsPath=> {
                                elemTrans.req.reqPath =
                                    JsPath
                        }
                        policyPermission = CssPath=> {
                                elemTrans.req.reqPath =
                                    CssPath
                        }
                        policyPermission = Deny => {
                                no CORP.policyType.
                                    htmlTransactions
                                noIdleElementsInCORP
                        }
                }
        }
}
```

Listing 17: Fact - CorpEnforcement

3.7 Predicate: ConfigureCORP

This predicate is used for configuring CORP. ConfigureCORP[0] removes the policy and ConfigureCORP[1] enables the policy.

```
pred ConfigureCORP[num: Int]{
        #CORP = num
}
```

Listing 18: Predicate - ConfigureCORP

3.8 Predicate: noIdleElements

As per the signature "sig ActiveHTMLElement {htmlTransactions: lone HTTP-Transaction}", there can be ActiveHTMLElement which do not make HTTP transactions. This predicate disables such elements.

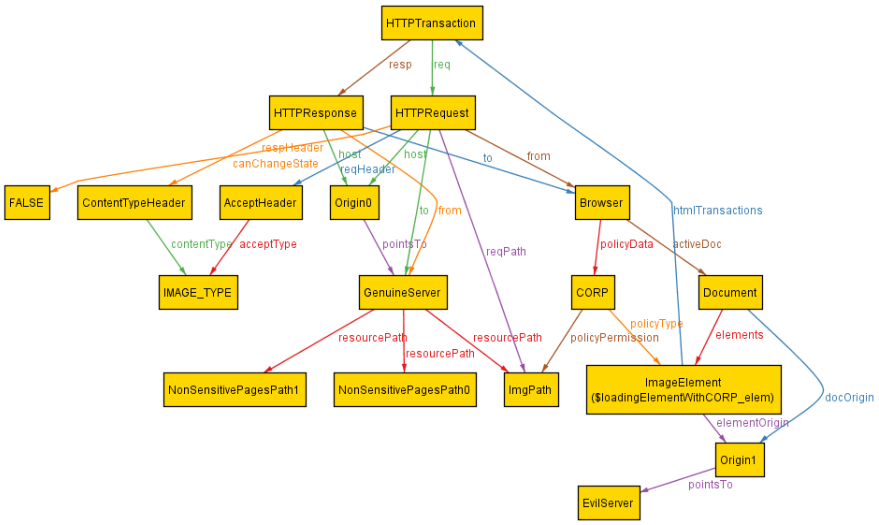

Figure 13: Predicate - loadingElementWithCORP

```
1  pred noIdleElements{
2          no (ActiveHTMLElement - htmlTransactions.
               HTTPTransaction)
3  }
```

Listing 19: Predicate - noIdleElements

3.9 Predicate: loadingElementWithCORP

This predicate shows instances of loading elements, with CORP configuration switched on. Figure 13 is one such instance. The following points need to be observed:

- CORP is configured on "Image" element and its value (resource path) is set to "ImgPath". So any request from images must point only to this path.
- A document loads from "Origin1" which points to an Evil server. An image in this evil document makes an HTTP request to "Origin2", which points to a Genuine server. However, the request is only sent to "ImgPath", as configured in CORP.
- Since the request originated from an image tag and the response is indeed an image, the accept-type and content-type will be the same.
- Since "ImgPath" contains only images and no sensitive pages (which can change the state on the server), there is no change of state in the Genuine server (HTTPRequest.canChangeState is "false" in the instance).

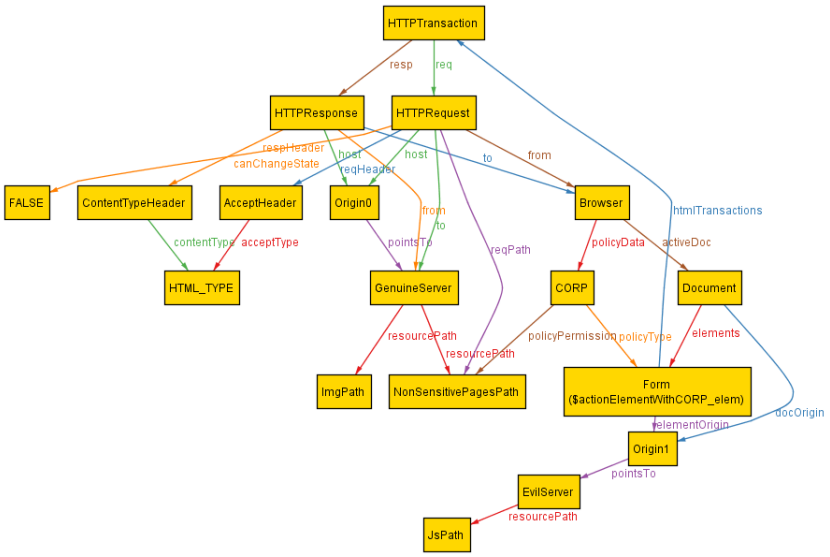

Figure 14: Predicate - actionElementWithCORP

```
pred loadingElementWithCORP{
        some elem: LoadingElement | {
                elem.elementOrigin.pointsTo=EvilServer
                elem.htmlTransactions.req.to=GenuineServer
        }
        ConfigureCORP[1]
        CORP.policyPermission=ContentPath
        noIdleElements
        #Browser = 1
        #Server = 2
        #HTTPTransaction =1
}
```

Listing 20: Predicate - loadingElementWithCORP

3.10 Predicate: actionElementWithCORP

This predicate shows instances of action elements, with CORP configuration switched on. CORP ensures cross origin attack scenarios do not appear in these instances. As explained in Section 3.9, similar observations can be made in the instances of this predicate.

```
pred actionElementWithCORP{
        some elem: ActionElement | {
                elem.elementOrigin.pointsTo=EvilServer
                elem.htmlTransactions.req.to=GenuineServer
        }
        ConfigureCORP[1]
        CORP.policyPermission=NonSensitivePagesPath
        noIdleElements
```

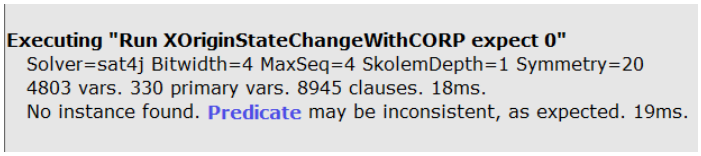

Figure 15: Predicate - XOriginStateChangeWithCORP

```
9           #Browser = 1
10          #Server = 2
11          #HTTPTransaction =1
12  }
```

Listing 21: Predicate - actionElementWithCORP

3.11 Predicate: XOriginStateChangeWithCORP

This predicate asks Alloy to show instances where X-origin state change is possible, after CORP is configured. As shown in Figure 15, Alloy does not produce any instance for this predicate.

```
1   pred XOriginStateChangeWithCORP{
2           XOriginTransaction
3           some elem:ActiveHTMLElement | elem.htmlTransactions.
                req.canChangeState=TRUE
4           ConfigureCORP[1]
5   }
```

Listing 22: Predicate - XOriginStateChangeWithCORP

3.12 Assert: wrongResponseTypeWithCORP

This assertion claims that there will be no instances of type mismatch between accept type and content type, when CORP is configured. As shown in Figure 16, Alloy fails to produce a counter example against this assertion.

```
1   assert wrongResponseTypeWithCORP{
2           no elem:ActiveHTMLElement, t:HTTPTransaction
3           {
4                   elem.htmlTransactions = t
5                   no (elem.htmlTransactions.req.reqHeader.
                        acceptType &
6                                       elem.htmlTransactions.resp.
                                            respHeader.contentType)
7                   ConfigureCORP[1]
8           }
9   }
```

Listing 23: Assert - wrongResponseTypeWithCORP

```
Executing "Check wrongResponseTypeWithCORP expect 0"
   Solver=sat4j Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   4694 vars. 324 primary vars. 8582 clauses. 22ms.
   No counterexample found. Assertion may be valid, as expected. 33ms.
```

Figure 16: Assert - wrongResponseTypeWithCORP

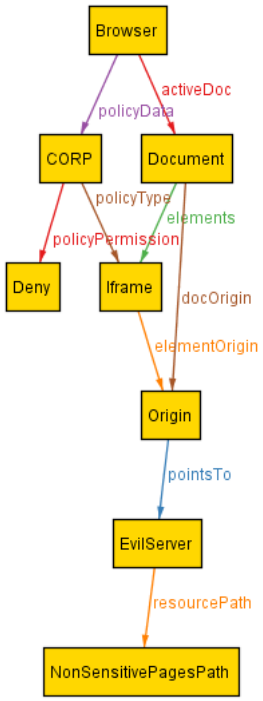

Figure 17: No framing attacks with CORP

3.13 Predicate: NoFramingAttacks

This predicate configures CORP for Iframes with a DENY rule. As shown in Figure 17, the instance produced by Alloy for this predicate does not show an HTTP transaction for iframes.

```
1 pred NoFramingAttacks{
2         ConfigureCORP[1]
3         CORP.policyType=Iframe
4         CORP.policyPermission=Deny
5 }
```

Listing 24: Predicate - NoFramingAttacks

References

[1] Devdatta Akhawe, Adam Barth, Peifung E Lam, John Mitchell, and Dawn Song. Towards a formal foundation of web security. In *Computer Secu-*

rity Foundations Symposium (CSF), 2010 23rd IEEE, pages 290–304. IEEE, 2010.

[2] Daniel Jackson. Software abstractions: Logic. *Language, and Analysis, The MIT Press*, 2006.