Privacy Breach by Exploiting postMessage in HTML5: Identification, Evaluation, and Countermeasure

Chong Guan[1,2,3]
guanchong@is.ac.cn

Kun Sun[4]
ksun@wm.edu

Zhan Wang[5]
kingzhan@gmail.com

Wen Tao Zhu[1,2]*
wtzhu@ieee.org

1. State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
2. Data Assurance and Communication Security Research Center, Chinese Academy of Sciences
3. University of Chinese Academy of Sciences
4. Department of Computer Science, College of William and Mary
5. RealtimeInvent, Inc.

ABSTRACT

The postMessage mechanism in HTML5 enables different webpage origins to exchange information and communicate. It becomes increasingly popular among the websites that need to import contents from third-party services, such as advertisements and preferable recommendations. Ideally, a receiver function should be locally implemented in the hosting page that needs to receive third-party messages. However, in the real world, the receiver function is usually provided by a third-party service provider, and the function code is imported via the HTML "script" tag so that the imported code is deemed as from the same origin with the hosting page. In the case that a site uses multiple third-party services, all the receiver functions imported by the hosting page can receive messages from any third-party provider. Based on this observation, we identify a new information leakage threat named the DangerNeighbor attack that allows a malicious service to eavesdrop messages from other services to the hosting page.

We study 5000 popular websites and find that the DangerNeighbor attack is a real threat to the sites adopting the postMessage mechanism. To defeat this attack, we propose an easily deployable approach to protecting messages from being eavesdropped by a malicious provider. In this approach, the site owner simply imports a piece of JavaScript code and specifies a mapping table, where messages from different origins are associated with corresponding receiver functions, respectively. The approach, which is transparent to the providers, ensures that a receiver function only receives messages from a specific origin.

*This author is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China

© 2016 ACM. ISBN 978-1-4503-4233-9/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2897845.2897901

Keywords

postMessage; HTML5; Privacy

1. INTRODUCTION

Third-party services such as advertisement, social recommendations, and performance measurement are commonly integrated in popular websites. In those scenarios, a site usually needs to communicate and exchange information with the third-party service provider. However, the principal security policy *Same Origin Policy* [12] in Web browsers is too restrictive to realize it. To address this, the fifth version of HTML (HTML5) introduces the postMessage mechanism, which enables web content from different origins being exchanged between different service providers.

Meanwhile, the postMessage mechanism raises new security problems. It is well-known that careless message content usage may introduce XSS vulnerability [15]. Barth et al. [18] carried out a comprehensive study of cross-frame communication in Web browsers and demonstrated attacks on the confidentiality of messages sent via postMessage under certain frame navigation policies. Cross-document messaging (i.e. postMessage) is considered the top security threat [14] in HTML5.

In this paper, we identify a new passive attack that appears when a hosting page receives messages from multiple senders (i.e. third-party providers). With postMessage mechanism, the message is able to be transmitted from one origin to another. Consequently, all the receiver functions in the hosting page are capable of receiving all the messages destined to the hosting page. Ideally, the receiver function should be implemented by the receiver site and there is no need to distinguish the receiver functions for the message. Unfortunately, in practice the receiver function is usually provided by the third-party service provider and is imported via the HTML script tag. If a site uses multiple third-party services, there will be many receiver functions from various providers all in the hosting page. In this scenario, since all the messages are shared among all receivers, some sensitive message data may be unintentionally leaked to a malicious provider. We call this attack *DangerNeighbor attack*.

To evaluate the threat of DangerNeighbor attack, we carry out a large-scale empirical study of the receiver functions

and messages in home pages of the Alexa [1] top 5000 sites. Further more, we focus on GIGYA [5] which is a popular customer identity management service provider and study the receiver functions and messages in the web sites that integrates the service provided by GIGYA. We find that it is common for hosting pages to contain multiple receiver functions from different providers. After clustering the collected messages data into a number of categories using the k-means algorithm, we find that the messages data contain various sensitive identity information such as userID, frameID, videoID, etc., which can be leveraged to distinguish a user and conduct user behavior tracking.

To defeat the DangerNeighbor attack, we propose an easily deployable protection approach based on JavaScript function wrapper technique. We wrap the built-in function that is used to attach a receiver function and replace the receiver function with a new one. The new receiver function searches the message origin in a message-receiverFunction table. If the corresponding receiver function in the table matches the original receiver function, the original function is invoked. Our approach can also help site owner thwart the XSS attack caused by the careless usage of the message in the provider's code. We also discuss an alternative encryption approach on the provider side to defend against the DangerNeighbor attack.

The contributions of this paper are threefold:

- First, we identify the DangerNeighbor attack in the HTML5 postMessage mechanism. This new attack enables a malicious third-party service provider to eavesdrop all the messages sent from other third-party service providers to the hosting page.
- Second, we evaluate the threat of DangerNeighbor attack in the real world and verify that the DangerNeighbor attack can be exploited to leak sensitive information on the websites that adopt the postMessage mechanism.
- Third, we propose a lightweight countermeasure to protect against DangerNeighbor attack. Our solution can also help site owner defeat the XSS attack due to the careless usage of the message in the provider's code. We also discuss an alternative approach based on the encryption on the provider side.

The remainder of this paper is organized as follows. Section 2 presents the background knowledge of this paper. Section 3 elaborates the DangerNeighbor attack and presents a test experiment. We evaluate the privacy breach caused by DangerNeighbor attack in Section 4. Section 5 provides a detailed description of our protection approach to defeat the DangerNeighbor attack. We discuss the related work in Section 6 and conclude the paper in Section 7.

2. BACKGROUND

2.1 The Same-Origin Policy

As the principal security policy in Web browsers, the *Same-Origin Policy* (SOP) [12] can effectively separate mutually distrusting Web content within the Web browser through origin-based compartmentalization. More precisely, the SOP allows a given JavaScript access only to resources that have the same origin. The origin is defined as the triple consisting of scheme, host, and port of the involved resources. For instance, a script executed under the origin of *attacker.org* is not able to access a user's personal information rendered under *webmail.com*. However, the Web content included using HTML script tag is not governed by the SOP. For instance, if a piece of JavaScript code from *thirdparty.com* is imported via <*script src="http://www.thirdparty.com/javascript.js"*>, it is the same origin with the hosting page.

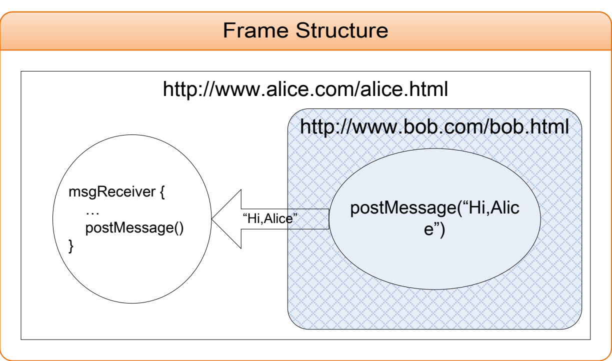


Figure 1: Example frame structure.

Listing 1: An example of using postMessage.

```
1  //Alice.html
2  <iframe src="http://www.bob.com/bob.html
       ">
3  </iframe>
4  <script>
5    function msgReceiver(event) {
6      if(event.origin=="http://www.bob.com
         "){
7        console.log("Bob says:" +
             event.data);
8        event.source.postMessage(
9          "Hi, Bob. Nice to meet you",
10         "http://www.bob.com");
11     }
12   }
13   window.addEventListener("message",
14     msgReceiver, false);
15 </script>
16
17
18 //Bob.html
19 <script>
20   window.parent.postMessage("Hi Alice.",
       "http://www.alice.com");
21 </script>
```

2.2 postMessage in HTML5

HTML5 is the fifth revision of the HTML standard, which is published as W3C Recommendation in October 2014 [6]. Most popular Web browsers such as Chrome [3] and Firefox [9] support HTML5.

The postMessage mechanism was first introduced by HTML5 for supporting cross document communication. The emerging postMessage breaks the origin-based compartmentalization of the SOP, since it allows one Web page to send string message from one origin to another.

The implementation of postMessage typically involves two

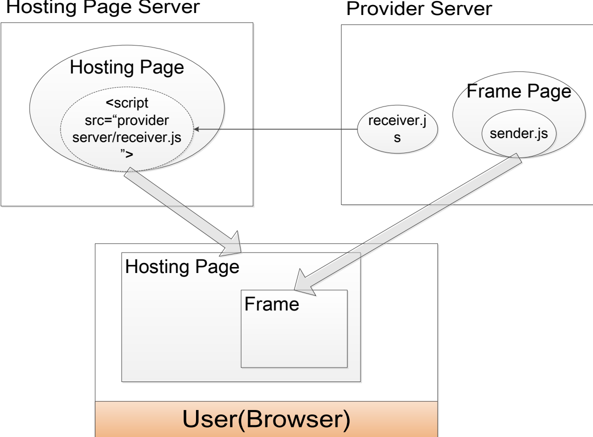


Figure 2: postMessage problematic usage pattern.

parties: the receiver and the sender. Figure 1 demonstrates the frame structure of an example where *Alice.html* is the receiver and *Bob.html* is the sender. The receiver *Alice.html* imports the content *Bob.html* as a frame and implements a receiver function *msgReceiver* in her origin. *Bob.html* sends message via the standard HTML5 postMessage API. Listing 1 elaborates the main HTML and JavaScript code used in the example. The function *msgReceiver* is a JavaScript callback function with one parameter. Whenever *Alice.html* receives a message, the MessageEvent is triggered and the callback function is invoked. The parameter is a *MessageEvent* object. It contains the message, the origin (scheme, host, and port) of the sender and a reference to the sender's frame. In this example, they correspond to the *event.data*, *event.origin* and *event.source*, respectively. As a sender, *Bob.html* first gets the reference of the target frame. In this example, the sender gets the reference by *window.parent* since *Alice.html* is his parent frame. Then the standard HTML5 postMessage API is called to send the message. The API accepts two parameters: the message and the receiver origin. The message parameter is a string usually in JSON [8] format. The sender relies on the receiver origin parameter to make sure the message is sent to the exact receiver since the frame may be navigated to an attacker's origin. Barth et al. elaborate why we need this parameter in [18]. However, in this paper, we demonstrate that it is not strong enough to secure the HTML5 postMessage.

2.3 postMessage in the Real World

The postMessage usage pattern is more problematic in the real world. The hosting page and the third-party service provider adopt the postMessage mechanism in the real world. We assume the hosting page is the receiver and the provider is the sender. The sender and receiver codes are both controlled by the provider, though the codes run in different origins. As is shown in Figure 2, the practical scenario involves three roles: the hosting page provider, the third-party service provider, and the user. The user requests the hosting page from the hosting page server. In the hosting page the third-party content is imported as a frame while the receiver code from the provider is imported via HTML script tag. The hosting page and the third-party content are both displayed on the user's browser with different origins.

Listing 2: JavaScript function wrapper demo.

```
1  //The original JavaScript widget
2  var obj = { // static object
3      // add function, supports adding 2 values
4      add: function(x, y) {
5          return x + y;
6      }
7  };
8
9  //Augment the original JavaScript widget
10 // move the existing function to another variable name
11 obj.__add = obj.add;
12
13 // override the existing function with your own, to support adding 2 or 3 values
14 obj.add = function(x, y, z) {
15     var val = obj.__add(x, y);
16     return z ? val + z : val;
17 }
```

However, the code in the HTML script tag has the same origin with the hosting page. It can receive messages from the third-party content for the hosting page and provides API for the hosting page. If the imported code is from a malicious provider, the hosting page will be compromised.

In this paper, we only consider the scenario that the imported code from the malicious provider eavesdropping the message transferred in the postMessage mechanism instead of compromising the hosting page. The general problem has been studied in detail in the previous research [23, 26, 31]. However, to the best of our knowledge, the existing countermeasure approaches cannot solve the problem incurred by postMessage.

2.4 JavaScript Function Wrapper

JavaScript function wrapper is a technique to extend or augment the behavior of certain functions without breaking the existing functionality when working with JavaScript libraries and widgets.

Listing 2 is a sample code using JavaScript function wrapper, which takes an existing function reference and maps it to a new reference. The original function is then overwritten with a new one that performs some extra actions before executing the original function. In Listing 2, a simple *add* that takes two parameters and adds them, is replaced by a new function that can take an optional third parameter. We use JavaScript function wrapper technique to develop our data collection tool and implement the protection approach.

3. DANGERNEIGHBOR ATTACK

When multiple receivers from different providers exist in the same hosting page, if there is a malicious provider, all the postMessage data may be leaked. We call such attack DangerNeighbor attack.

To make the following description more concise, we define several terms for the rest of the paper:

- ImportedJavaScript: The JavaScript code that is imported into the hosting page via script tag from the provider. It is the same origin with the hosting page, but it is controlled by the provider.

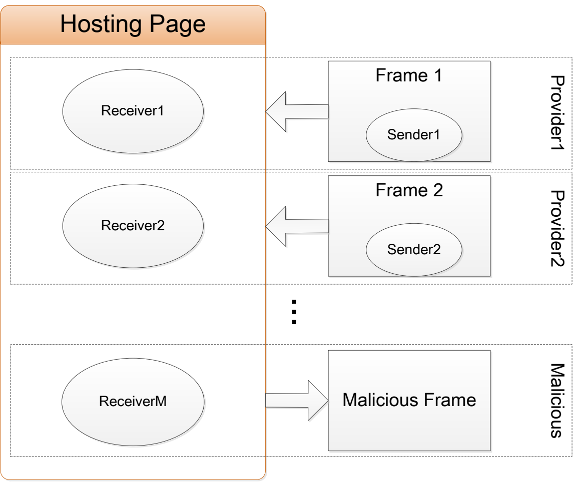


Figure 3: DangerNeighbor attack example.

- Receiver: For simplicity, we call the receiver function mentioned in Section 2.2 as the receiver in the following paper. A hosting page may contain many receivers. The receiver is usually included in the ImportedJavascript.

- Source: The triple consisting of scheme, host, and port of the ImportedJavascript file. It can be extracted from the *src* attribute of the script tag. It can be treated as the source of the receivers in the ImportedJavaScript.

- Frame content: The content in the frame from third-party service provider. It has different origin with the hosting page.

- Message: The text message transferred in the postMessage mechanism, namely the data property of the event parameter of the receiver.

- Origin of the message: The origin of the message sender, namely the origin property of the event parameter of the receiver.

3.1 A New Passive Attack

When a hosting page contains multiple receivers from different providers, the malicious provider can obtain all messages by eavesdropping. The browser delivers the message to the hosting page when a provider frame sends message to the hosting page. While all the receivers in the ImportedJavaScript has the same origin with the hosting page, if there are multiple receivers in the hosting page, all the receivers can receive the message. That means if a site owner imports multiple frames from different providers, the provider's receivers in the ImportedJavascript can receive each other's message. Figure 3 describes such a scenario. The hosting page imports the Frame1, Frame2, and the malicious Frame. The receiver1, receiver2, and receiverM run in the origin of the hosting page. The receiverM collects the messages and sends them to the malicious frame. The HTML5 standard suggests that the receiver should check the origin of the message and abandon the message from unknown senders.

Table 1: Configuration of the test.

Name	Domain	IP
Hosting page	*localhost*	127.0.0.1
Alice	*www.alice.com*	
Bob	*www.bob.com*	
Eve	*www.eve.com*	

This is Alice

A frame from http://www.alice.com

sendMessage

This is Bob

A frame from http://www.bob.com

sendMessage

This is Eve

A frame from http://www.eve.com

getMessage: http://www.alice.com says: Nice to meet you

getMessage: http://www.bob.com says: Hello

The host receives message:

Alice says:Nice to meet you

Bob says:Hello

Figure 4: Attack experiment result.

However, Son et al. show that many receivers perform semantically incorrect origin checks or even no origin checks [29]. Therefore, the receivers in the hosting page open the gate for malicious provider to collect the other one's personal data.

We focus on the DangerNeighbor attack and do not consider the general problem caused by JavaScript inclusion. We assume that the host has included a piece of script from the malicious provider. This piece of script is included as an external script from the domain of malicious provider, which manages to inject malicious script into the hosting page. Afterwards, the malicious script seizes all the privileges of the hosting page to steal the sensitive data. As a general problem in Web security, it has been well studied [23, 26, 31]. However, as a specific JavaScript inclusion problem on postMessage mechanism, the DangerNeighbor attack has not been clearly defined and studied.

3.2 Verification of DangerNeighbor Attack

To verify the existence of DangerNeighbor attacks, we perform a real experimental study. We deploy a Web server that hosts four pages. By modifying the host file on the user's operation system, we make four different domains directed to the Web server so that the pages have different origin in the browser. Table 1 shows the domains and gives each of them a different name. The hosting page imports the other pages as a frame. Eve is the malicious provider. Alice will send "Nice to meet you" to the hosting page when the *sendMessage* button is clicked. It is the same with Bob except that the sending message is "Hello". Eve's receiver collects all the messages sent to the hosting page and sends them to his frame page. Figure 4 shows the experimental results. The hosting page receives the messages and displays the messages in the bottom of the page. As a malicious provider, Eve collects the messages in the meantime and displays the messages in his frame. Eve may choose to send the messages to his server using AJAX technique [2] instead of displaying the messages.

4. EVALUATION OF DANGERNEIGHBOR ATTACK

4.1 Evaluation Framework

We evaluate the threat of DangerNeighbor attack in the real world. Figure 5 demonstrates the outline of our data collection framework. We implement a tool to automatically collect the receivers and the messages, which is saved to a MySql database [10]. We collect the data in the home page of Alexa top 5000 sites and customers' home page of GIGYA, which is a customer identity management service provider. After the data collection step, we process the data on the server with machine learning method and analyze the processed data. The statistics of the data show that the DangerNeighbor attack scenario is very common in the real world. The analysis of the message data confirms that the message data may expose user's privacy data.

4.2 Data Collection

4.2.1 Outline

We implement a tool to automatically collect the receivers and the messages. We also deploy a server to cooperate with the tool. The server is configured with a list of URLs and save the receiver-message data collected by the tool to a MySql database. Our data collection tool is a Mac OS X application based on the same technique with the RVSCOPE [29]. RVSCOPE is an automatic receiver collection tool, as an extension to the Chrome browser augmented with a Web proxy application. The advantage of RVSCOPE 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. Our data collection tool collects the receivers with the same technique as the RVSCOPE. Additionally, we attach a new receiver to the hosting page in our tool to collect the messages.

The main component of our tool is a webView component. The application requests the URL from the server and then gets the Web page content of the URL first. After injecting a script tag on the head of the Web page, the webView loads the modified content. The JavaScript code on our server is imported via the script tag and collects the receivers and messages when the modified Web page is loaded. By wrapping the *addEventListener*, the injected JavaScript code collects the receiver information in the Web page. The injected JavaScript code also attaches a new receiver to the Web page so that we can collect the messages. After collecting the receivers and the messages, the injected JavaScript code reports the collected data to the native part of the application, which sends the collected data to our server. The server is configured with all the URLs to be processed and save the data collected by our tool to the SQL database for further analysis.

We did not implement the tool as a Chrome extension [4] as the RVSCOPE did. The reason is that we cannot wrap JavaScript function of the original site with Chrome extension. The code injected by Chrome extension runs in an isolated world in Chrome browser and cannot interact with the original code in the Web page. The RVSCOPE developer did not explain how to bypass this restriction.

Listing 3: Get stack information.

```
1  function getStack () {
2    var str;
3    try {
4      //Throw a new exception
5      throw new Exception();
6    } catch (e) {
7      //Get the stack information
8      str = e.stack;
9    }
10   return str;
11 }
```

Listing 4: Stack information in JavaScript exception object.

```
1  getStack@http://sss.dcs-ssg.net/
       experiment/static_collect.js
       :27:18
2  addEventListener@http://sss.dcs-ssg
       .net/experiment/static_collect.
       js:83:43
3  http://ir.ebaystatic.com/rs/v/
       bbny5kmf3e0ijcb1f5aeubvbmu5.js
       :264:59
4  global code@http://ir.ebaystatic.
       com/rs/v/
       bbny5kmf3e0ijcb1f5aeubvbmu5.js
       :264:109
```

4.2.2 Collecting the Receivers and the Messages

We collect the receivers and the messages by injecting a piece of JavaScript code. We get the Web page content with a customized HTTP request first. Then we inject a script tag on the head of the Web page. The script tag imports a piece of JavaScript code which we call the injected JavaScript code in the Web page. The injected JavaScript code attaches a new receiver so that we can collect the messages, since the messages are broadcast to each receiver in the origin. We collect the message data and the origin of the sender. It also wraps the *addEventListener* method of the window object to do some additional work when a new receiver is attached. Generally, all the receivers are attached by the *addEventListener* method. So every time the Web page attaches a new receiver to the window object, our method is called and the receiver data will be collected and reported to the native part of our tool.

In the augmented *addEventListener*, we collect the receiver information in detail. We throw out an exception and catch it immediately. From the stack information included in the exception object, we can extract the source of the receiver and the JavaScript filename, the line number, and the column number where the receiver is attached. Listing 3 shows how to get stack information.

Listing 4 shows the stack information we collect from Ebay's home page on Chrome browser. Each line is a piece of stack frame. Each stack frame consists of four parts: the function name, the URL of the JavaScript file, the row number, and the column number. The text before character "@" is the function name in which the exception is thrown out. This part is empty if the function is an anonymous function. The third line of the Listing 4 is an example. The text between character "@" and ":" is the full URL of the

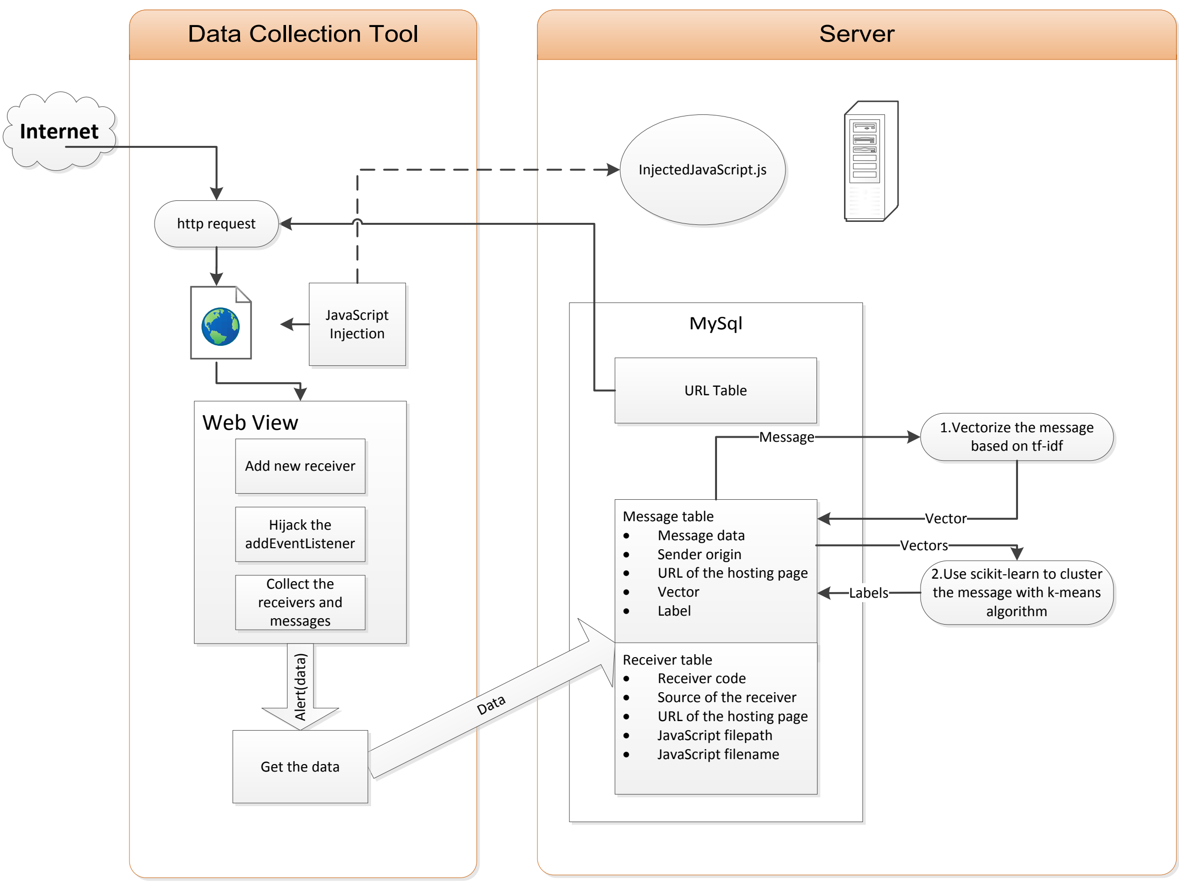


Figure 5: Data collection Framework.

JavaScript file. A full URL consists of scheme, host, port, file-path, and filename. By default the port is 80 and ignored. The numbers between two ":" are the row number and the column number of the JavaScript code. They indicate the exact position where the exception is thrown out. All the source information of the receiver can be found in the second part.

4.2.3 Sending the Data to the Server

The injected JavaScript code has the same origin with the Web page, so it cannot send the collected data to our server that has a different origin with the Web page. In order to send the collected data to our server we handle the alert event of the webView in the native part and alert the collected data in the injected JavaScript code. The native part assembles the collected data from the alert and sends it to our server.

4.2.4 Preprocessing of the Data

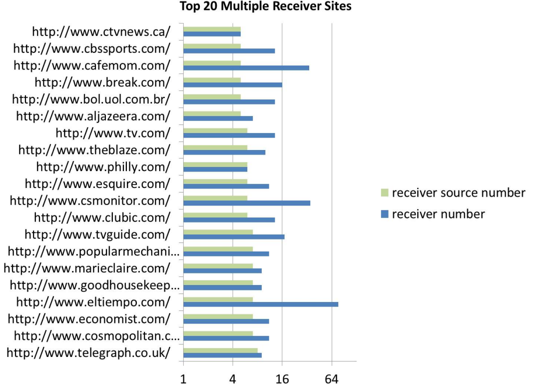
We process the message data using machine learning method to first filter out the similar messages. We vectorize every message by calculating the tf-idf [13] value of each word of the message so that the message data can be handled with machine learning algorithm. Term frequency-inverse document frequency (tf-idf) is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. We cluster the vectorized message data into a number of (e.g.,500) categories following the k-means algorithm. This is implemented with a development toolkit on machine learning called Scikit-Learn [11], which is written in Python. The clustering operation gives every message a label to identify the category of the message. To make the result more precise, we conduct regular matching filterings before and after the clustering operation. Regular matching filtering before the clustering operation filters out the most common messages. They are no more than 10 types, while accounts for more than half of the data set. After the clustering operation, the messages are separated into several hundred categories and we manually process the result with regular matching filtering to make it more precise.

4.3 Statistics of the Usage of postMessage

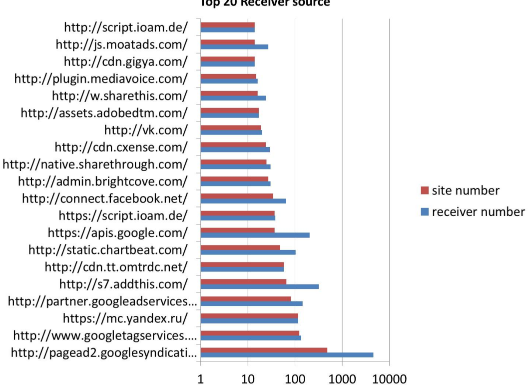
After the data collection discussed in Section 4.2, we get the data of the receivers and the messages in the corresponding sql tables. The receiver table and message table in Figure 5 show the design. We use SQL query to get the statistic data. We collect the data of the home pages of Alexa Top 5000 websites and the customers' home page of GIGYA [5]. GIGYA is a customer identity management service provider.

4.3.1 Top 5000 Sites

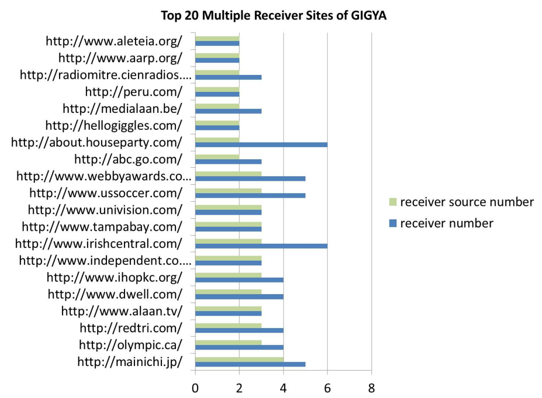
Receivers: We collect 7807 receivers from 1440 sites and the receivers belong to 752 different sources. 867 sites have multiple receivers and among them 544 sites import receivers



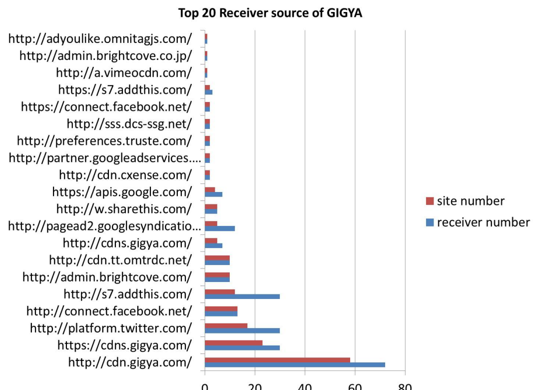
(a) Top 20 multiple receiver sites.



(b) Top 20 receiver source.



(c) Top 20 multiple receiver sites of GIGYA.



(d) Top 20 receiver source of GIGYA.

Figure 6: Statistics of multiple receiver functions and sources.

from more than one sources. The average source number is 1.63 ignoring the sites without receiver. Figure 6(a) shows the top 20 multiple receiver sites. Figure 6(b) shows the top 20 sources. The most common source is http://pagead2.googlesyndication.com. Its receivers appear 4585 times in the top 5000 sites. The service is used by 479 sites. Its data is much larger than other sites.

Messages: We collect 15457 messages from 620 different sites. After clustering operation, the messages are clustered into 163 categories. It is about a quarter of the number of source number of the receivers. The reason that message category number is much smaller than source number is that the same ImportedJavascript may be served on different server. The source URL number of the same ImportedJavascript is usually more than one.

4.3.2 Customers of GIGYA

GIGYA is a customer identity management service provider and it adopts postMessage in its library. GIGYA lists its main customers in its site. Following this list, we get the distinguished home pages from 562 customers. Among the 562 sites, 183 sites have receivers in their home page. 95 sites have multiple receivers and 53 sites import receivers from more than one source among them. The average source number is 1.36 ignoring the sites without receiver. Figure 6(c) shows the top 20 multiple receiver sites. Figure 6(d) shows the top 20 sources.

4.3.3 Analysis

From the statistic data, we can figure out that DangerNeighbor Attack scenario is common in the real world. About one third of the sites take advantage of the postMessage mechanism. About a quarter of them contain receivers from different sources. The top sites listed in Figure 6a have a great risk of the DangerNeighbor Attack. The top sites listed in Figure 6b are more likely to conduct the DangerNeighbor Attack, since they can gather more information than other providers.

We manually classify the message data into to five groups: Remote Procedure Call, ACK of the request, IDs, configuration information, and the others. The remote procedure call consists of function name and arguments. For example: a JSON string {*"type":"hideById","elementId":"showroomSide"*} is a remote procedure call message. The example is a function call that hides an element and the argument is the elementID. There are many ACK messages that act as the response to the remote procedure call message. Various IDs present themselves in the message data. We find various identity information such as userID, videoID, frameID, callbackID. These identity information can leak user's privacy data. For example, the malicious provider can know which user skips which advertisement and watches which video. The developer also uses postMessage to transfer the configuration information of the frame. The message contains the frame width, height, position and so on. There are the other less common types of message. These messages may contain

various temporary tokens or keys. We even found an OAuth token in a site, but it only appears several days.

5. COUNTERMEASURE

The statistics demonstrate that the DangerNeighbor attack scenario is common in the real world. The problem can be solved by modifying the HTML5 standard API - replacing the receiver origin parameter with the receiver ID. The browser can identify the receiver function by the new parameter for the message. However, as a completed standard, HTML5 should not be modified frequently. To solve this problem, we propose a simple but effective protection approach. The site owner can protect the messages from being eavesdropped by importing a piece of JavaScript code on the head of the HTML file and configuring a message-receiver table to declare the intended receiver for a message. It is transparent for the provider and the site owner does not need to modify the original JavaScript code in the site. Alternatively, we give an protection approach on the provider side when the site owner does not allow any changes.

5.1 On Site Owner Side

DangerNeighbor attack misuses the hosting page origin, so the hosting page is an ideal place to thwart the DangerNeighbor attack. The target of the protection approach is to deliver the message to the correspond receiver instead of broadcast the message in the whole origin. To achieve the target, the site owner needs to solve two problems: how to distinguish the receivers and how to customize the message delivery.

5.1.1 *Customizing the Message Delivery*

To customize the message delivery, the site owner can wrap the *addEventListener* method. The site owner replaces all the receivers with the new receiver when the *addEventListener* method is invoked. We name the new receiver *filter*. When the filter receives message, the site owner decides if the receiver should handle the message by checking the message-receiver table. Only when the receiver source and message origin match in the table, the original receiver is invoked with the message as the argument. Listing 5 shows the function wrapping of *addEventListener*.

5.1.2 *Distinguishing the Receivers*

We have defined the source of the receiver in Section 3 and discussed how to get the source of the receiver in Section 4.2.2. Now we can use the source to identify the receiver. We note that the stack information is different from one browser to another browser, so we develop a specific regular expression for each browser to extract useful information.

5.1.3 *Potential Attacks against the Approach*

With the technique discussed above, we can deliver the message to the target receiver. Similar approaches have been discussed in [28, 24, 25]. They aim at protecting the host pages from being attacked by the malicious third-party JavaScript inclusion. They wrap the builtin function with a wrapper and add security check in the wrapper. The malicious JavaScript inclusion is a general problem in the JavaScript protection. In this paper, we focus on protecting the message in postMessage mechanism from being leaked

Listing 5: AddEventListener Wrapper.

```
1   (function(old_EventListener){
2     //Wrap the addEventLister function
3     window.addEventListener =
4       function(type, listener, useCapture)
            {
5         if(/message/i.test(type)) {
6         //Only replace the postMessage
              receiver
7
8       var listenerSourceMap = {};
9           newReceiver = function(event) {
10        //Get the source of the receiver
11        listenerSourceMap[listener] =
              getSource();
12        //Check the message-receiver table
13            var allow = checkTable(
                  listenerSourceMap[listener
                  ], event.origin);
14            if (allow)
15              listener(event);
16          };
17          return old_EventListener.apply(
              this,
18            [type, newReceiver, useCapture
                ]);
19        }
20        return old_EventListener.apply(
            this,
21          [type, listener, useCapture]);
22      }
23  })(window.addEventListener);
```

to the malicious third-party JavaScript. Based on the attacks inferred in [28, 24, 25], we list potential attacks to our approach as follows.

Incomplete Mediation

In JavaScript protection, the approach needs to enforce the security policy to the builtin functions by wrapping them. However, the functions in JavaScript can be access by many different means. For example, the developer may define a user-defined alias for a builtin function. The user-defined alias is beyond the protection and the attacker can bypass the protection by calling the alias to invoke the builtin function. This is the major reason that we emphasize the protection code must be run before any other JavaScript code. If the protection code runs first, even the developer defines an alias for the addEventListener function, the alias can still be protected.

Another incomplete mediation is that the browser may provide several different means to invoke the same function. For example, the ObjectA.call() is the same with ObjectA.prototype.call() in some cases. If the protection approach only wraps the ObjectA.call(), the attacker can invoke the original function by using ObjectA.prototype.call(). Fortunately, it is not a big problem. In our approach, we only wrap the addEventListener method of window object. The other DOM object also has an addEventListener method; however, the browser only triggers the MessageEvent on window object when receiving a message. The receiver attached to the other DOM objectd cannot handle the message. The window object has no prototype property. The addEventListener cannot be invoked by window.prototype.addEventListener.

The attacker can also get the original builtin function from

the other same origin frame. First, the attacker creates a new frame with the same origin as the host page. Second, he gets the reference of the new frame. Third, he can obtain the original addEventListener via the contentWindow of the reference. In [28, 24, 25], they enforce a new policy to prevent the attacker from creating new frames and make sure that all the existing frames deploy the protection approach. However, enforcing a new policy may incur several new problems in our approach. Thus, we do not provide protection against this attack. Fortunately, it can be easily detected by static analysis.

Incomplete Policy

In JavaScript protection, the policy is designed by the developer. If the policy is not robust enough, the protection will not work well. While in our approach, there is only one policy: Only the authorized receiver can handle the message from certain origin. The policy in our approach is simple and easy to enforce completely.

Parameter Type Forgery

JavaScript can call the toString method to convert an argument to string type automatically when the conversion is needed. The attacker may overwrite the toString method and return a different value when it is invoked as the security check parameter. For example, the attacker wants to change the url property of the location object. The wrapper will check whether the new url is in the white list to decide whether the location should be set to the new value. Instead of passing a string for the parameter url, the attack can pass an object with a malicious toString method that return a different value each time it is invoked. Its first invocation occurs at whitelist[url], where it tricks the whitelist by returning a safe URL. The second invocation of toString can return a different value not in the whitelist.

Parameter type forgery is not a problem in our approach, which has only two parameters: event type string and the receiver function reference. Counterfeiting the the event type string has no meaning, because it is used just once. JavaScript does not do conversion for the function type parameter. As a result, the receiver function reference cannot be forged.

Root Prototype Poisoning

All JavaScript objects inherit basic properties and methods from the Object prototype. The attacker can add a field to the Object prototype: Object.prototype['http://www.evil.com']='http://www.alice.com'. As a result, the check on messageReceiver['http://www.evil.com']=='http://www.alice.com' evaluates to true. Similar attacks can target the Function prototype and other globals.

Since we use a message-receiver table in our approach, attackers may target at poisoning the message-receiver table. Fortunately, JavaScript provides the ability to check whether the property is inherited from the prototype. Therefore, we can check the property and make sure the property is not inherited from the root prototype.

Untrusted Parameter Callbacks

The wrapper may accept an untrusted object as the parameter. If the wrapper passes the original builtin function to a method of the untrusted object, the original builtin function is leaked. Our approach does not suffer this attack.

Table 2: Builtin Function or Object Used in Our Approach

name	type
Error	object
String.match	method
Function.apply	method
RegExp	object
RegExp.test	method

Callstack Inspection

JavaScript provides an exploitable form of stack inspection. The attacker can use the caller property of the function object and the callee property of the arguments object to trace its function call stack. If an untrusted function is called in the original builtin function, the untrusted function can get the original function reference by accessing the caller property.

In our approach, the receiver function reference is the only untrusted function. This untrusted function can get the filter function reference by accessing the caller property. It does not cause any damage for the entire approach. The Callstack Inspection is not a threat to our approach.

Builtin Function Hijacking

The approach needs to use some builtin functions or objects to realize the security utility in the wrapper. However, all the builtin functions or objects may be hijacked by the attacker. For example, the wrapper uses string.split(":") to split the string "A:B". Then it checks whether the "B" is in the whitelist. The attacker can hijack the split function and return a fake value to pass the security check.

To solve the problem, the approach should store all the builtin functions or objects in local variables on the first run. When any wrapper needs to use builtin functions or objects, it uses the local variable instead of the builtin reference. Table 2 shows the builtin functions or objects used in our approach.

Delete Operator

JavaScript provides a delete operator to delete unused properties. When the delete operator is applied to the builtin functions, it deletes any wrappers and restores the original. Our solution suffers from this attack. Fortunately, the delete operator is rarely applied on builtin functions. By analyzing the code, it is easy to detect this kind of attack.

5.1.4 Message-Receiver Table

In our approach, the message-receiver table should be in a JSON file named *table.json* on the server. The JavaScript code in the approach will get the table from the server when the hosting page is loaded. The receiver source and the message origin are represented in regular expression. In some cases, the sender of the messages is created dynamic. The map relation cannot be determined beforehand. To solve this problem, we treat the receiver that will process messages from dynamic sender as a special receiver. The message will be delivered to the normal receiver first. If none of the normal receivers matches the message sender, the special receiver will process the message. If there is more than one special receiver, DangerNeighbor attack is available among the special receivers. We leave it as our future work.

Listing 6: Encryption approach.

```
1  (function(){
2    function receiver(event){
3      /*The key should be set when
4        the script is generated.*/
5      var key = "14235123";
6      if(event.origin == "http://
          www.alice.com"){
7        var plaintextMessage = decrypt(
            event.data, key);
8        /*handler the message and generate
9          the response message.*/
10       ...
11       event.source.postMessage(encrypt(
            response,
12         key), "*");
13     }
14   }
15   function encrypt(message,key){
16     //encrypt method. Such as AES
17   ...
18     return cipherText;
19   }
20   function decrypt(message, key){
21     //decrypt method. Such as AES
22   ...
23     return plainText;
24   }
25 }());
```

5.1.5 *Additional advantage*

Our approach can bring additional advantages on solving other security problems of postMessage. The attacker can comprise the hosting page by exploiting the XSS vulnerability caused by the careless usage of the message. This type of vulnerability can be avoided by checking the origin of the message. Our approach ensures all the receivers in the site owner's hosting page check the origin of the message in the wrapped receiver even if the original receiver does not check or mischecks the origin of the message. That means the site owner's hosting page is protected from this traditional problem of postMessage. The site owner can modify our approach and attach an XSS filter in it to filter out the XSS attack payload message.

5.2 On the Third Party Service Provider Side

The third-party service provider can protect their message by encryption. Stark et al. gave a symmetric cryptography library in JavaScript [30]. Bai et al. [17] presented an algorithm of HTML code encryption based on polymorphic JavaScript programs. However, there are a couple of caveats in practical usage.

First, the key is difficult to distribute. Since the ImportedJavaScript code is in the same origin with the hosting page, it has the same privilege with the hosting page code. The ImportedJavaScript code from the malicious provider can do any thing that the other ImportedJavaScript does. The key cannot be distributed in the ImportedJavaScript code. The key should be distributed on the provider's server when generating the JavaScript file. However, generally speaking, the JavaScript file is static file. It is uncommon to generate JavaScript file dynamically. Also, the key should not appear in any global variable so that the ImportedJavaScript from the malicious provider cannot get the key.

Second, the encryption component should not be a global method in case the malicious provider hijacks the encryption component and gets the plaintext message. Consequently, in order to interact with the encryption component, the receiver must be in the same function scope with the encryption component which means each receiver needs its own encryption component. Listing 6 shows how to implement the encryption component in the same function scope with the receiver and adopts the encryption component to protect the message. If the site owner contains a lot of third-party service provider's receivers, the JavaScript code will be even longer. To make things worse, the JavaScript is an interpretive language and the encryption operations will further extend the response time when users visit the site.

6. RELATED WORK

Researchers have studied the security problems in the postMessage mechanism. Barth et al. [18] carried out a comprehensive study of cross-frame communication in Web browsers and demonstrated attacks on the confidentiality of messages sent via postMessage under certain frame navigation policies, including the descendant policy. Since their research on the postMessage mechanism, the HTML5 standard Committee add the origin parameter into the postMessage API. We demonstrated that such modification failed to make the postMessage mechanism strong enough to keep the messages safe in this paper. In this paper, we advise the origin parameter of the postMessage API should be augmented with the receiver ID.

Steve et al. [20] analyzed the uses of postMessage in Facebook Connect and Google Friend Connect, and showed how incomplete origin checks and guessable random tokens compromise message integrity and confidentiality. Son et al. [29] found a large number of legitimate scripts that use postMessage incorrectly and can be exploited because of flawed origin checks. DangerNeighbor attack is different from the attacks in these papers. It is a passive attack and more difficult to detect. Our approach not only defeats DangerNeighbor attack, but also help thwart the active attack referred in those papers.

The root of DangerNeighbor attack problem is that the ImportedJavaScript code from the provider runs in the same origin as the hosting page. ScriptInspector [31] is a modified browser that can intercept, record, and check third-party script accesses to critical resources against security policies. However, it failed to detect the DangerNeighbor attack when we try it in our experiment. Research work has been done to confirm that JavaScript inclusion may cause vulnerabilities in the website. Nikiforakis et al. [26] performed a large-scale survey of the JavaScript inclusions and it gave the conclusion that even the top Internet sites may trust remote providers which can be comprised by attackers easily. Lekies et al. [23] discussed the problem of JavaScript inclusions from another perspective: the hosting page attacks the remote JavaScript. The hosting page includes the remote JavaScript and gets user's privacy data owned by the remote provider. Our approach bypasses this problem by identifying the provider's code and isolating the receivers from each other.

The protection of the privacy in the Web environment is a major issue in the Web security. Jang et al. [21] found 43 instances of privacy-violating information flows in the Alexa top sites. Gervais et al. [19] provided a quantitative methodology for evaluating users' web-search privacy. Acquisti et

al. [16] investigated individual privacy valuations in a series of experiments informed by theories from behavioral economics and decision research. Olejnik et al. [27] performed a privacy analysis of Cookie Matching and Real-Time Bidding and quantified the leakage of users' browsing histories due to these mechanisms. In this paper, DangerNeighbor attack is a new attack to obtain user's privacy data. We confirmed that the messages delivered by postMessage indeed carry user's privacy data.

The Same-Origin Policy (SOP) is the principal security policy on the browser. The postMessage gives a legitimate way to break the SOP. Unfortunately, it results in the problem in the [18, 20, 29]. Similarly, when the smart phone encounters the SOP, new problems emerge. Hybrid application [7] is a native mobile applications, which involves Web based technologies such as JavaScript, HTML, etc. The frameworks used to develop hybrid application are called hybrid frameworks. Son et al. [29] analyzed the software stack created by hybrid frameworks and demonstrated that it does not properly compose the access control policies governing Web code and local code, respectively. Son et al. studied the scenario that the Web codes in the hybrid application access the local resource without authorization. Jin et al. [22] studied the hybrid application from another perspective. They found a new form of code injection attack, which inherits the fundamental cause of Cross-Site Scripting attack (XSS), but it uses many more channels in hybrid application to inject code than XSS. Our work on the DangerNeighbor attack is inspired by the research on the hybrid application.

7. CONCLUSION

In this paper, we identify a new DangerNeighbor attack misusing the HTML5 postMessage mechanism. This passive attack enables a malicious third-party service provider to eavesdrop the messages between other service providers and the hosting page. We develop a toolset to evaluate the threat of the DangerNeighbor attack in the home pages of the Alexa top 5000 sites and a customer identity management website. The experimental results show that the DangerNeighbor attack is common in the real world, particularly, on deriving user privacy data from the various leaked identity information. To thwart the DangerNeighbor attack, we propose a solution on the site owner side using the JavaScript function wrapper technique. Alternatively, we propose an encryption-based protection approach on the third-party service provider side. We also suggest that the postMessage API in HTML5 should be modified to avoid DangerNeighbor attack.

8. ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant 61272479, the National 973 Program of China under Grant 2013CB338001, the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDA06010702. Kun Sun's work is supported by ONR grants N00014-15-1-2396 and N00014-15-1-2012.

9. REFERENCES

[1] Alexa, top sites on the web. http://www.alexa.com/topsites.

[2] Asynchronous javascript and xml. http://www.w3schools.com/ajax/ajax_intro.asp.

[3] Chrome browser from google. http://www.google.cn/intl/en/chrome/browser/desktop/index.html.

[4] Chrome extension. https://developer.chrome.com/extensions.

[5] Gigya, a customer identity management service provider. http://www.gigya.com.

[6] Html5.1. http://www.w3.org/TR/html5/.

[7] Hybrid frameworks. https://developer.jboss.org/wiki/GetStartedWithHybridApplicationFrameworks?_sscc=t.

[8] Javascript object notation. http://www.w3schools.com/json/.

[9] Mozilla, firefox browser. http://www.mozilla.org.

[10] Mysql. a popular open source database. http://www.mysql.com/.

[11] A python module for machine learning built on scipy and distributed under the 3-clause bsd license. http://scikit-learn.org.

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

[13] Tf-idf:term frequency and inverse document frequency. https://en.wikipedia.org/wiki/Tf-idf.

[14] Top 5 security threats in html5. http://www.esecurityplanet.com/trends/article.php/3916381/Top-5-Security-Threats-in-HTML5.htm.

[15] Xss, "cross-site scripting". owasp. https://www.owasp.org/index.php/XSS.

[16] A. Acquisti, L. K. John, and G. Loewenstein. What is privacy worth? *The Journal of Legal Studies*, 42(2):249–274, 2013.

[17] Z. Bai and J. Qin. Webpage encryption based on polymorphic javascript algorithm. In *Proceedings of the Fifth International Conference on Information Assurance and Security, IAS 2009, Xi'An, China, 18-20 August 2009*, pages 327–330, 2009.

[18] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 17–30, 2008.

[19] A. Gervais, R. Shokri, A. Singla, S. Capkun, and V. Lenders. Quantifying web-search privacy. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 966–977, 2014.

[20] 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 *Proceedings of the Web*, volume 2, 2010.

[21] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 270–283, 2010.

[22] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 66–77, 2014.

[23] S. Lekies, B. Stock, M. Wentzel, and M. Johns. The unexpected dangers of dynamic javascript. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 723–735, 2015.

[24] J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers*, pages 239–255, 2010.

[25] L. A. Meyerovich, A. P. Felt, and M. S. Miller. Object views: fine-grained sharing in browsers. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 721–730, 2010.

[26] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 736–747, 2012.

[27] L. Olejnik, M. Tran, and C. Castelluccia. Selling off user privacy at auction. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[28] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia, March 10-12, 2009*, pages 47–60, 2009.

[29] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postmessage in HTML5 websites. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[30] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in javascript. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, pages 373–381, 2009.

[31] Y. Zhou and D. Evans. Understanding and monitoring embedded web scripts. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 850–865, 2015.