Reconstructing Interactions with Rich Internet Applications from HTTP Traces

Sara Baghbanzadeh, Salman Hooshmand, Gregor Bochmann, Guy-Vincent Jourdan, Seyed Mirtaheri, Muhammad Faheem, Iosif Viorel Onut

Chapter 8

RECONSTRUCTING INTERACTIONS WITH RICH INTERNET APPLICATIONS FROM HTTP TRACES

Sara Baghbanzadeh, Salman Hooshmand, Gregor Bochmann, Guy-Vincent Jourdan, Seyed Mirtaheri, Muhammad Faheem and Iosif Viorel Onut

Abstract This chapter describes the design and implementation of ForenRIA, a forensic tool for performing automated and complete reconstructions of user sessions with rich Internet applications using only the HTTP logs. ForenRIA recovers all the application states rendered by the browser, reconstructs screenshots of the states and lists every action taken by the user, including recovering user inputs. Rich Internet applications are deployed widely, including on mobile systems. Recovering information from logs for these applications is significantly more challenging compared with classical web applications. This is because HTTP traffic predominantly contains application data with no obvious clues about what the user did to trigger the traffic. ForenRIA is the first forensic tool that specifically targets rich Internet applications. Experiments demonstrate that the tool can successfully handle relatively complex rich Internet applications.

Keywords: Rich Internet applications, user session reconstruction, HTTP logs

1. Introduction

Over the past few years, an increasing number of application developers have opted for web-based solutions. This shift has been possible due to the enhanced support of client-side technologies, mainly scripting languages such as JavaScript [11], asynchronous JavaScript and XML (Ajax [13]).

JavaScript enables application developers to modify the document object model (DOM) [23] via client-side scripting while Ajax provides asynchronous communications between scripts executing on a client browser and web server. The combination of these technologies has created rich Internet applications (RIAs) [12]. These web-based applications are highly dynamic, complex and provide users with an experience similar to "native" (i.e., non-web) applications.

Typically, in a web application, the client (i.e., user web browser) exchanges messages with the server using HTTP [10] over TCP/IP. This traffic is partially or entirely logged by the web server that hosts the application. The traffic is easily captured as it traverses a network (e.g., using a proxy). The captured traffic generated by a user during a session with a web application is called a user session log.

A user session log can be used to reconstruct user browser interactions for forensic analysis after a incident has been detected. For example, suppose the owner of a web application discovers that a hacker had exploited a vulnerability a few months earlier. The system administrator is tasked to find out what happened and how it happened using the only available resource – the server-generated logs of previous user sessions. This task would not be too challenging for a classical web application because tools have been developed for this purpose (e.g., [18]). However, if the web application is a modern rich Internet application, then manual reconstruction would be extremely difficult and time consuming, and no tools are available that could help with the task.

This chapter describes the design and implementation of ForenRIA, a tool intended to help recover information about an intrusion using the available logs. ForenRIA reconstructs screenshots of user sessions, recovers user inputs and all the actions taken by a user during a session. The tool satisfies two important design goals with regard to digital forensics [5]. First, as a result of security concerns, the forensic analysis should be sandboxed; in other words, connections to the Internet are minimized. Second, the reconstructed pages are rendered as closely as possible to the pages viewed at the time of the incident. The first goal is achieved by the offline replay of traffic in a browser. The second goal is met by preserving the correct state of the document object model.

This research was conducted in collaboration with the IBM QRadar Incident Forensic Team. When it comes to recovering user interactions from HTTP logs of rich Internet applications, QRadar forensics relies on a mix of tools and expert knowledge. Thus, the research goal was to develop a general and fully-automated method for inferring user interactions with complex rich Internet applications.

The ForenRIA tool described in this chapter automatically reconstructs user interactions with rich Internet applications using only previously-recorded user session logs as input. Efficient methods are proposed to find user actions; these are based on the notions of "early-action" and "non-existent" clicks that prioritize candidate clicks based on the event history. New techniques are described for inferring user inputs from HTTP traces. Also, a method is presented for handling random parameters in requests during offline traffic replay.

2. Session Reconstruction Methodology

This section describes the user session reconstruction methodology and its implementation in the ForenRIA tool.

2.1 Inputs, Outputs and Assumptions

The goal of this research was to automatically reconstruct entire sessions involving user interactions with rich Internet applications using only HTTP logs as input. During an interaction with a rich Internet application, a user is presented with the rendering of the current document object model by the browser. Some of the user actions (e.g., mouse moves and clicks) trigger the execution of client-side code. In this work, the execution of this code is referred to as an "event."

Events are usually triggered by user actions on HTML elements or automatically by scripts, for example, after a delay or timeout. Some of these event executions trigger requests back to the server. Some of these requests are synchronous while others are asynchronous (i.e., a response might arrive later, and several other requests/responses might occur in the meantime). It is also common for a user to provide inputs via menu selections or as text. These inputs are typically sent along with some requests and the responses that are returned usually depend on the user inputs. Thus, reconstructing user interactions involves the recovery of the series of document object models rendered by the browser as well as all the events executed during the session. Typically, for each event, it is necessary to know the type of event (click, mouse hover, etc.) as well as the XPath of the HTML element on which the event occurred. It is also necessary to recover all user inputs, including the values that were provided and where the values were provided.

It is relatively straightforward to reconstruct user interactions from the log in a "traditional" web application because the data sent by the server is more or less what was displayed by the browser. The situation is very different for rich Internet applications where many requests are generated by script code running on the browser and the responses typically contain only a small amount of data used by the receiving scripts to partially update the document object models (Figure 1 shows an example). Thus, many of the request-response pairs are part of a series

```
{"soapenv:Body":{"response":{"returnValue":{"value":{"ImplClassName":
"com.ibm.team.service.jts.internal.mailer.MailerService","properties":[{
"defValue":"false",...,"eQClassName":"http:\/\/schemas.xmlsoap.org"}}}}}}
```

Figure 1. Body of a typical HTTP response.

of interactions and cannot be analyzed in isolation. Consequently, it is difficult to infer all the steps taken by the user, the inputs provided and the elements that were clicked when the only information provided is in the recorded user session log. The ForenRIA tool described in this chapter addresses these challenges and obtains the details of a security incident by automatically recovering every user interaction.

The input to the tool is a sequence of previously-captured HTTP traffic $\{t_1, t_2, ..., t_n\}$ where t_i is the i^{th} request-response pair. It is necessary to identify the set of actions $\{a_1, a_2, ..., a_m\}$ performed by the user. If the execution of a_i generates a set of requests T_i, then the requests in T_i should directly follow T_{i-1} in the input log.

The following information is of interest with regard to the outputs:

- **User Actions:** The precise sequence of actions performed by a user during a session (clicks, selections, etc.) and the exact elements of the document object models on which the actions were performed.

- **User Inputs:** The exact inputs provided by a user during a session and the fields in which the inputs were provided.

- **Document Object Models:** The series of document object models that appeared in a user's browser during a session, including information such as the values and parameters of cookies.

- **Screenshots:** The series of screens that were displayed to a user.

It is assumed that the input is the log corresponding to a single user. Extracting the trace for an individual user from server logs is a well-studied problem (see e.g., [21]) that is outside the scope of this research. In the context of this research, the traffic has already been recorded and no additional instrumentation of the user's browser is possible. In addition, the reconstruction is performed offline with no access to the original rich Internet application.

2.2 Architecture and Approach

Figure 2 presents the architecture of the ForenRIA tool. The input is a previously-recorded user session access log with two instances of

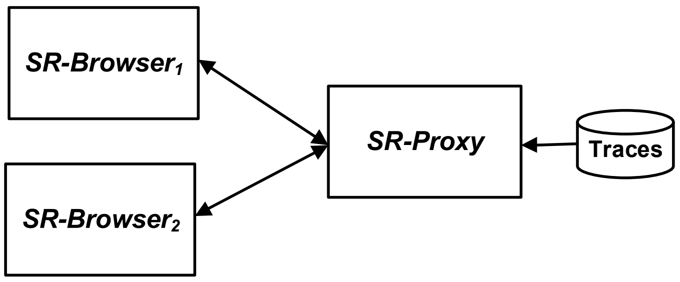


Figure 2. ForenRIA tool architecture.

a programmable browser (SR-Browser$_1$ and SR-Browser$_2$) and a proxy (SR-Proxy) that reconstructs user interactions.

The two browsers re-enact the user actions and the proxy plays the role of the original web server. When the browsers generate the correct actions, the proxy receives a set of requests that match the pre-recorded requests and sends the corresponding responses. Otherwise, an error is returned and another action is selected by the browsers. Note that the two browsers implement the same deterministic algorithm and, thus, always select the same actions.

Algorithm 1 : SR-Browser algorithm.

```
1: while not finished do
2:     LoadAndInstrumentLastKnownGoodDOM(ReconstructedActionsList);
3:     ActionCandidate = ChooseAndExecuteNextAction();
4:     WaitForStableCondition();
5:     if SequenceCheck() then
6:         ReconstructedActionsList.push(ActionCandidate);
7:     end if
8: end while
9: return ReconstructedActionsList;
```

Algorithm 1 specifies the steps executed by SR-Browser$_1$ and SR-Browser$_2$. At each step, a browser reloads the last known-correct state (or the initial URL) if necessary, and instruments the document object model by overwriting JavaScript functions such as `addEventListener` and `setTimeout` to control the subsequent execution of JavaScript code (Line 2). Next, it is necessary to find the next user interaction candidate. To decide efficiently, SR-Browser and SR-Proxy collaborate on the selection. After the next user interaction candidate (and possibly the corresponding user inputs) is chosen, the corresponding event is triggered by the browser (Line 3). SR-Browser then waits until a stable state is reached (Line 4) and verifies the correctness of the selected action by

Algorithm 2 : SR-Proxy algorithm.

```
1:  while not finished do
2:      req := GetCommonRequestsFromBrowsers();
3:      if req is an "HTTP" request then
4:          if matchFound(req) then
5:              ReturnMassagedResponsereq;
6:          else
7:              Return404NotFound();
8:          end if
9:      else
10:         respondToCtrlMessage(req);
11:     end if
12: end while
```

asking SR-Proxy about the previously-generated sequence of requests (Line 5). If the selected action is correct, then the action is added to the output (i.e., actual user interaction along with the XPath of the element on which it was exercised, possible user input, copy of the current document object model, reconstructed screenshot of the state of the browser, etc.).

Algorithm 2 specifies the steps executed by SR-Proxy. SR-Proxy waits for requests sent by the browser. If the request received is a normal HTTP request (Line 3), then it attempts to find the request in the user logs. If the request is found (Line 4), then the corresponding response is sent to the browser with some modifications (Line 5); otherwise, an error message is returned to the browser (Line 7). If the request is not a normal HTTP request, then it is a control message (e.g., a collaboration message to find the next user interaction candidate or a sequence check) that must be answered (Line 10).

Loading the Last Known-Good State. Frequently, during the reconstruction process, SR-Browser is not in the correct client state. This could be because the reconstruction has not yet started or because SR-Proxy has signaled that a generated request does not match the recorded request. When SR-Browser is not in the correct client state, then it needs to restore the previous state and try another user action. In traditional web applications, it is sufficient to reload the previous URL. However, the situation is more complex with rich Internet applications because the previous state is usually the result of the execution of a series of scripts and interactions with the browser. The proposed approach is to clear the browser cache, reload the initial URL and re-execute all the previously-detected interactions to bring SR-Browser back to the previous state. A

```
XMLHttpRequest.prototype.sendOriginal = XMLHttpRequest.prototype.send;
XMLHttpRequest.prototype.send = function (x){
      var onreadyStateChangeOriginal = this.onreadystatechange;
      this.onreadystatechange = function(){
           onreadystatechangeOriginal(this);
           parent.ajaxFinishNotification();
     }
      parent.ajaxStartNotification();
      this.sendOriginal(x);
}
```

Figure 3. Hijacking Ajax calls to detect when a callback function completes.

more efficient alternative is to implement the technique described in [19] and directly reload the previous document object model.

Thus, the state that is reached needs to be instrumented by overwriting a range of base JavaScript calls in order to fully control what happens next. The basic idea is to overwrite methods such as `addEventListener`, `setTimeout` and `XMLHttpRequest.prototype.send` so that SR-Browser code is executed when these calls are triggered. Figure 3 shows an example involving the function `XMLHttpRequest.prototype.send`. The other methods are overwritten in a similar manner.

Finding the Next User Interaction Candidate. At each step, the browser has to find the correct user action and, possibly, the correct user inputs. There are typically hundreds, if not thousands, of possible events for each document object model, so a blind search is not practical. Thus, SR-Browser and SR-Proxy collaborate to create a list of possible choices in decreasing likelihood as follows:

- **Actionable Elements:** Priority is given to document object model elements that are actionable – these are visible elements with some JavaScript attached. Sometimes, this code is assigned explicitly to an attribute of an element (e.g., the `onclick` attribute of a `div` tag), but in other cases the listener is added dynamically to the element using methods like `addEventListener`. Having overwritten these methods as explained previously, ForenRIA can keep track of these elements as well. The algorithm then prioritizes actionable elements by increasing the distances to leaves in the document object model tree.

- **Explicit Clues:** In some cases, it is easy to guess the next action taken by the user simply by examining the next requests in the log. For example, the user may have clicked on a simple hyperlink and SR-Browser can find an anchor element (i.e., `<a href=...>`) in the current document object model with the corresponding URL

```
window.onload = function(){
     var all_tds = document.getElementsByTagName("td");
     for (var e = 0; e < all_tds.length; e++){
          all_tds[e].onClick = function(){
            id = this.getElementsByTagName('strong')[0].firstChild.nodeValue;
            getAjax("GET", "content/content" + id + ".html", true);
             }
      }
}
```

Figure 4. Example of an `onclick` handler created dynamically.

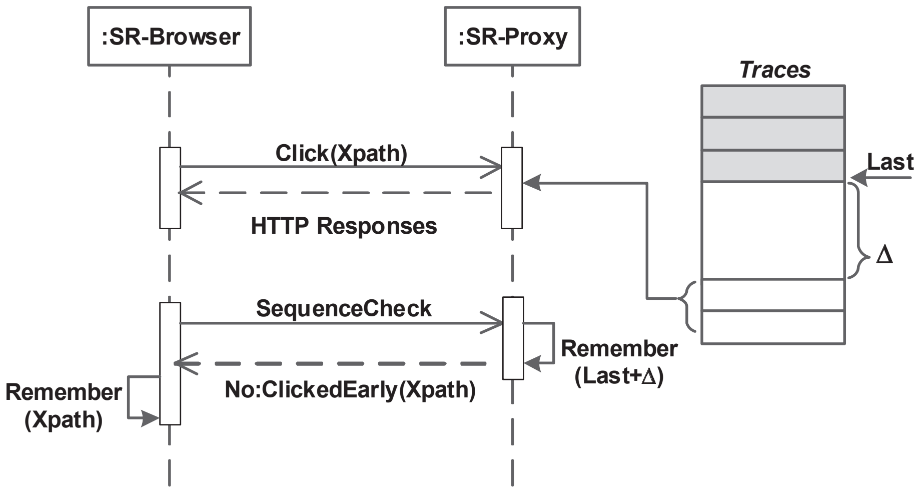
in its `href` tag; or, when an obvious clue is found, such as the parameters of the next URL in the `href` or the `onclick` handler of an element in the document object model. These cases, which were the norm in older web sites, are rarely encountered in rich Internet applications.

- **Implicit Clues:** In a rich Internet application, there are usually no obvious clues about the element that triggered an HTTP request. Figure 4 presents an example involving PeriodicTable, a simple rich Internet application. In this case, a single event handler `onclick` is attached to the common parent of several elements in a document object model. Its execution triggers an Ajax call that dynamically detects the element that was originally clicked. The following steps are taken to tackle such situations:

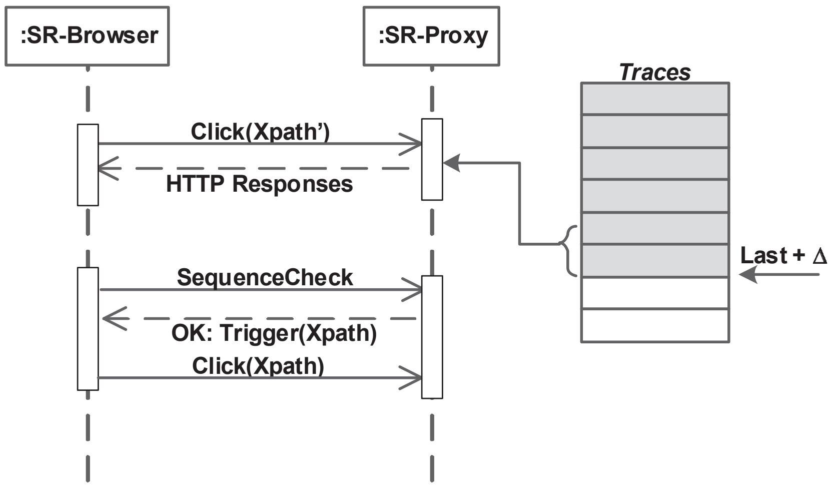
  – *Known JavaScript Libraries:* Many rich Internet applications are built using popular JavaScript libraries such as JQuery, Dojo and Angular. It is often relatively easy to detect the fingerprints of these libraries. Each library has a standard way of generating requests when a user interacts with an element. By implementing code specific to each library, elements can be mapped to clues in the requests dynamically, essentially turning implicit clues into explicit clues.

  – *Early Action:* An incorrect action may be selected during the course of a reconstruction. In some cases, the action on an element e generates a series of unexpected requests (i.e., they are not the next ones in the traces), but these requests still appear as a block later in the trace. This can mean that algorithm has taken the action on e too early. This situation is remembered and the same action on e is taken when the corresponding block is next in the queue.

    Figure 5 demonstrates the situation. In Figure 5(a), the tool has selected an element early and the requests are found in



(a) Requests found in the log further down the queue.



(b) Requests become the next unconsumed requests in the queue.

Figure 5. Early action handling.

the recorded log, but further down the queue. As shown in Figure 5(b), the requests subsequently become the next unconsumed requests in the queue. SR-Proxy then instructs SR-Browser to select the same element. Note that "Last" refers to the last consumed resource in the trace.

  - *Non-Existent Action:* The dual notion of early action is a non-existent action. This is triggered when a request generated by selecting an element e does not exist in the trace. This is an indication that e was not used in the session and the priority of e can be reduced.

- **Finding User Inputs:** User input is an integral part of a web application and recovering the input is necessary for a forensic analysis as well as to perform a successful reconstruction. In web applications, user inputs are usually captured in fields that can be grouped into a form. However, in a rich Internet application, interacting with a single field may trigger network traffic; for example, typing in a search box can load partial results. This is a common behavior that does not follow the HTTP standard for form submission, which involves several input fields being submitted at one time to the server. The standard specifies that the user input values should be encoded as <`name, value`> pairs. To address this situation, SR-Proxy scans the log for occurrences of <`name, value`> pairs (encoded in one of the standard formats). When matches are found, the pairs are sent to SR-Browser, which uses them to populate input fields in the current document object model. If a single value is sent, corresponding to a single input field, an action is attempted on the field (e.g., `onchange` or `onkeyup`). When a set of values is sent back, SR-Browser looks for a form containing input fields with all the given names and attempts to submit the form.

Client-Side Randomness. One difficulty to overcome is randomness. Server-side randomness is not an issue because the same traffic is replayed. However, if the client includes some randomly-generated values in its requests (e.g., random selection of a news feed), then the requests do not match those saved in the log file.

To address this problem, two concurrent instances of SR-Browser, both running the same deterministic algorithm, are used. The two browsers select the same action and the same user input in the same sequence, and send the same requests to SR-Proxy. SR-Proxy compares the requests received from the two browsers. Usually, the requests are identical. However, when client-side randomness is involved, some requests do not match. The parts that do not match are deemed to be randomly generated by the client and are not expected to match the recorded traffic. In such cases, the proxy looks for partial matches in the log file.

Temporal Headers in Responses. HTTP responses have temporal headers such as `date`, `expires` and `last-modified` that may impact replay; this is because the time of replay does not correspond to the time of capture. For example, if a resource has an `expires` header set to time t_1 and it is replayed at time $t_2 > t_1$, then SR-Browser automatically requests the resource again, impacting the replay. Therefore, SR-Proxy updates all the headers using the current time and the relative time offset at the time of recording.

Browser Stable Condition. After executing an event, SR-Browser waits until it reaches a stable condition, a state in which no further changes occur unless a new event is executed. SR-Browser decides if the state is stable or not by overwriting JavaScript functions. In particular, it keeps track of pending requests by overwriting the methods used to send and receive requests (e.g., `send` and `onreadystateschange` methods of the *XMLHttpRequest* object; Figure 3 shows an example). Additionally, some scripts can be executed after a given delay via a JavaScript timeout. In order to keep track of these events, SR-Browser also redefines the built-in `settimeout` method.

Sequence Check. After an element is clicked, if no error is returned by SR-Proxy, then SR-Browser interacts with SR-Proxy to confirm that the clicked element was chosen correctly. One difficulty with this approach is that a single click can generate multiple calls to the server, so several requests and responses can occur after a single click. Another difficulty is that the asynchronous nature of server calls (i.e., Ajax) means that requests may be received in different orders in different runs. Thus, it is not simply a matter of going down a list in the correct order on the SR-Proxy side. Instead, the entire set of requests and responses should eventually form a gapless block in the recorded trace after SR-Browser reaches the stable condition, and this gapless block should be right after the previous block. SR-Proxy checks for this situation upon receiving the sequence check message from SR-Browser after it reaches a stable condition.

Basic Solution. Any system that attempts to reconstruct user interactions with rich Internet applications must be able to handle user input recovery, client-side randomness, sequence checks and previous state restoration; otherwise, reconstruction may not be possible. A basic solution is to perform an exhaustive search on the elements of a document object model to find the next action; this solution does not use the techniques proposed in ForenRIA. A search of the literature reveals that no other published solution has the characteristics of the basic solution. Thus, no other solution can help reconstruct rich Internet application sessions, even in an inefficient manner.

3. Experimental Results

This section presents the experimental results obtained using the ForenRIA tool. The SR-Browser implementation uses the PhantomJS headless programmable browser. PhantomJS supports the recreation of the document object models, the creation of screenshots and the retrieval of the XPaths of elements on which inputs and actions are performed. SR-Proxy is implemented as a PHP application.

The experimental evaluation limited the test cases to rich Internet applications because other tools already perform user interaction reconstruction on non-Ajax web applications (e.g. [18]). The test applications included: OpenCart, an open-source e-commerce solution; OSCommmerce, an e-commerce platform; IBM's Rational Team Concert (RTC), an agile application lifecycle management rich Internet application based on DOJO; El-Finder, an open-source file manager for the web; (v) Engage, a web-based goal setting and performance management application built using the Google web toolkit; Test RIA, Altoro Mutual and PeriodicTable, three simple rich Internet applications built for testing purposes; and a Joomla test site, a popular content management system that uses jQuery. The complete HTTP traffic corresponding to user interactions was captured for each test application using `fiddler` and `mitmproxy` (for sites using SSL/TLS).

The experiments measured the effectiveness of reconstructing user interactions. Two factors were considered: (i) time taken; and (ii) cost (i.e., number of actions required by SR-Browser to reconstruct all the user interactions). The experiments were run on Linux computers with an Intel Core i7 2.7 GHz CPU and 4 GB RAM. The ForenRIA results were compared with those obtained using the basic solution.

Table 1 compares the reconstruction times and costs for ForenRIA and the basic method. Note that Actions is the number of user actions and Requests is the number of HTTP requests in a recorded session. ForenRIA successfully reconstructed the interactions in all the test cases, including recovering all the document object models, the XPaths of the elements on which actions were taken, the user inputs, and the details of all the cookies and screen captures of the rich Internet applications as originally presented to the user. The lengths of the captured sessions ranged from 25 to 150 user actions, with many forms being filled in several complex scenarios. For example, the OpenCart session required 150

Table 1. Reconstruction times and costs for ForenRIA and the basic method.

RIA	**Actions**	**Requests**	**Time (hh:mm:ss)**		**Cost**	
			ForenRIA	**Basic**	**ForenRIA**	**Basic**
OpenCart	150	325	0:10:26	76:10:45	3,221	1,808,250
OSCommerce	150	532	0:02:44	21:23:15	150	501,806
RTC	30	218	0:46:54	50:53:44	1,423	94,242
El-Finder	150	175	0:14:55	07:24:40	12,533	376,820
Engage	25	164	0:31:13	01:47:02	7,834	17,052
Test Ria	31	74	0:00:37	00:22:51	302	15,812
PeriodicTable	89	94	0:07:38	36:20:45	4,453	1,559,796
AltoroMutual	150	204	0:01:41	25:24:30	358	815,302
Joomla	150	253	0:48:20	N/A	344	N/A

actions, including 101 user inputs to register a user, add products to the shopping cart and check out. On average, ForenRIA required 59 events and 23 seconds to find the next action. In contrast, the basic method required on average 5,034 events and 1,503 seconds to find the next action. The basic method on Joomla was stopped after reconstructing only 22 actions because it was too slow. The time required for these 22 actions was 15:24:30 with a cost of 2,274; however, since the basic method was terminated, the corresponding entries in Table 1 are marked as N/A.

In addition, El-Finder and Engage had random parameters in their requests. Without detecting these parameters, it was not possible to even load the first pages of these sites. However, ForenRIA correctly found all the random parameters for these websites, enabling the complete reconstruction of the sessions.

The experimental results demonstrate the effectiveness of the proposed approach for reconstructing user interactions with rich Internet applications. No current tool can reconstruct user interactions only using HTTP traces.

To demonstrate the capabilities of ForenRIA, an attack scenario involving a vulnerable banking application was created by IBM for test purposes. In the scenario, an attacker visits the vulnerable web site and uses an SQL injection attack to gain access to private information and then transfers a considerable amount of money to another account. The attacker also discovers a cross-site scripting vulnerability that could be exploited later against other users.

Given the full HTTP traces, ForenRIA was able to fully reconstruct the attack in just six seconds. The output included screenshots of all the pages seen by the hacker, document object models of all the pages and the steps and inputs used for the unauthorized login, SQL injection attack and cross-site scripting vulnerability. A forensic analysis of the attack would have been quite straightforward using ForenRIA, including the discovery of the cross-site scripting vulnerability. In contrast, the same analysis without ForenRIA would have taken much longer and the cross-site scripting vulnerability would probably have been missed. A demonstration of this case study as well as sample inputs/outputs of the ForenRIA tool are available at `ssrg.site.uottawa.ca/sr/demo.html`.

4. Related Work

A survey paper by Cornelis et al. [6] reveals that deterministic replay has been investigated for many years in a variety of contexts. Some of the systems were designed to debug operating systems or analyze cache coherence in distributed shared memory systems [17, 25]. However, these systems do not handle client-side non-determinism in the parameters sent to the server, which is critical to dealing with rich Internet applications.

The Selenium IDE (implemented as a Firefox plugin) [20] records user actions and replays traffic. Recording can only be done using Firefox, but replays can be performed across browsers using synthetic JavaScript events. A number of methods have been proposed to capture and replay user actions in JavaScript applications (e.g., Mugshot [16] and WaRR [1]). The client-side of Mugshot records events in in-memory logs; Mugshot also creates a log entry containing the sequence number and clock time for each recorded event. The WaRR [1] recorder is also embedded in a web browser and captures a complete interaction trace; it logs a sequence of commands, where each command is a user action. Atterer et al. [3] have used a proxy to inject a script that hijacks all JavaScript events triggered by a user. However, all these approaches require an alteration of the live rich Internet application to record all the user action information needed to recreate the events at replay time. As mentioned above, the goal of the research discussed in this chapter was to reconstruct a user session from raw traces without instrumenting the client or the modifying rich Internet application. Thus, none of the above solutions satisfy the required goal.

Some traffic replay systems [2, 14] have focused on replaying traffic at the IP or TCP/UDP levels. Hong and Wu [14] have implemented a system that interactively replays traffic by emulating a TCP protocol stack. However, these levels are too low for efficient reconstructions of user interactions with rich Internet applications.

WebPatrol [4] has introduced the concept of automated collection and replay of web-based malware scenarios. The prototype has two components, a scenario collection component and a scenario replay component. The scenario collection component automatically gathers malware infection trails using its own "honey clients" and builds a web-based malware scenario repository. The scenario replay component reconstructs the original infection trail at any time from the stored data. WebPatrol is not designed to reconstruct generic rich Internet application interactions from traces; therefore, it cannot be used for this purpose.

Historically, session reconstruction has meant finding the pages visited by users and distinguishing between different users in server logs, often as a pre-processing step for web usage mining (see e.g., [7, 8, 22]). However, the proposed work assumes that individual user sessions have already been extracted from logs, perhaps using one of these techniques.

ReSurf [24] is a referrer-based click inference tool; however, it is of limited use for rich Internet applications. Spiliopoulou et al. [21] have discussed the importance of noisy and incomplete server logs and have evaluated several heuristics for reconstructing user sessions. Their experiments demonstrate that there is no single best heuristic and that heuristic selection depends on the application at hand. Such a method, which is based on server logs, is termed as "reactive" by Dohare et al. [9].

The tool that is most closely related to ForenRIA is ClickMiner [18]. ClickMiner also reconstructs user sessions from traces recorded by a passive proxy. Although ClickMiner has some level of JavaScript support, it is unable to handle rich Internet applications.

5. Conclusions

The ForenRIA tool described in this chapter can automatically reconstruct user interactions with rich Internet applications solely from HTTP traces. In particular, ForenRIA recovers all the application states rendered by the browser, reconstructs screenshots of the states and lists every action taken by the user, including recovering user inputs. ForenRIA is the first forensic tool that specifically targets rich Internet applications. Experiments demonstrate that the tool can successfully handle relatively complex rich Internet applications.

From a digital forensic point of view, ForenRIA has several advantages. The reconstruction process is performed offline to minimize security concerns, it is fully automated and it is even able to render the pages seen by users. ForenRIA also saves precious time during an investigation. Nevertheless, it is intended to complement – not replace – other tools: ForenRIA focuses on the application user interface and works at the HTTP level whereas other tools operate at the lower network level. ForenRIA is, thus, unsuitable when an attack bypasses the user interface; however, in such a case, it is quite straightforward to understand the attack directly from the log, without having to use an advanced tool. It is important to note that ForenRIA is not limited to forensic analyses. It can be used in other situations, such as to automatically retrace the steps that led to any (non-necessarily security-related) incident and to automatically reproduce the steps involved in a bug report.

Future research will test ForenRIA on a larger set of rich Internet applications. Also, it will attempt to enhance the algorithm to discover implicit clues. Additionally, a more efficient technique will be developed to restore the previous state when the selected element is not the right one; this technique will leverage the approach described in [15].

The effective handling of browser caching is an open problem. In particular, SR-Browser has to automatically adapt to the caching strategies adopted by user browsers. The behavioral differences between browsers is a problem that requires more research. Finally, some user actions may not generate any traffic, but the actions are, nevertheless, necessary to continue the session. Dealing with this issue when reconstructing user sessions is also an open question.

Acknowledgement

This research was supported by the IBM Center for Advanced Studies, the IBM Ottawa Software Lab and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] S. Andrica and G. Candea, WaRR: A tool for high-fidelity web application record and replay, *Proceedings of the Forty-First IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 403–410, 2011.

[2] AppNeta, Tcpreplay, Boston, Massachusetts (`tcpreplay.appneta.com`), 2016

[3] R. Atterer and A. Schmidt, Tracking the interaction of users with AJAX applications for usability testing, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1347–1350, 2007.

[4] K. Chen, G. Gu, J. Zhuge, J. Nazario and X. Han, WebPatrol: Automated collection and replay of web-based malware scenarios, *Proceedings of the Sixth ACM Symposium on Information, Computer and Communications Security*, pp. 186–195, 2011.

[5] M. Cohen, PyFlag – An advanced network forensic framework, *Digital Investigation*, vol. 5(S), pp. S112–S120, 2008.

[6] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere and K. Bosschere, A taxonomy of execution replay systems, *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine and Mobile Technologies on the Internet*, 2003.

[7] R. Dell, P. Roman and J. Velasquez, Web user session reconstruction using integer programming, *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, pp. 385–388, 2008.

[8] R. Dell, P. Roman and J. Velasquez, Web user session reconstruction with back button browsing, in *Knowledge-Based and Intelligent Information and Engineering Systems*, J. Velasquez, S. Rios, R. Howlett and L. Jain (Eds.), Springer, Berlin Heidelberg, Germany, pp. 326–332, 2009.

[9] M. Dohare, P. Arya and A. Bajpai, Novel web usage mining for web mining techniques, *International Journal of Emerging Technology and Advanced Engineering*, vol. 2(1), pp. 253–262, 2012.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, Hypertext Transfer Protocol – HTTP/1.1, RFC 2616, 1999.

[11] D. Flanagan, *JavaScript: The Definitive Guide*, O'Reilly Media, Sebastopol, California, 2011.

[12] P. Fraternali, G. Rossi and F. Sanchez-Figueroa, Rich Internet applications, *IEEE Internet Computing*, vol. 14(3), pp. 9–12, 2010.

[13] J. Garrett, Ajax: A new approach to web applications, Adaptive Path, San Francisco, California (`www.adaptivepath.com/ideas/ajax-new-approach-web-applications`), February 18, 2005.

[14] S. Hong and S. Wu, On interactive Internet traffic replay, *Proceedings of the Eighth International Conference on Recent Advances in Intrusion Detection*, pp. 247–264, 2006.

[15] J. Lo, E. Wohlstadter and A. Mesbah, Imagen: Runtime migration of browser sessions for JavaScript web applications, *Proceedings of the Twenty-Second International Conference on World Wide Web*, pp. 815–826, 2013.

[16] J. Mickens, J. Elson and J. Howell, Mugshot: Deterministic capture and replay for JavaScript applications, *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, 2010.

[17] S. Narayanasamy, G. Pokam and B. Calder, BugNet: Recording application-level execution for deterministic replay debugging, *IEEE Micro*, vol. 26(1), pp. 100–109, 2006.

[18] C. Neasbitt, R. Perdisci, K. Li and T. Nelms, ClickMiner: Towards forensic reconstruction of user-browser interactions from network traces, *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 1244–1255, 2014.

[19] J. Oh, J. Kwon, H. Park, and S. Moon, Migration of web applications with seamless execution, *Proceedings of the Eleventh ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 173–185, 2015.

[20] Selenium, Selenium Web Application Testing System (`seleniumhq.org`), 2015.

[21] M. Spiliopoulou, B. Mobasher, B. Berendt and M. Nakagawa, A framework for the evaluation of session reconstruction heuristics in web-usage analysis, *INFORMS Journal on Computing*, vol. 15(2), pp. 171–190, 2003.

[22] J. Srivastava, R. Cooley, M. Deshpande and P. Tan, Web usage mining: Discovery and applications of usage patterns from web data, *ACM SIGKDD Explorations Newsletter*, vol. 1(2), pp. 12–23, 2000.

[23] World Wide Web Consortium, Document Object Model (DOM) Level 3 Core Specification, Version 1.0, W3C Recommendation, Cambridge, Massachusetts (`www.w3.org/TR/DOM-Level-3-Core`), 2004.

[24] G. Xie, M. Iliofotou, T. Karagiannis, M. Faloutsos, and Y. Jin, ReSurf: Reconstructing web-surfing activity from network traffic, *Proceedings of the IFIP Networking Conference*, 2013.

[25] M. Xu, R. Bodik and M. Hill, A “flight data recorder” for enabling full-system multiprocessor deterministic replay, *Proceedings of the Thirtieth Annual International Symposium on Computer Architecture*, pp. 122–135, 2003.