JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions

Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci
Department of Computer Science, University of Georgia
{bo,vadrevu,khlee,perdisci}@cs.uga.edu

Abstract—**In this paper, we propose JSgraph, a forensic engine that is able to efficiently record fine-grained details pertaining to the execution of JavaScript (JS) programs within the browser, with particular focus on JS-driven DOM modifications. JSgraph's main goal is to enable a detailed, post-mortem reconstruction of ephemeral JS-based web attacks experienced by real network users. In particular, we aim to enable the reconstruction of social engineering attacks that result in the download of malicious executable files or browser extensions, among other attacks.**

We implement JSgraph by instrumenting Chromium's code base at the interface between Blink and V8, the rendering and JavaScript engines. We design JSgraph to be lightweight, highly portable, and to require low storage capacity for its fine-grained audit logs. Using a variety of both in-the-wild and lab-reproduced web attacks, we demonstrate how JSgraph can aid the forensic investigation process. We then show that JSgraph introduces acceptable overhead, with a median overhead on popular website page loads between 3.2% and 3.9%.

I. Introduction

It is well known that JavaScript (JS, for short) is the main vehicle for web-based attacks, enabling the delivery of sophisticated social engineering, drive-by malware downloads, cross-site scripting, and other attacks [20], [26], [29], [8], [14]. It is therefore important to develop systems that allow us to analyze the inner workings of JS-based attacks, so to enable the development of more robust defenses. However, while extensive previous work exists on JS code inspection [9], [8], [42], [41] and web-based attack analysis [4], [37], [35], [44], [2], an important problem remains: to evade defense systems and security analysts, web-based attacks are often developed to be ephemeral and to deliver the actual attack code only if certain restrictive conditions are met by the potential victim environment [26], [20], [45]. Therefore, there is a need for JS-based attack analysis tools that can enable real-time *in-browser recording*, and subsequent detailed reconstruction, of *live* security incidents that affect real users while they simply browse the web.

In this paper, we aim to meet the above mentioned needs by proposing JSgraph, a forensic engine that is able to efficiently record fine-grained details pertaining to the execution of JavaScript programs within the browser, with particular focus on JS-driven DOM modifications. Ultimately, our goal is to enable a detailed, post-mortem reconstruction of ephemeral JS-based web attacks experienced by real network users. For instance, we aim to enable the reconstruction of social engineering attacks that result in the download of malicious executable files or browser extensions, among other attacks.

Our main target deployment environment is enterprise networks, including both mobile and non-mobile network-connected devices. In such networks, it is common practice to perform forensic investigations after a security incident is discovered, and our primary goal is to aid such forensic investigations by providing fine-grained details about web-born attacks to the network's devices.

To achieve our goal, we design JSgraph to satisfy the following main requirements:

- *Efficient Audit Log Recording*. Because we aim to record web attacks in real time, *as they affect real victims*, and in consideration of the fact that most web attacks are both difficult to anticipate and ephemeral, we need audit log recording to be *always on*. Consequently, the main challenge we face is whether it is feasible to record highly detailed information related to in-browser JS code execution without significantly impacting the browser's performance and usability.
- *No Functional Interference*. We aim to avoid any modification to the browser's code base that would alter its functionalities. For instance, some debugging tools that perform in-browser record and replay, such as Time-Lapse [4] and ReJS [44], alter the rendering engine to force it to effectively run in single-threaded mode. As this may have an impact on both rendering performance and behavior, we deliberately avoid making any such changes.
- *Portability*. To make it easily adoptable, we aim to implement a system that is highly portable. To this end, we build JSgraph by instrumenting Chromium's code base at the interface between its rendering engine (Blink) and the JavaScript engine (V8). By confining the core of JSgraph within Blink/V8 (more precisely, within Chromium's content module [6]), we are able to inherit Chromium's portability, thus making it easier to deploy JSgraph on multiple platforms (e.g., Linux, Android, Mac, Windows), and different Blink/V8-based browsers (e.g., Opera, Yandex, Silk, etc.) with little or no changes.
- *Limited Storage Requirements*. Because security incidents are often discovered weeks or even months after the fact, we aim to minimize the storage requirements for JSgraph's audit logs, making it feasible to retain the logs for extended periods of time (e.g., one year or longer).

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23319
www.ndss-symposium.org

In a nutshell, JSgraph works as follows (system details are provided in Section II). Given a browser tab, JSgraph monitors every navigation event, logs all changes to the DOM that occur for each page loaded within that tab, records how JS code is loaded (i.e., whether it is defined "inline" or loaded from an external URL), follows the execution of every compiled JS script, and logs every change that a script (or a callback) makes to the DOM. This enables the reconstruction of how a page's DOM evolved in time, and how changes to that DOM exactly came about. Ultimately, this enables a forensic analyst to trace back what JS script or function was responsible for making a given DOM change, including pinpointing what JS scripts were responsible for presenting a social engineering attack to the victim, and how the attack was actually constructed within the DOM.

To make JSgraph efficient, we implement its core logging functionalities by extending the DOM and JS code tracing functionalities offered by Chromium's DevTools. We then show that our system introduces acceptable performance overhead. For instance, we show that, on the top 1,000 websites according to Alexa, JSgraph running on Linux introduces a median website page load overhead of 3.2%, and a 95th-percentile overhead of 7.4%. Besides building an instrumented browser that can efficiently record fine-grained audit logs, JSgraph also implements a module for abstracting its fine-grained logs into more easily interpretable graphs. A motivating example that illustrates how this can help in analyzing in-the-wild web attacks is provided in the next Section I-B.

A. Threat Model

JSgraph aims to accurately record information that enables the reconstruction of web attacks, with an emphasis on social engineering malware attacks, but excluding attacks to the browser software itself. Namely, we assume the browser's code is part of our trusted computing base (TCB), along with the operating system's code. As JSgraph is implemented via lightweight instrumentation of the browser, we also assume that JSgraph's code is part of the TCB.

This entails that fully recording the behavior of drive-by exploit kits [16], for example, is outside the scope of this paper. Nonetheless, we should notice that JSgraph is capable of accurately recording the execution of malicious JS code delivered by exploit kits, up to the point in which the browser itself is compromised. If the exploit succeeds, we cannot guarantee that JSgraph will not be disabled, or that the logs produced afterwards will be accurate, because the exploit code could alter the logging process. At the same time, the logs recorded *before* a successful exploit could be securely stored outside the reach of possible tampering from the compromised browser, for example by using append only log files [31], [3], [34].

B. Motivating Example

In this section, we walk through a motivating example to show how JSgraph can aid the forensic investigation of web security incidents. Specifically, we analyze a real-world social engineering malware download attack promoted via malicious advertisement. The attack was observed on May 12, 2017.

Overview: The attack works as following (see Figure 1). (a) The user simply searches for "wolf of wall street full movie"; (b) After clicking on the first search result, the browser navigates to gomovies[.]to. (c) Clicking on the play button to start streaming the movie causes a new window to popup, under the pressupdateforsafesoft[.]download domain name. An alert dialog is displayed, with the message "Update the latest version of Flash Player. Your current Adobe Flash Player version is out of date." Notice also that the same page displays a "Latest version of Adobe Flash Player required [...]" message right under the URL bar. (d) Clicking the OK button causes a download dialog box to be shown. (e) Finally, clicking on the "Download Flash" (or "OK") button initiates a `.dmg` file download. Interestingly, after the download starts, the attack page also displays the instructions that the user needs to follow to install the downloaded software.

Attack Properties: Searching for the downloaded file's SHA1 hash[1] on VirusTotal produced no results. Upon submission, 10 out of 56 anti-viruses found the file to be malicious. At the time of writing, Symantec labels the file as `OSX.Trojan.Gen`.

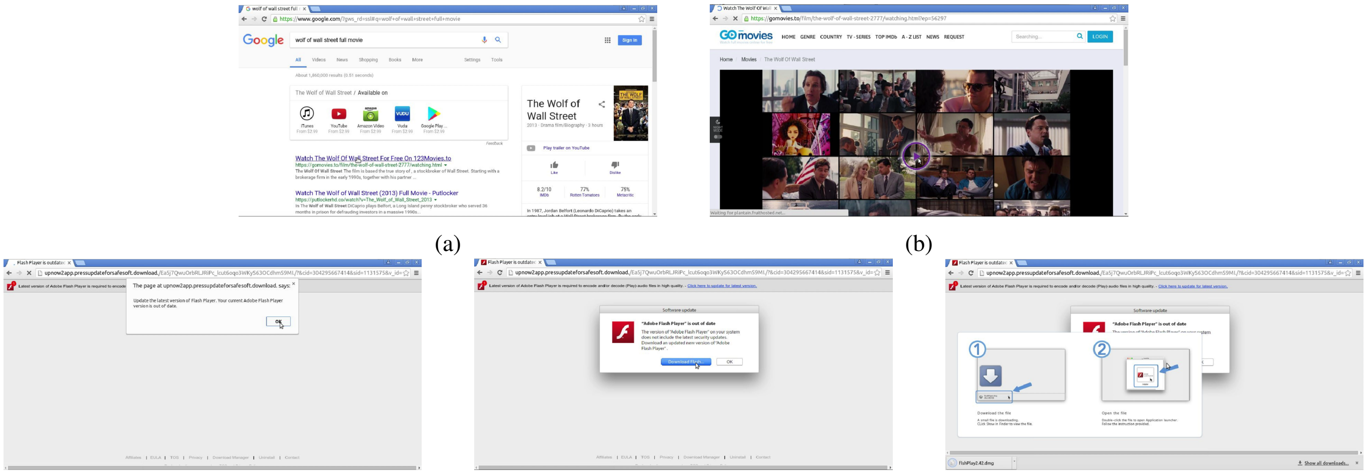
By leveraging a passive DNS database and domain registration information, we discovered that the two domain names that are used to deliver the malicious binary, namely pressupdateforsafesoft[.]download and pressbuttonforupdate[.]bid, are related to more than 300 domain name variations that are highly likely used for a large malware distribution campaign, because they shared close name similarity, date of registration, and resolved IP addresses (e.g., pressandclickforbestupdates[.]download, pressyoourbest-button2update [.]download, clickforfreeandbestupdate[.] download, click2freeupdatethebest[.]bid, etc.). In addition, we found that in a time window of about eight days, more than one thousand clients (roughly one third of which were located in the US) may have fallen victim to this malware campaign.

How JSgraph can Help: The question we would like to answer is: "how did this attack work under the hood?" Answering this question is important, because knowing how the attack is delivered can greatly help in developing effective countermeasures. Below, we discuss how JSgraph can help in answering this question.

Remember that JSgraph is an always-on in-browser record-only system, which aims to perform an efficient recording of any DOM change, with particular focus on DOM changes triggered by JS code execution. Our goal is to record highly detailed audit logs that can enable the reconstruction of complex JS-based attacks. At the same time, we aim to provide a tool that can present a forensic analyst with a high-level and thus more easily interpretable description of how the attack played out.

Our analysis of the attack starts with retrieving, from the JSgraph logs, the URL that served the executable file download. One may ask "how can the forensic analyst know where to look for potential malware downloads?" To help answering this question and aid the analysis process, JSgraph instruments the browser so that it can record if a file download (of any kind) is initiated, the URL from which the download occurs, and the hash and storage path where the file was saved (while not currently implemented, JSgraph can also easily store a copy of every downloaded file in the audit logs). Similarly, JSgraph also instruments the browser to record the download and installation of new browser extensions. It is therefore straightforward to explore JSgraph's logs to identify all file (or extension) download events. This allows a forensic

[1] `flshPlay2.42.dmg: 1b9368140220d1470d27f3d67737bb2c605979b4`



wolf of wall street full movie
Watch The Wolf Of Wall Street For Free On 123Movies.to
https://gomovies.to/film/the-wolf-of-wall-street-2777/watching.html
Watch The Wolf of Wall Street (2013) Full Movie - Putlocker
The Wolf of Wall Street
GO movies
Waiting for plantain.fruithosted.net...
Flash Player is outdated
upnow2app.pressupdateforsafesoft.download./Ea5j7QwuOrbRLJRiPc_lcut6oqo3WKyS63OCdhmS9MI./?&cid=304295667414&sid=1131575&v_id=
OK
Download Flash...
OK
FlashPlay2.42.dmg
Show all downloads...

The page at upnow2app.pressupdateforsafesoft.download. says:

Update the latest version of Flash Player. Your current Adobe Flash Player version is out of date.

Latest version of Adobe Flash Player is required to encode and/or decode (Play) audio files in high quality. - Click here to update for latest version.

Software update

"Adobe Flash Player" is out of date

The version of "Adobe Flash Player" on your system does not include the latest security updates. Download an updated new version of "Adobe Flash Player" .

(a) (b)

(c) (d) (e)

Fig. 1: Overview of in-the-wild social engineering malware download attack

analyst to spot potential malicious software installations. In the particular example we consider here, a forensic analyst may notice that an executable file named `flshPlay2.42.dmg` was downloaded from a suspicious .bid domain name (i.e., pressbuttonforupdate[.]bid). We assume this to be our starting point for attack analysis.

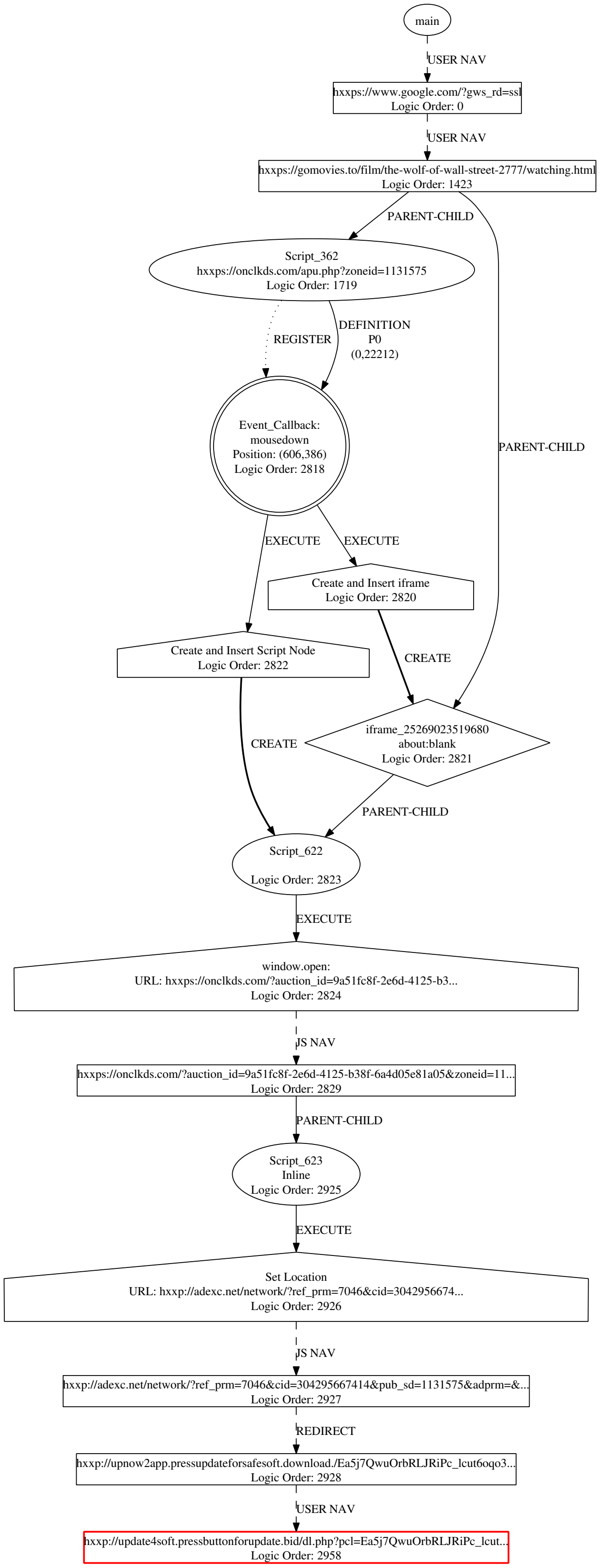
JSgraph's audit logs report fine-grained details about where a given piece of JS code originated from, what event listeners it registered (if any), exactly what DOM modifications it requested, and how those changes were made (e.g., via `document.write`, explicit DOM node creation and insertion, change of a DOM element's parameters, etc.). Now, let us refer to the graph in Figure 2, which we automatically derived by post-processing and abstracting JSgraph's audit logs (see also the legend in Figure 6 in Section III). The details on how this graph was generated are provided in Section III. In this section, we will leverage the graph simply as an example of how JSgraph can help in simplifying the analysis of web attacks.

The graph was computed by starting from the download URL (the node at the bottom highlighted in red) and backtracking along browsing events, until the beginning of the browsing session (e.g., until a parent tab first opened). What the graph shows is that the user first visited www.google.com. Notice that the search query string typed by the user is not shown in the first graph node. The reason is that Google uses `XMLHttpRequests` to send search keywords to the server and dynamically load the search results, and that the page's URL is changed by JS code by leveraging `history.pushState()` without triggering any navigation. This type of information is captured in detail in the JS audit logs, as shown in Figure 3; however, for the sake of simplicity our log visualization tool does not include them in the graph. Nonetheless, the forensic analyst could use the graph to identify nodes of interest, and then further explore the related detailed logs, whenever needed.

Figure 2 shows that the user then navigated to gomovies[.]to. There, the browser was instructed to load and execute a piece of JS code (Script_362) that registered an event listener for `mousedown` events on an element of the page. As the user clicked to watch the movie (see Figure 1b), the callback was activated, which first created a "no source" `iframe` element (the source is indicated as `about:blank`), dynamically generated some JS code, and injected the new `script` (Scrip_622) in the context of the newly created `iframe`, as also shown in Figure 4. As the new JS code is injected into the DOM, it is compiled and executed, triggering a `window.open` call. A new window is then opened, with content loaded from onclkds[.]com, including a JS `script` that redirects to adexc[.]net by resetting the page's location. Then, an HTTP-based redirection takes the browser to a page on pressupdateforsafesoft[.]download. As we will see later, this page renders as shown in the screenshots of Figures 1c-1e (notice that while JSgraph does not log visual screenshots, this functionality could be easily implemented very efficiently with the approach used by ChromePic [43]). As the user clicks on the download button (see Figure 1d), this corresponds to clicking on an HTML anchor that navigates the browser to the pressbuttonforupdate[.]bid, triggering the `.dmg` file download.

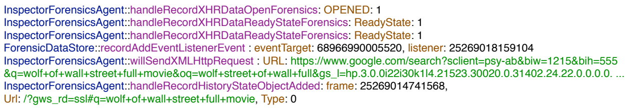
We would like to emphasize that this backtracking graph provides a high-level, and more easily interpretable abstraction of the highly complex web content loaded by the browser. In fact, the gomovies[.]to page alone contains 121 scripts, for a total of more than 6.2MB of (mostly obfuscated) JS code. Also, the pressupdateforsafesoft[.]download page contains a large amount of JS code, which is needed to create the social engineering portion of the attack. JSgraph condenses these to report only the content of interest that had a direct role in leading to the actual malware attack.

To further analyze the social engineering code delivered by the attack, and how the malware download is actually triggered in practice, the forensic analyst could then focus on the last step of the attack, namely the page under pressupdateforsafesoft[.]download, and ask JSgraph to perform forward tracking. The resulting graph is shown in Figure 8 in Section IV. While we defer a detailed explanation of the forward tracking graph to Section IV, from Figure 8 we can notice that the JS code shows an alert popup, listens to the user's clicks (which is needed to begin the file download), and schedules callbacks, which we found are used to display the installation instructions shown in Figure 1e.



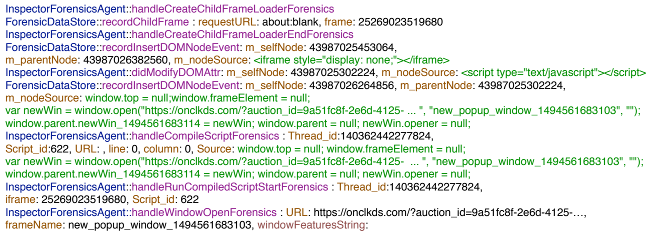
main
USER NAV
hxxps://www.google.com/?gws_rd=ssl
Logic Order: 0
USER NAV
hxxps://gomovies.to/film/the-wolf-of-wall-street-2777/watching.html
Logic Order: 1423
PARENT-CHILD
Script_362
hxxps://onclkds.com/apu.php?zoneid=1131575
Logic Order: 1719
REGISTER
DEFINITION
P0
(0,22212)
PARENT-CHILD
Event_Callback:
mousedown
Position: (606,386)
Logic Order: 2818
EXECUTE
EXECUTE
Create and Insert iframe
Logic Order: 2820
CREATE
Create and Insert Script Node
Logic Order: 2822
CREATE
iframe_25269023519680
about:blank
Logic Order: 2821
PARENT-CHILD
Script_622
Logic Order: 2823
EXECUTE
window.open:
URL: hxxps://onclkds.com/?auction_id=9a51fc8f-2e6d-4125-b3...
Logic Order: 2824
JS NAV
hxxps://onclkds.com/?auction_id=9a51fc8f-2e6d-4125-b38f-6a4d05e81a05&zoneid=11...
Logic Order: 2829
PARENT-CHILD
Script_623
Inline
Logic Order: 2925
EXECUTE
Set Location
URL: hxxp://adexc.net/network/?ref_prm=7046&cid=3042956674...
Logic Order: 2926
JS NAV
hxxp://adexc.net/network/?ref_prm=7046&cid=304295667414&pub_sd=1131575&adprm=&...
Logic Order: 2927
REDIRECT
hxxp://upnow2app.pressupdateforsafesoft.download./Ea5j7QwuOrbRLJRiPc_lcut6oqo3...
Logic Order: 2928
USER NAV
hxxp://update4soft.pressbuttonforupdate.bid/dl.php?pcl=Ea5j7QwuOrbRLJRiPc_lcut...
Logic Order: 2958

Fig. 2: Malware attack analysis using JSgraph: backtracking graph.



```
InspectorForensicsAgent::handleRecordXHRDataOpenForensics: OPENED: 1
InspectorForensicsAgent::handleRecordXHRDataReadyStateForensics: ReadyState: 1
InspectorForensicsAgent::handleRecordXHRDataReadyStateForensics: ReadyState: 1
ForensicDataStore::recordAddEventListenerEvent : eventTarget: 68966990005520, listener: 25269018159104
InspectorForensicsAgent::willSendXMLHttpRequest : URL: https://www.google.com/search?sclient=psy-ab&biw=1215&bih=555
&q=wolf+of+wall+street+full+movie&oq=wolf+street+of+wall+full&gs_l=hp.3.0.0i22i30k1l4.21523.30020.0.31402.24.22.0.0.0.0. ...
InspectorForensicsAgent::handleRecordHistoryStateObjectAdded: frame: 25269014741568,
Url: /?gws_rd=ssl#q=wolf+of+wall+street+full+movie, Type: 0
```

Fig. 3: JSgraph audit logs – Excerpt 1 (simplified)



```
InspectorForensicsAgent::handleCreateChildFrameLoaderForensics
ForensicDataStore::recordChildFrame : requestURL: about:blank, frame: 25269023519680
InspectorForensicsAgent::handleCreateChildFrameLoaderEndForensics
ForensicDataStore::recordInsertDOMNodeEvent: m_selfNode: 43987025453064,
m_parentNode: 43987026382560, m_nodeSource: <iframe style="display: none;"></iframe>
InspectorForensicsAgent::didModifyDOMAttr: m_selfNode: 43987025302224, m_nodeSource: <script type="text/javascript"></script>
ForensicDataStore::recordInsertDOMNodeEvent: m_selfNode: 43987026264856, m_parentNode: 43987025302224,
m_nodeSource: window.top = null;window.frameElement = null;
var newWin = window.open("https://onclkds.com/?auction_id=9a51fc8f-2e6d-4125- ... ", "new_popup_window_1494561683103", "");
window.parent.newWin_1494561683114 = newWin; window.parent = null; newWin.opener = null;
InspectorForensicsAgent::handleCompileScriptForensics : Thread_id:140362442277824,
Script_id:622, URL: , line: 0, column: 0, Source: window.top = null; window.frameElement = null;
var newWin = window.open("https://onclkds.com/?auction_id=9a51fc8f-2e6d-4125-  ... ", "new_popup_window_1494561683103", "");
window.parent.newWin_1494561683114 = newWin; window.parent = null; newWin.opener = null;
InspectorForensicsAgent::handleRunCompiledScriptStartForensics : Thread_id:140362442277824,
iframe: 25269023519680, Script_id: 622
InspectorForensicsAgent::handleWindowOpenForensics : URL: https://onclkds.com/?auction_id=9a51fc8f-2e6d-4125-…,
frameName: new_popup_window_1494561683103, windowFeaturesString:
```

Fig. 4: JSgraph audit logs – Excerpt 2 (simplified)

C. Differences w.r.t. Previous Work

We now discuss how the same attack described in Section I-B could be analyzed using previous work, and compare these alternative approaches to JSgraph. We should first remember that one of our main requirements is that we need to be able to record the "real" attack, as it happens on the user's system. The reasons for this requirement are multiple: (i) Web attacks are often ephemeral, and visiting the attack URLs at a later time (e.g., using high-interaction honeypots) would likely produce different or no results [19]. (ii) The attack code is often environment-sensitive, and may behave differently on other machines, compared to what the victim actually experienced. (iii) As we are interested in social engineering attacks, user actions are critical to "activate" the attack [39]; however, user actions are often difficult to reproduce exactly, unless a highly detailed recording of user-browser interactions is performed at the time of the attack. (iv) Some social engineering attacks (e.g., malware attacks) are delivered via malicious advertisement; because ad-serving networks may introduce a high level of non-determinism (e.g., due to the ad bidding process typical of online ad networks), it may be difficult to reproduce the exact same attack multiple times.

Keeping the real-time recording requirement in mind, there exist a few alternative approaches that may enable the analysis of in-the-wild web attacks that affect real users. One possible way would be to record, and later statically analyze, all the HTML and JavaScript content loaded by the browser during a time window that includes the attack. This could be done by recording all network traffic traces, or by using a lightweight system such as ChromePic [43]. However, understanding how the browser loaded, parsed, interpreted, and rendered the web content from network traces is notoriously hard [38]. Also, while ChromePic can efficiently record screenshots and DOM snapshots from inside the browser, it does so only at significant user interactions (e.g., at every mouse click, key press, etc.). This limits the visibility on DOM changes and JavaScript behavior that occurred in between such interactions. In addition, in these scenarios code analysis presents several challenges, since the code may need to be re-executed at a later time on a separate system, to try to fill the gaps, thus suffering from limitations similar to the ones faced by honey-clients.

Concretely, referring to the example in Figure 2, ChromePic would not be able to track and reconstruct fine-grained details about the JS code that enables the social engineering attack. For instance, ChromePic would not be able to log any detailed information about how Script_362 injects an `iframe` into the page, about the existence of Script_622 (which is dynamically generated) and how it opens a new window, and how Script_623 redirects the browser towards the malware download URL.

Another possible approach is to use record and replay (R&R) systems. However, VM-level R&R systems [12], [11] tend to be very inefficient, preventing them from being deployed on mobile devices, for example. On the other hand, OS-level R&R systems [36], [10] are more efficient, though they are not easily portable to different devices. Unfortunately, both these types of systems leave a large semantic gap that makes analyzing web attacks difficult. In fact, while they can re-run browsing sessions, they cannot interpret what is happening inside the browser, such as interpreting the interactions between the JS engine (e.g., V8) and the rendering engine (e.g., Blink) that carried out the attack. Attaching a JS debugger inside the browser (e.g., via DevTools) at replay time would alter the browser execution, compared to the recorded traces, and thus prevent a correct system-level replay to move forward.

Browser R&R systems such as TimeLapse [4] and WebCapsule [37] may come to help, in that they are able to record fine-grained details internal to the browser (rather then "external", as in system-level R&R systems), and thus fill the semantic gap that characterizes VM- and OS-level R&R systems. Unfortunately, because they attempt to record and replay all events at the rendering engine level (e.g., inside WebKit or Blink), both these systems tend to have high time and storage overhead and may fail to deterministically replay the recorded browsing traces. For instance, in an attempt to achieve deterministic replay, TimeLapse changes the rendering engine to effectively prevent multi-threading, thus violating the *no functional interference* requirement. On the other hand, WebCapsule does not explicitly record JS-level events such as scheduled actions, and is therefore incapable of performing deterministic replay [37].

JavaScript-level R&R debugging tools, such as Mugshot [35] and ReJS [44], offer direct visibility into JS execution and JS-driven DOM changes, and could therefore be used to perform a replay and step-by-step analysis of JS attack code. However, these systems were not intended for always-on recording, and are not suitable for analyzing adversarial JS code. For instance, Mugshot is not transparent, in that it modifies the JS environment, and could be detected (and potentially also disabled) by the JS attack code being recorded. On the other hand, ReJS forces the rendering engine to run in single-threaded mode, thus impacting the browser's functionality and performance in a way similar to TimeLapse.

Unlike the works mentioned above, JSgraph aims to be an efficient, always-on, record-only system that is capable of producing highly detailed audit logs related to browsing sessions, and that can assist in the investigation of in-the-wild web attacks.

II. JSGRAPH SYSTEM

In this section, we explain how JSgraph works internally.

A. Overview

JSgraph consists of two components: (i) an efficient, fine-grained *audit logging engine*, and (ii) a *visualization module* (detailed in Section III) that can post-process the audit logs to produce a higher-level description of navigation events, JS code inclusion and execution, DOM modifications, etc.

To efficiently record internal browsing events, we leverage and extend Chrome's DevTools. Specifically, we implement a new `InspectorAgent`, extending the `InspectorInstrumentation` APIs to collect fine-grained information that is not otherwise gathered by existing DevTools agents. This makes JSgraph highly portable. In fact, because the vast majority of JSgraph's code resides within Chromium's content module [6], it could be easily adapted and integrated in other browsers that make use of Blink/V8 for rendering and JS execution, such as Opera, Yandex, Amazon Silk, etc.

B. Efficiently Recording Page Navigations

Reconstructing the sequence of pages visited by a user is essential to understanding how modern web attacks work. For instance, the social engineering attack we described in Section I-B is delivered through multiple pages/URLs. To efficiently record fine-grained details about how the browser navigates from one page to another, we extend Chromium's DevTools instrumentation hook `didStartProvisionalLoad`, and register our JSgraph inspector agent to listen to the related callbacks. Furthermore, we instrument `receivedMainResourceRedirect` to efficiently record HTTP-based page redirections.

C. Logging `iframe` Loading Events

Unlike page navigations, to record the loading of an `iframe` whose content loads from a URL expressed in the `src` parameter, we create a new instrumentation hook into `WebLocalFrameImpl:: createChildFrame`. This allows us to record a pointer to the `iframe` to be loaded and the URL from which the content will be retrieved. As the `iframe`'s web content is loaded asynchronously by the browser, this information allows us to correctly track all DOM changes related to the `iframe`'s DOM, including the compilation and execution of JS code and callbacks within the `iframe`'s context.

D. Tracking DOM Changes

Our main goal in recording DOM changes is to be able to reconstruct the state of the DOM right before each JS code execution, thus allowing us to understand how potentially malicious code modifies the DOM to launch an attack. To improve efficiency, instead of creating a full DOM snapshot every time a JS script or callback function is executed, we incrementally record all DOM changes applied by Blink, including all changes requested by the HTML parser and the JS engine via the Blink/V8 bindings. To achieve this, we leverage six different DevTools instrumentations: `didInsertDOMNode`, `characterDataModified`, `willRemoveDOMNode`, `didModifyDOMAttr`, `didRemoveDOMAttr`, and `didInvalidateStyleAttr`. Moreover, to efficiently store information about the node that was added/removed or modified, we take advantage of Blink's DOM serialization functionalities[2].

We now provide more details about how we leverage the `InspectorInstrumentation` APIs listed above.

[2] see /src/third_party/WebKit/Source/core/editing/serializers/Serialization.h

- `didInsertDOMNode` monitors the insertion of DOM nodes. To allow us to later reconstruct the exact position of the inserted node in the page DOM, its parent node pointer, its next sibling and the HTML markup of the node (using `createMarkup`). This will also record all node attributes, including the `src` parameter, if content needs to be loaded from an external source. Because the DOM tree can be built by assembling document fragments (e.g., by inserting an entire DOM subtree via JS code), the inserted node could actually represent the root of a subtree with many children nodes. Therefore, we log the markup representation for the entire subtree. Notice that knowing the subtree root's parent and next sibling is still sufficient to correctly reconstruct the state of the DOM tree during analysis.
- `characterDataModified` logs any modifications to text nodes. For instance, during DOM construction, if a text node is too large to load at once, the parser will create a node with partial data and perform a character data modification once the content of the node finishes loading. JSgraph simply records the node pointer and the final state of the node content. Because text nodes do not have attributes, and for efficiency reasons, we record the value of the text node without having to store the full node markup.
- `willRemoveDOMNode` monitors the deletion of a DOM node. We record the pointer of the node that is going to be removed, so that the event can be reconstructed by parsing the audit logs and matching the deleted node pointer to the related entry in the reconstructed DOM tree.
- `didModifyDOMAttr` and `didRemoveDOMAttr`, record all changes to a DOM node's attributes, whereas `didInvalidateStyleAttr` is called when a node's style change is requested.

E. Logging Script Executions and Callbacks

Before explaining how we record scripts and callbacks execution, we first need to provide some high-level background on how JS *scripts* and *callbacks* are executed in Blink/V8. Let us first consider *scripts*. Essentially, a scripts can be defined "inline," as part of the page's HTML, or can be loaded from an external source, e.g., by expressing a URL within the `src` parameter of a `script` HTML tag. When a `script` node is inserted into the DOM, Blink will retrieve the related source code and pass it to V8 to be compiled. The JS compiler will give the script's code a unique script identifier within that V8 instance, and will then execute the script right after compilation. On the other hand, *callbacks* are JS functions that are defined either within a JS script or as a *DOM level 0* event handler, and will be executed when a certain circumstance to which they "listen" arises (e.g., an event such as mousedown, keypress, etc.). There exist multiple types of callbacks, including event callbacks, scheduled callbacks, animation callbacks, mutation observers, errors, and idle task callbacks. It is also worth noting that a callback function could be defined in a JS script *script_A*, but registered as a callback for an event (e.g., using `addEventListener`) by a separate script *script_B*.

To record complex relationships between DOM elements, scripts, and callback functions, which can greatly help in understanding the inner-workings of JS-driven web attacks, we extend Chromium's DevTools by adding a number of instrumentation hooks within the code bindings that link Blink to V8 and allow JS code to access and modify the DOM.

Specifically, we instrument Chromium's `V8ScriptRunner` and `ScriptController`, adding five instrumentation hooks: to handle events such as *CompileScript*, *RunCompiledScriptStart*, *RunCompiledScriptEnd*, *CallFunctionStart*, and *CallFunctionEnd*.

At the moment in which V8 is called to compile a script, we record detailed information that will be difficult to retrieve once the code is compiled, such as the source code, the source URL from which the code was retrieved, and the start position of the code in the HTML document (in terms of text coordinates) for "inline" scripts. We also record the script ID assigned by V8 to the compiled code, to link future executions of the script to its source code. When *RunCompiledScriptStart* is called, we also log the script ID and its execution context, by recording the address of the frame (or page) within which the script was loaded.

Because JavaScript execution within a tab can be seen as single-threaded (notice that WebWorkers do not have direct access to the DOM), all the DOM changes that are made by JS code in between the start and end of a *RunCompiledScript* can be uniquely attributed to a specific script ID recorded in the audit logs. Similarly, observing when a *CallFunction* starts and ends allows us to record the name of the callback function, the script ID related to the source code where it was defined, and the line and column number where the function is located in the source code. However, these instrumentation hooks do not allow us to determine how the callback functions were registered and triggered. To this end, we additionally instrument calls to `addEventListener` and `willHandleEvent`, to log the execution of the callbacks. This allows us to determine what JS script registered a certain callback function, and for what particular event. In addition, when a callback is triggered, we can record the details of the event that triggered it. For instance, if the event is a *mousedown*, we can record the event type and mouse coordinates; if the event is a *keypress*, we record the key code; etc. (our instrumentation also takes event bubbling into account, to record the correct target DOM element). In a similar way, we also record callbacks associated to `XMLHTTPRequests`, for which we record the request URL, request header, ready state, response content, etc. We follow a similar logging process to record details related to scheduled callbacks, animation callbacks, idle task callbacks, etc. JSgraph also records messages passed between frames, thus enabling the reconstruction of possible multi-frame attacks. In addition, JSgraph can naturally handles asynchronous scripts. From JSgraph's point of view, `script` tags with an ”async” attribute do not differ from synchronous scripts. The reason is that for all scripts, whether they run asynchronously or not, JSgraph will record the exact time when a script is parsed and compiled by the browser, as well as whenever a script performs an action on the page.

Notice that, because we automatically log DOM and JS events belonging to different tabs into different log files, the recorded events described above can be correctly attributed to a specific web page and related frames. This per-tab logging approach also serves the purpose of enabling opportunistic offloading and improving log security and privacy, because each tab can be independently encrypted (with different keys from a key escrow) and archived.

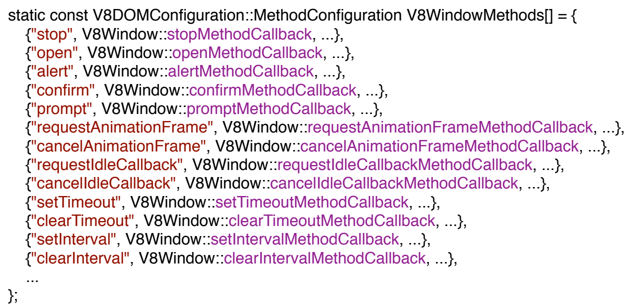
Nested Scripts and Callbacks – One factor that complicates the logging and reconstruction of the relationship between scripts and callbacks, is the possibility of *nested* execution. The nested execution of JS code may occur due to dynamic JS code generation, such as when a JS script, *script_A*, adds an additional `script` tag into the DOM (e.g., via `document.write()`), thus triggering the execution of a new script, *script_B*. In this case, the execution of *script_A* will pause until *script_B* is compiled and executed, after which the execution of *script_A* will resume (a similar scenario may occur in other corner cases; for instance, if an `iframe` with no source and a DOM level 0 `onload` event callback is dynamically added to the DOM via JS code). JSgraph is able to correctly reconstruct such nested executions as well.

F. Logging Critical Events

Of course, logging only DOM changes does not allow us to have a complete picture of how JS code may impact the user's browsing experience. To this end, we instrument a number of critical JS methods and attributes related to changing the page's location (e.g., with `location.replace() or location.href`, opening a new tab or window (e.g., with `window.open()`), making asynchronous network requests (e.g., sending an XMLHttpRequest), etc.

Identifying what JS methods and attributes to instrument is challenging, because there exist literally thousand of APIs available to JS code. Fortunately, we are only interested in JS APIs that have an effect on the page, by either modifying the current DOM tree, changing the page URL, opening new pages, loading new web content, passing messages between page components, etc. Conversely, we do not need to log calls to APIs that allow for reading the value a variable (e.g., `Node.nodeType()`, `location.toString()`, etc.), as they have no effect on the page/DOM, and are therefore less important to understand how a piece of malicious JS constructed page elements to launch an attack (e.g., a social engineering attack). To identify what APIs are of interest, we proceed as explained below.

In practice, Blink and V8 communicate via an interface referred to as "bindings." Essentially, all calls to JS methods or attributes that request or pass data to the rendering engine (e.g., to insert or remove a DOM node or change its attributes, read/change the URL, open a new window, etc.) must pass through these bindings. The bindings are dynamically generated when Chromium is compiled, via a fairly complex process (explaining this process is out of the scope of this paper; we refer the reader to [7] for details). However, once the bindings are compiled, they can be accessed at a specific disk location[3], which for brevity we refer to as `blink/bindings`. Under `blink/bindings`, a large number of C++ classes are created, within multiple subdirectories and .cpp files, that enable access to Blink from JS code. Especially, `V8DOMConfiguration::MethodConfiguration` mappings are of particular interest. For instance, these include methods such as `Document::write`, `Window:: setTimeout`, `XMLHttpRequest::send`, and so on, just to name a few. A small excerpt from the bindings code for the `Window`'s `MethodCallbacks` is shown in Figure 5.



```
static const V8DOMConfiguration::MethodConfiguration V8WindowMethods[] = {
    {"stop", V8Window::stopMethodCallback, ...},
    {"open", V8Window::openMethodCallback, ...},
    {"alert", V8Window::alertMethodCallback, ...},
    {"confirm", V8Window::confirmMethodCallback, ...},
    {"prompt", V8Window::promptMethodCallback, ...},
    {"requestAnimationFrame", V8Window::requestAnimationFrameMethodCallback, ...},
    {"cancelAnimationFrame", V8Window::cancelAnimationFrameMethodCallback, ...},
    {"requestIdleCallback", V8Window::requestIdleCallbackMethodCallback, ...},
    {"cancelIdleCallback", V8Window::cancelIdleCallbackMethodCallback, ...},
    {"setTimeout", V8Window::setTimeoutMethodCallback, ...},
    {"clearTimeout", V8Window::clearTimeoutMethodCallback, ...},
    {"setInterval", V8Window::setIntervalMethodCallback, ...},
    {"clearInterval", V8Window::clearIntervalMethodCallback, ...},
    ...
};
```

Fig. 5: Excerpt from Blink/V8 bindings code we instrumented.

To select what methods should be instrumented, we proceeded as follows. First, we automatically instrumented the bindings of an unmodified version of Chromium, so to output a log message every time a Blink/V8 `MethodConfiguration` callback is called. Then, we used this instrumented version of Chromium to browse highly-dynamic websites, using the top ten global sites list from Alexa.com. Finally, we compiled a list of all Blink/V8 binding callbacks that were activated during these browsing sessions. This gave us a little less than one hundred APIs that we had to manually inspect. As the vast majority of API names clearly communicate the API's functionality, it was quite straightforward to select the API calls to be included in the audit logs, because they either directly impacted the page's content (e.g., changing page location, passing messages between page components, etc.) or represented critical events (e.g., opening a new window, showing an alert popup, etc.), and the ones that should be excluded. For a few APIs, we had to refer to the related documentation (i.e., JavaScript documentation or HTML standard) to understand their effect on the page. However this process was also straightforward. Once we identified the APIs to be logged, the more time consuming part of this process was to actually instrument the APIs at Blink's side, which required us to interpret and serialize all objects passed as arguments to each API of interest.

Notice that the API selection process discussed above is simply meant to reduce engineering effort. With more engineering time, our instrumentations could be extended to all APIs, and could potentially also be automated using Chromium's own dynamic code generation process for the bindings [7]. At the same time, the APIs currently instrumented by JSgraph are the most commonly used, and are therefore suitable for demonstrating JSgraph's capabilities and estimating performance overhead. Finally, as we will show in Section IV, the current instrumentation is sufficient to capture complex malicious code behavior.

G. Some Optimizations

When `didModifyDOMAttr`, `didRemoveDOMAttr`, or `didInvalidateStyleAttr` hooks are called, we need to be careful about what we log. As mentioned earlier, we use Blink's `createMarkup` function to log the HTML markup related to DOM nodes. However, `createMarkup` logs both the DOM node that is being modified as well as all its children, thus potentially generating a large (and costly) log at every node attribute modification. To avoid logging the entire subtree under a node, we therefore implemented a customized version of `createMarkup` to log only the actual node markup (along with the node pointer, parent, and next sibling pointer), without

[3] /src/out/Debug/gen/blink/bindings/

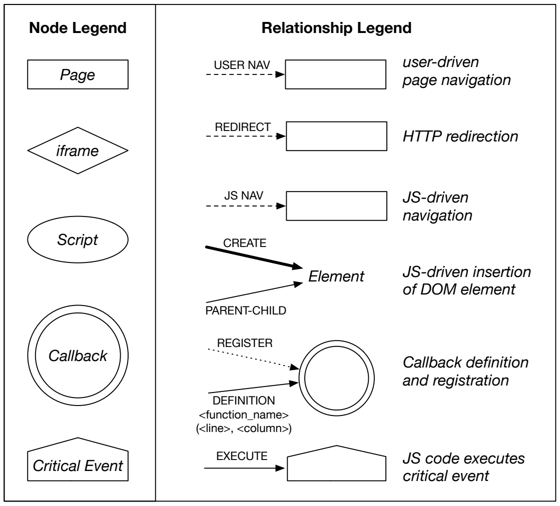

Node Legend
Page
iframe
Script
Callback
Critical Event
Relationship Legend
USER NAV
user-driven page navigation
REDIRECT
HTTP redirection
JS NAV
JS-driven navigation
CREATE
Element
PARENT-CHILD
JS-driven insertion of DOM element
REGISTER
DEFINITION <function_name> (<line>, <column>)
Callback definition and registration
EXECUTE
JS code executes critical event

Fig. 6: Audit Logs Visualization – Graph Legend

logging its children. In addition, we should notice that some HTML elements may contain attributes with large amounts of data. For instance, the `img` tag may have a `src` that embeds an entire (e.g., base64 encoded) image into a `data:` URL[4]. Similarly, CSS styles could also include `data:` URLs (e.g., to include a background image)[5]. To avoid storing the same large markup every time a DOM attribute or style is changed, therefore improving performance and storage overhead, we proceed as follows. The first time a node containing a `data:` URL is observed by our instrumentation hooks, we cache a hash of the `data:` URL. Next time an attribute or style is modified and we log the event, if the `data:` URL has not changed we only log a placeholder that indicates that the `data:` URL has not changed since we have last seen that node. This will be reflected in the logs, from which it is then easy to reconstruct the complete representation of the node by retrieving the full `data:` URL from the earlier logs related to the same node.

In large part, the overhead imposed by JSgraph comes from the log I/O overhead (i.e., writing the logs to disk). To reduce this overhead, we offload the job of storing the audit logs to disk to a separate Blink thread. To this end, we leverage `base::SingleThreadTaskRunner`[6], which allows us to create log writing tasks that are responsible for periodically storing batches of recorded events and can be executed in a separate thread (via `PostTask`).

III. Visualizing JSgraph's Audit Logs

As discussed in Section II, JSgraph's audit logs are very detailed, as they contain fine-grained information about all DOM modifications, the source code of JS scripts, critical JS API calls and parameters, file download events, etc. Finding interesting information among these detailed logs can be time consuming.

To aid the investigation process, JSgraph allows for visualizing important events captured in the audit logs in the form of a graph. A complete legend showing the meaning of the node shapes and what relationships are tracked by JSgraph is shown in Figure 6. The visualization process works in two steps. First, the analyst selects an event or object of interest. For instance, in the malware download attack we analyzed in Section I-B, the forensic analyst selects the suspected malware-serving URL as starting point. Then, given the starting point, JSgraph can produce two different graphs: a backward tracking graph and a forward tracking graph.

The backward tracking graph follows "causal" relationships, and visualizes the chain of events that directly affected the node of interest. As an example, let us refer again to the example in Section I-B, and consider the `window.open` event in Figure 2. From that event, the next iteration of the backward tracking process flags *Script_622* as having caused the `window.open` event. Notice that other JS scripts that may be present on the same page are deliberately not shown (unless they directly affected the currently considered node). Going one step further (or one causal relationship "up"), *Script_622* was directly affected (created and inserted into an `iframe`) by an event callback triggered by a *mousedown* event; and so on. The backward tracking ends when no new causal relationships can be found.

Referring again to the legend in Figure 6 and the example backward tracking graph in Figure 2, we should notice that the *critical events* essentially represent calls to the JS APIs we discussed in Section II-F. Also, notice that a script can *create* a node and insert it into the DOM as *child* of another *parent* node, thus producing a *parent-child* relationship. Similarly, a JS script can *define* a JS function, and then *register* that function as a callback.

The forward tracking graph aims to visualize different type of information. Specifically, given a starting node, we visualize significant events that have been "caused" by the starting node. We then recursively proceed by considering all nodes affected by the starting node, and performing forward tracking from each of them. An example of forward tracking graph related to the example in Section I-B is shown in Figure 8 (in Section IV). This graph was obtained by selecting the second-to-last URL from the backward tracking graph in Figure 2 (i.e., the URL of the page immediately preceding the malware download event), and walking forward through the logs.

To better explain what type of relationships are captured by JSgraph's visualization module, we now provide another example, for which we can analyze both the HTML content and the related graph. Figure 7 shows the forward tracking graph related to the HTML content in the top left quadrant. The logs were produced using our instrumented browser to load the HTML page, and then click on the "Click me" button.

Notice that the `showHello` function is defined as part of a `script`, but registered as an event listener via a *DOM level 0* `onclick` attribute. Also, notice that the definition of the anonymous function that is set as a callback for `setTimeout`, is also represented in the graph, with an edge from *Script_52* to the *Scheduled Callback* node (notice that the function name is missing from the graph, since this is an anonymous function). Also, the graph shows that *Script_51* is loaded from an external URL, and that it performs critical operations on the `window` object (an attempt to create a *popunder* window).

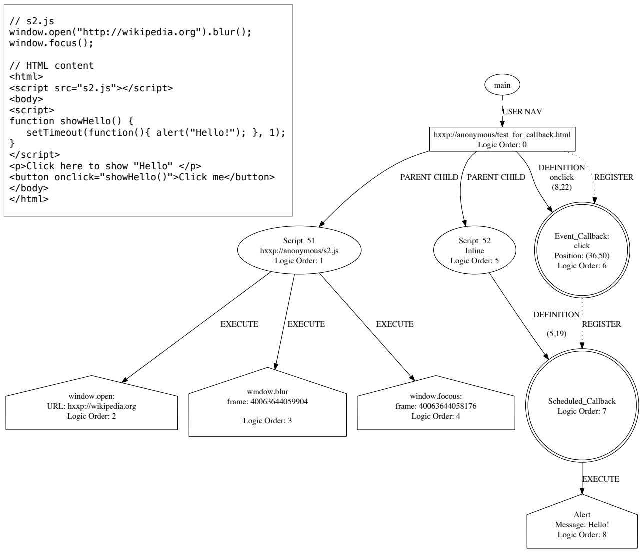
IV. Analysis of Web Attacks

In this section, we report details on three experiments aimed at demonstrating how JSgraph can record fine-grained

[4] https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs

[5] https://css-tricks.com/data-uris/

[6] see /src/base/single_thread_task_runner.h



```
// s2.js
window.open("http://wikipedia.org").blur();
window.focus();

// HTML content
<html>
<script src="s2.js"></script>
<body>
<script>
function showHello() {
   setTimeout(function(){ alert("Hello!"); }, 1);
}
</script>
<p>Click here to show "Hello" </p>
<button onclick="showHello()">Click me</button>
</body>
</html>
```

Fig. 7: HTML+JS content and related forward tracking graph

details about web-based attacks and make their post-mortem analysis easier. We will first provide details on the forward tracking graph for the malware download attack discussed in Section I-B. Then, we will analyze an in-the-wild social engineering attack that tricks users into installing a malicious extension, and a phishing attack based on a cross-site scripting (XSS) vulnerability in real web software [18].

A. Forward Tracking for Malware Download Attack

In Section I-B, we presented the backward tracking graph in Figure 1, which reconstructs the navigation steps and events that took the user from the starting page (the Google search) to the malware download event. On the other hand, Figure 8 reconstructs the JS scripts, callbacks, critical events, and navigations that occurred starting from the URL the user visited right before the malware download event (i.e., starting from the second-to-last node in Figure 1).

Figure 8 shows that an "inline" (i.e., not externally loaded) script (Script_624) first defines an anonymous function (at source line 13, column 19) to be registered as a scheduled callback. The scheduled callback registration is actually executed later, after a user's click, which activates the event callback at logic order 2956. This behavior corresponds to the excerpt from the attack code shown below. By analyzing the audit logs related to these graph nodes, we found that the `onclick` callback will be used later to display the installation instructions (hence the function name "showStep") for the downloaded software (see Figure 1e).

```
//DOM level 0 event
<a href="hxxp://update4soft.pressbuttonforupdate.bid/..."
    onclick="showStep();" class="download_link"></a>
//Script_624 (simplified)
<script>
function showStep() {
 window.onbeforeunload=null;
 var nAgt=navigator.userAgent;
 ...
 setTimeout(function(){
   window.location=
   "hxxp://update4soft.pressbuttonforupdate.bid/..."; },1000);}
</script>
```

Script_625 and Script_627 define and register an event listener for the `load` and `DOMContentLoaded` events, whereas Script_628 defines the `showPopup` function that will display the "fake" download dialog box in Figure 1d, and registers it as a scheduled callback. As it executes, Script_629 will raise a system alert with the message "Update the latest version of Flash Player. Your current Adobe Flash Player version is out of date," as shown in Figure 1c. This has the effect of "freezing" the tab, including the execution of all scheduled callbacks and the parsing of the rest of the page, until the user clicks "OK". As the user clicks on "OK" to close the alert window, the browser finishes loading the page, and fires the `DOMContentLoaded` and `load` event listeners, at logic order 2936 and 2937, respectively. Then, the scheduled callback at logic order 2938 is activated to show the "fake" download dialog box (Figure 1d), using JS-driven animations activated at logic order 2939-2955. When the user clicks on the download button, the static HTML anchor shown in the previous attack code excerpt is activated, to navigate to the malware download URL. At the same time, the DOM level 0 `onclick` callback will execute the registration of the scheduled callback, which will be triggered one second later (at logic order 2957) to make sure the malware download is indeed initiated.

B. Social Engineering Extension Download Attack

We also found that visiting the gomovies[.]to site from a Linux machine would lead to the installation of a malicious browser extension, rather than a `.dmg` software package[7].

As in the malware download case, clicking on the play button on gomovies[.]to causes a new window to popup, under the getsportscore[.]com domain name. As shown in Figure 9, a popup dialog box lures the user to add an extension called *Sport Score* to Chrome, which has been found to be responsible for delivering unwanted ads and PUP software[8] and is detected by the ESET anti-virus as *JS/Adware.StreamItOnline*[9]. Then, clicking the "ADD TO CHROME" button causes a browser extension installation popup.

The backward and forward tracking graphs for this attack are shown in Figure 10 and 11, respectively. The backward tracking graph is quite similar to the malware download case (though the ad-delivering and extension serving domains are different), and we therefore show only part of it, for space reasons. The forward tracking graph is more complex. The reason is that the install.getsportscore[.]com site, which lures the user into installing the extension, contains a large amount of *user tracking* code (due to space constraints, we omit a detailed analysis of the tracking code). However, the mechanism that triggers Chrome's extension installation authorization popup is fairly straightforward, and can be seen in both the backward and forward tracking graphs. Specifically, the JS code at install.getsportscore[.]com uses jQuery to first register a callback on mouse clicks, as shown in the attack code snippet below (extracted from our audit logs).

```
$addToBrowser.click(function (e) {
    e.preventDefault();
    installExtension();
});
```

[7] The User-Agent string used during the recording of the previous malware download attack was purposely set to advertise a Mac OS machine, rather than a Linux machine

[8] Simply search for: chrome "Sports Score" extension adware

[9] http://www.virusradar.com/en/JS_Adware.StreamItOnline/map/day

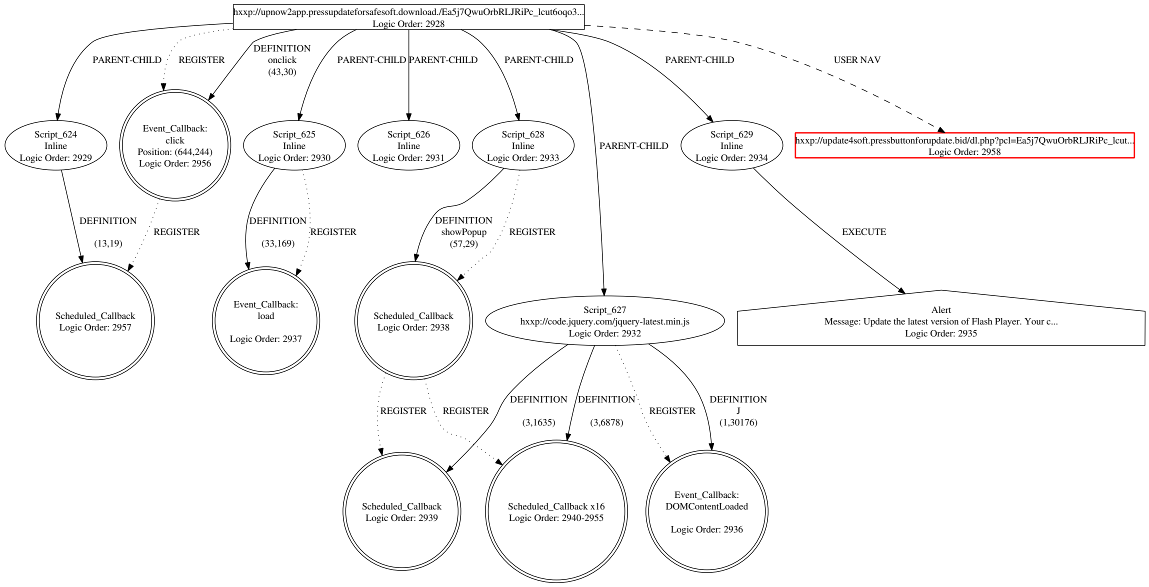
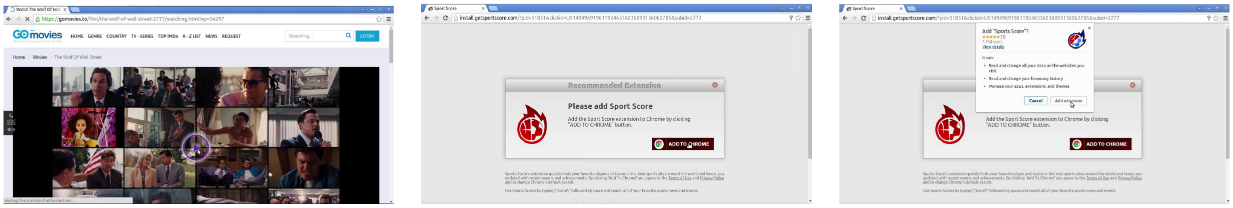


Fig. 8: Forward tracking of a social engineering malware download attack.



(a) (b) (c)

Fig. 9: In-the-wild social engineering extension download attack

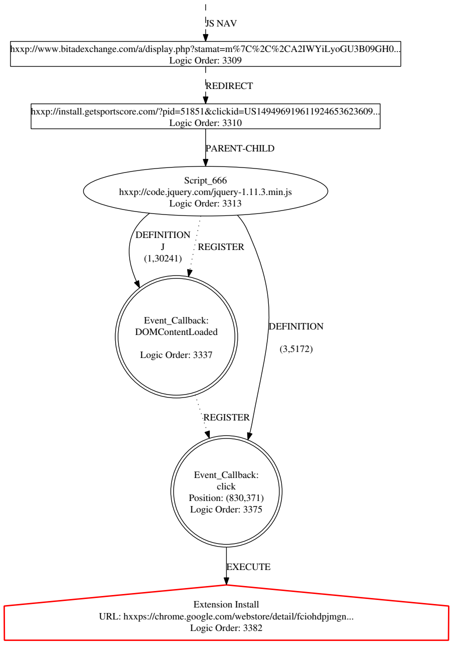


Fig. 10: Extension download attack: backtracking graph (partial)

The jQuery library translates the above code into the registration of two callbacks: one on the `DOMContentLoaded` event, which in turn registers a callback for click events on the “ADD TO CHROME” button shown in Figure 9b.

C. *XSS Attack Analysis*

We now discuss an attack based on an XSS vulnerability on the PHPEcho CMS 2.0-rc3, a content management system (this vulnerability was first disclosed by Jose Luis Gongora Fernandez in June 2009 [18]). We use this vulnerability to conveniently reproduce a possible XSS-driven *phishing attack using a keylogger* to steal Facebook login credentials. To reproduce the attack, we deploy PHPEcho CMS 2.0-rc3 on a virtual machine with CentOS 5.11, Apache 2.2.3, PHP 5.1.6, and MYSQL 5.0.95, to satisfy PHPEcho’s software dependencies. We then leverage third-party attack code to trigger the XSS vulnerability, and launch the phishing attacks.

We reproduce the Facebook phishing attack by making use of a JS-based key-logger adapted from [18]. First, using the XSS vulnerability, a fake Facebook login user interface is injected and forced to alway appears in the middle of the page, as shown in Firgure 12a. A site visitor may get confused by this window, and type in their username and password to make the window disappear. In the background, a key-logger captures the victim’s keypresses and sends them to the attacker in real time. Even if the victim realized that this may be a phishing attempt before submitting the credentials, the attacker will have gained precious information that may be used for reducing the search space in a following brute-force attack, or other social

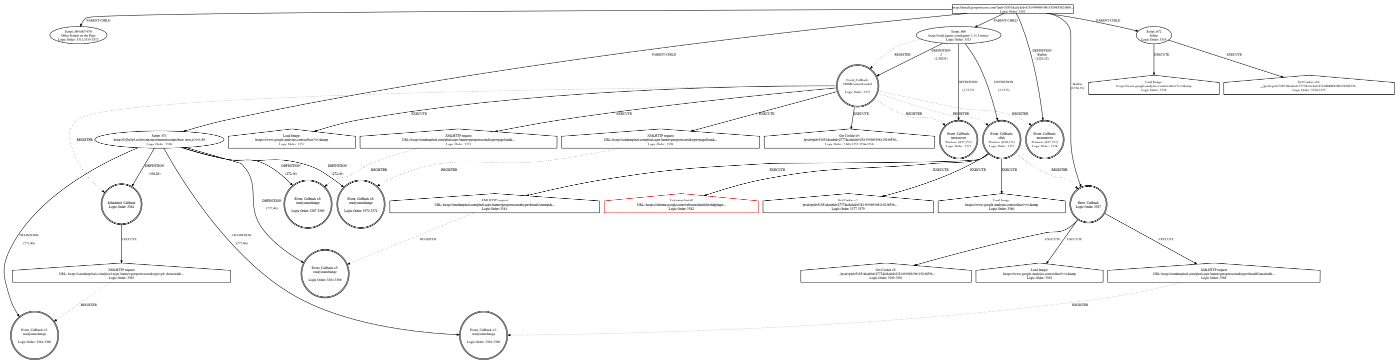


Fig. 11: Extension download attack: forward tracking graph

engineering efforts, for example.

To identify similar attacks in the audit logs, an analyst may start by looking for frequent callbacks triggered by keypress events, paired with critical events such as *XMLHttpRequests*, loading a third-party image, iframe, etc., that may be used to exfiltrate the stolen information. In our specific example, the analysis may start from the pair of keypress event callback and loading of a third-party image, as highlighted in red in Figure 12b. An analysis of the (partial) backward tracking graph, drawn by starting from those events, shows that *Script_62* is responsible for registering the keypress callbacks. Also, the script registers a scheduled callback that periodically loads an external image. Looking at the image's URL parameters, we can notice that this is likely used to encode the key code captured by the keypress callback, thus sending them to the attacker. From the forward tracking graph in Figure 12c, which was drawn starting from the page that contains *Script_62*, we can see that the scheduled callback defined by *Script_62* at line 15, column 27, is activated multiple times during the attack (once every 200 milliseconds, via a `setTimeInterval`), and that every time it is called, it loads the same third-party image with different parameter values.

V. Performance Evaluation

In this section, we present a set of experiments dedicated to measuring the overhead introduced by our JSgraph browser instrumentations.

A. Experimental Setup

JSgraph is built upon Chromium's codebase version 48.0.2528.1. Our source code modification amount to approximately 2,400 lines of C++ code, 150 lines of IDL code and 800 lines of Python code. We plan to make JSgraph available at https://github.com/perdisci/JSgraph. To evaluate the overhead imposed by our code changes to Chromium, we performed three different sets of experiments using both Linux and Android systems, as described below. In all experiments, we leveraged Chromium's `TRACE_EVENT` instrumentation infrastructure [5] to accurately measure the time spent executing our instrumentation code, and to create the baseline performance measurements needed to compute the relative overhead introduced by JSgraph.

Linux – automated browsing (Linux Top1K): The goal of this experiment is to measure the performance of JSgraph on a large set of popular websites. To this end, we leverage the list of top 1,000 most popular websites according to Alexa.com. Because it is very time consuming to manually visit all these websites, we created an automated browsing process. Specifically, we implemented a tool that allows us to automatically visit the top 1,000 websites, and browse on each one for about two minutes. To roughly mimic the browsing behavior of a human user, during the two minute time interval, our system clicks on three randomly selected links, in an attempt to navigate through different pages on each site. For this, we leverage `xdotool`[10], and program it to send a random number of *Tab* plus *Enter* keystrokes, to simulate a click on a random link. To account for variability in the performance measurement due to random inputs, we visit each website 5 times. Overall, our automated browsing system spent about 167 hours browsing on these top websites. In order to perform this experiment, we used a machine with 32 CPU cores (AMD Opteron 6380) and 128 GB of RAM, and 10 QEMU-based virtual machines running Linux Ubuntu 14.04.

Linux – manual browsing (Linux Top10): With this experiment, we further explore JSgraph's performance on ten top US websites. This includes performing searches on Google, watching videos on Youtube, browsing on Facebook, sending emails in Gmail, posting tweets on Twitter, browsing on Reddit, etc. We used JSgraph to manually browse on each of these highly dynamic websites for about five minutes, using a Linux-based Dell Inspiron 15 laptop with a Core-i7 Intel CPU and 8GB of RAM.

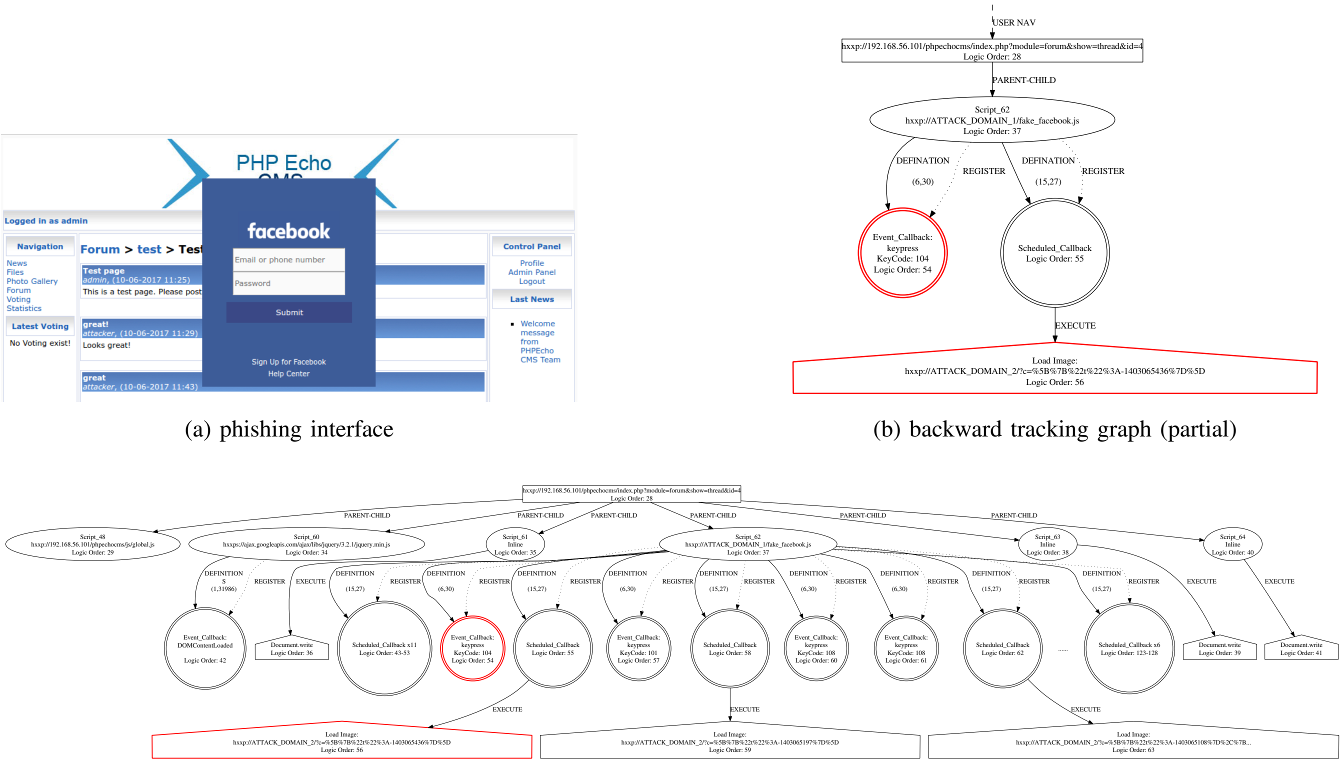
Android - manual browsing (Android Top10): We repeated the experiment outlined above on an Android-v6.0 Google Pixel-C tablet with an Nvidia X1 quad-core CPU and 3GB of RAM. To this end, we compiled an APK version of JSgraph, and used the `adb` bridge to collect JSgraph audit logs and `TRACE_EVENT` measurements for analysis.

B. Performance Traces

We now provide some details on how we leveraged Chromium's `TRACE_EVENT` instrumentation infrastructure for profiling JSgraph's performance. We use three types of trace events: `TRACE_EVENT0`, `TRACE_EVENT_BEGIN0`, and `TRACE_EVENT_END0`.

When placed at the beginning of a function, `TRACE_EVENT0` records the execution time spent on executing the whole function. We add this at the beginning of all JSgraph's instrumentation hooks. In addition, we add `TRACE_EVENT_BEGIN0` to `didStartProvisionalLoad` to monitor the exact time

[10] https://github.com/jordansissel/xdotool


PHP Echo CMS
Logged in as admin
Navigation
News
Files
Photo Gallery
Forum
Voting
Statistics
Latest Voting
No Voting exist!
Forum > test > Test
Test page
admin, (10-06-2017 11:25)
This is a test page. Please post
great!
attacker, (10-06-2017 11:29)
Looks great!
great
attacker, (10-06-2017 11:43)
facebook
Email or phone number
Password
Submit
Sign Up for Facebook
Help Center
Control Panel
Profile
Admin Panel
Logout
Last News
Welcome message from PHPEcho CMS Team
USER NAV
hxxp://192.168.56.101/phpechocms/index.php?module=forum&show=thread&id=4
Logic Order: 28
PARENT-CHILD
Script_62
hxxp://ATTACK_DOMAIN_1/fake_facebook.js
Logic Order: 37
DEFINATION
(6,30)
REGISTER
DEFINATION
(15,27)
REGISTER
Event_Callback: keypress KeyCode: 104 Logic Order: 54
Scheduled_Callback Logic Order: 55
EXECUTE
Load Image: hxxp://ATTACK_DOMAIN_2/?c=%5B%7B%22t%22%3A-1403065436%7D%5D Logic Order: 56
Script_48 hxxp://192.168.56.101/phpechocms/js/global.js Logic Order: 29
Script_60 hxxps://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js Logic Order: 34
Script_61 Inline Logic Order: 35
Script_63 Inline Logic Order: 38
Script_64 Inline Logic Order: 40
DEFINITIONS (1,31986)
DEFINITION
EXECUTE
Event_Callback: DOMContentLoaded Logic Order: 42
Document.write Logic Order: 36
Scheduled_Callback x11 Logic Order: 43-53
Event_Callback: keypress KeyCode: 101 Logic Order: 57
Scheduled_Callback Logic Order: 58
Event_Callback: keypress KeyCode: 108 Logic Order: 60
Event_Callback: keypress KeyCode: 108 Logic Order: 61
Scheduled_Callback Logic Order: 62
......
Scheduled_Callback x6 Logic Order: 123-128
Document.write Logic Order: 39
Document.write Logic Order: 41
Load Image: hxxp://ATTACK_DOMAIN_2/?c=%5B%7B%22t%22%3A-1403065197%7D%5D Logic Order: 59
Load Image: hxxp://ATTACK_DOMAIN_2/?c=%5B%7B%22t%22%3A-1403065108%7D%2C%7B... Logic Order: 63

(a) phishing interface

(b) backward tracking graph (partial)

(c) forward tracking graph

Fig. 12: Analysis of phishing attack with key-logger

when a user navigation is request, and to *CallFunctionStart* and *RunCompiledScriptStart* to monitor the start of each JavaScript code execution. Furthermore, we add `TRACE_EVENT_END0` to *CallFunctionEnd* and *RunCompiledScriptEnd*, to record the end of each JavaScript code execution, and allow us to separately analyze JS execution time from page/DOM construction and idle times. Also, we inject `TRACE_EVENT_END0` into `loadEventFired`, to monitor the firing of page/frame `load` events.

Using this instrumentation, we measure four types of overhead:

- The *page load* overhead measures the time spent executing JSgraph's code between the time the web page first starts loading and when the `load` event[11] is fired for that same page. The baseline is represented by the execution time spent by the browser (excluding the time spent into JSgraph's hooks) between calls to the `didStartProvisionalLoad` and `loadEventFired` instrumentation hooks.
- Similarly, the *DOM construction* overhead measures the time spent by JSgraph's code (and related baseline execution time) in between when the first DOM node is inserted in the DOM tree for the page and when the user triggers the navigation to a new page (excluding the time spent in JS execution).
- The *JS execution* overhead is measured by considering the total time spent by the browser to execute JS code during a given browsing session. Essentially, we sum up all time intervals in between *RunCompiledScriptStart* and *RunCompiledScriptEnd*, and between *CallFunctionStart* and *CallFunctionEnd*.
- The *overall* overhead is measured by considering the entire time spent on a page. For instance, this is often equal to the time in between when a request to load the page is made, and when the user triggers the navigation to a new page. Specifically, we can measure this time interval by measuring the time distance between consecutive calls to the `didStartProvisionalLoad` hook.

In summary, to compute JSgraph's overhead relative to the original Chromium code, we use the following simple formula: $o = \frac{O}{T-O}$, where o is the relative overhead, O is the absolute time spent on JSgraph's code execution, and T denotes the time interval between browser events as discussed above ($T - O$ is the baseline time).

C. Experimental Results

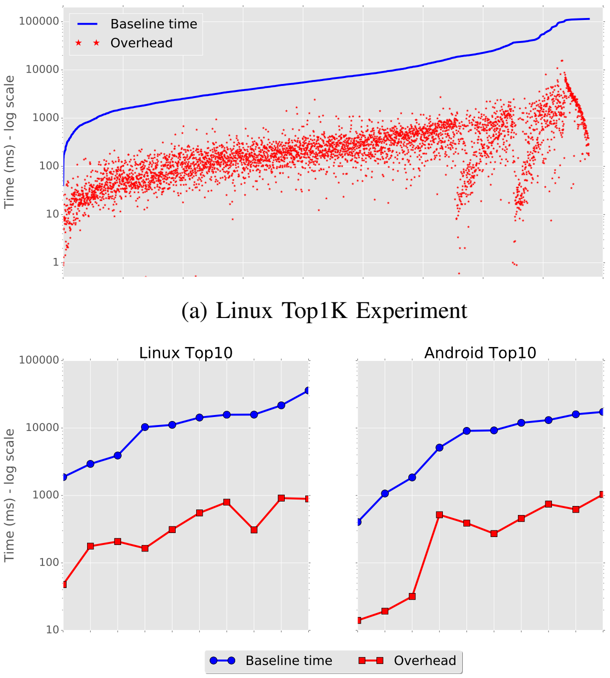
Table I lists the results of the three experiments performed to measure JSgraph's overhead described in Section V-A. Each row indicates the results for one of the three experiments. The columns correspond to the four types of overhead measurements we described in Section V-B. Each table cell reports the median and 95-th percentile of the relative overhead, o, seen during the experiments.

The *page load* column is particularly significant, since high loading time overhead could frustrate a user and drive them away from a web page (the relation between page load time and user satisfaction has been established in previous research [13]). As can be seen from Table I, the 95-th percentile for the page load overhead is at most 8.2%.

[11] https://developer.mozilla.org/en-US/docs/Web/Events/load

TABLE I: *Performance overhead (50th- and 95th-percentile) percentage overhead*

Experiment	Overall	Page load	DOM Construction	JS Execution
Linux Top1K	0.5%, 3.1%	3.2%, 7.4%	0.2%, 1.6%	6.8%, 20.1%
Linux Top10	1.6%, 3.7%	3.3%, 5.7%	0.6%, 1.2%	9.6 %, 17.1%
Android Top10	1.5%, 4.7%	3.9%, 8.2%	0.4%, 1.7%	10.2%, 17.3%



(a) Linux Top1K Experiment

(b) Linux Top10 and Android Top10 Experiments

Fig. 13: Overhead and baseline execution time for *page loads*

Linux Top1K experiment results indicate the median page load overhead is only about 3.2%. The JS execution time overhead median value is also low, at 6.8%. Note that the results for *Linux Top10* and *Android Top10* experiments are also very similar, even though those experiments involved very active browsing by a human user.

The three graphs in Figure 13 provide further insight into the performance of JSgraph during the *page load* phase of all the experiments reported in Table I. The X-axis represents the number of domains crawled during the experiment, while the Y-axis represents time in microseconds, in log scale. In all the graphs, the solid blue curve represents the base execution time (i.e., $T - O$) spent by the browser, excluding any JSgraph overhead. The curve is obtained by plotting the absolute execution time for each website visit (i.e. each domain will be represented at multiple points on the X-axis). The instances are arranged in increasing order of the baseline execution time. The red marker indicates the overheads introduced by JSgraph. We can see that in all the 3 graphs the overhead is about one order of magnitude smaller than the baseline execution time.

D. Dromaeo Performance Benchmark

To further analyze the overhead introduced by JSgraph, we make use of Dromaeo, a JavaScript performance benchmark suite from Mozilla (see dromaeo.com). Using a modern laptop running Ubuntu Linux, we ran the Dromaeo tests two times: (1) with JSgraph enabled, thus including the overhead discussed in Section V-B; and (2) with JSgraph disabled, so that our instrumentation hooks are not called by Chromium.

With JSgraph enabled, the browser was able to perform 4143 runs/s[12]; whereas with JSgraph disabled, the browser performed 4341 runs/s[13]. Using the relative overhead definition defined in Section V-B, this translates to about 4.6% overhead. These results show that JSgraph performed approximately as in the Linux Top10 experiments (on the same device) reported in the *JS Execution* column of Table I.

E. Storage Requirements

The storage requirements for JSgraph are limited. In the experiments reported in Table I, rows 1-2 (Linux-based experiments), we observed that a total of 50 minutes of very active browsing on 10 highly dynamic, popular websites resulted in 37 MB of compressed audit logs. This means the average disk space requirement is only about 0.74 MB per minute of active browsing. Assuming 8 hours of active browsing per work day, multiplied by 262 workdays per year, gives us less than 84GB of audit logs per network user per year, or less than 84TB of storage for 1,000 network users, for one entire year. For mobile devices, this requirements reduce even further, to 0.34 MB/minute, or less than 42TB of storage for 1,000 network users for one year. This is likely due to the more limited web content typically delivered by websites to resource-constrained mobile devices. Considering the low cost of archival storage, this represents a sustainable cost for an enterprise network.

VI. Discussion

Our proof-of-concept implementation of JSgraph has some limitations. For instance, as discussed in Section II, with more engineering effort we could instrument all Blink/V8 bindings that have an impact on any aspect of the page. However, we should notice that our current instrumentations capture all such bindings that are activated by JS code running on popular websites. Therefore, adding audit log instrumentation to rarely used APIs is unlikely to significantly affect our overhead estimates, for example.

We should also point out that while the Chromium code based tends to evolve fairly rapidly, porting JSgraph to newer versions of Chromium is possible with reasonable effort. In fact, a large part of the effort for our research team was to design the system and identify how to extend the DevTools instrumentation infrastructure to enable the necessary fine-grained audit logs without introducing high overhead or altering the browser's functionalities. Now that this research task has been performed, and because the DevTools inspector instrumentation infrastructure is fairly stable, porting our efforts to newer versions of Chrome mostly involves engineering time. This also implies that, with adequate engineering effort, JSgraph updates could be deployed with a timeline comparable to Chrome browser releases. Furthermore, to facilitate deployability JSgraph could integrate a way for administrators to enable/disable logging, or to whitelist highly sensitive websites that should be excluded from recording.

[12] Archived results: http://dromaeo.com/?id=268497

[13] Archived results: http://dromaeo.com/?id=268495

VII. Additional Related Work

Along with the previous works discussed in Section I-C, there exist other studies that are related to JSgraph from different aspects, as discussed below.

Graph-based Forensic Analysis. Causal graphs that show the causality relations between subjects (e.g., process) and objects (e.g., file) are widely used in system-level attack analysis [24], [15], [23], [25], [27]. They record important system events (e.g., system calls) at runtime and analyze them in a post-mortem attack analysis. Recently, a series of works [28], [33], [32] have proposed to provide accurate and fine-grained attack analysis. They divide long-running processes into multiple autonomous execution units and identify causal dependencies between units. A node in their causal graphs represents fine-grained execution unit instead of a process in the previous system call based approaches and an edge shows causal relations between those units. Bates at el. [1] propose a novel technique for auditing data provenance of web service components, called Network Provenance Functions (NPFs).

Dynamic taint analysis techniques [40], [21], [17] can also be used for causality analysis. They monitor each program instruction to identify data-flow between system components (e.g., memory object, file, or network). A causal graph constructed by the taint analysis shows data-flow between those system components.

These techniques present causal relations between system or network components, however, it is difficult to understand JavaScript execution from their analysis due to a large semantic gaps between system-level events and JS execution inside a browser. JSgraph can complement these techniques and fill the gap by providing detailed behaviors of JavaScript execution. For instance, incorporating JSgraph with a system-level analysis technique will enable seamless reconstruction of both system-level and in-browser attack provenance.

Record and Replay: System-level record and replay (R&R) techniques [12], [23], [15], [10], [36] have been proposed to allow forensic analysis or to recover the system from the attack. System-level record and replay systems might not be very helpful to analyze what happend inside the web-browser because there is a large semantic gap between the system-level events (i.e., system call) and the high-level events happen inside the browser such as interaction between the JavaScript engine (e.g., V8) and the rendering engine (e.g., Blink).

As we discussed earlier, Web-browser R&R systems [4], [37] and JavaScript R&R techniques [35], [44] have been proposed, however, they have limitations to allow accurate forensic analysis of JS execution. Details are discussed in section I-C.

Static JS Analysis: A few static analysis techniques have been proposed to identify malicious JS code [9], [14]. For example, ZOZZLE [9] classifies JS code based on contextual information from the abstract syntaxt tree (AST) of the program. Caffein Monkey [14] identifies malicious JS code based on the usage of obfuscations and methods in the program. However, the dynamic features of JavaScript make it difficult to statically analyze JS code.

Dynamic JS Analysis: Dynamic anlaysis is widely used to monitor dynamic behaviors of JS programs. Cova et al. [8] developed a system that can detect and analyze malicius JS codes by executing them in the emulated environment. They extract a number of features from the JS code execution and use machine learning techniques to identify the characteristics of malicious JS programs. There are a number of symbolic execution techniques for JavaScript have been proposed such as SymJS [30] Kudzu [41], Jalangi [42]. Recently, a forced execution engine for JavaScript, called J-Force [22], has proposed to identify possible malicious execution paths from the JS code. J-Force iteratively explore execution paths until all possible paths are covered including the hidden paths by event and exception handlers. Symbolic executions and forced execution techniques for JavaScript are generally heavy-weight and requires special execution environment (e.g., VM-based framework) as they focus on off-line analysis to reveal security issues. On the other hand, JSgraph focuses on recording the "real" attacks as we discussed in Section I-C.

VIII. Conclusion

We proposed JSgraph, a forensic engine aimed at efficiently recording fine-grained audit logs related to the execution of JavaScript programs. JSgraph's main goal is to enable a detailed, post-mortem reconstruction of ephemeral JS-based web attacks experienced by real network users, with particular focus on social engineering attacks.

We implemented JSgraph by instrumenting Chromium's code base at the interface between Blink and V8, and design our system to be lightweight, highly portable, and to require low storage capacity for its fine-grained audit logs. Using a number of both in-the-wild and lab-reproduced web attacks, we demonstrated how JSgraph can aid the forensic investigation process. We also showed that JSgraph introduces acceptable overhead on the browser, which could be further reduced with some more engineering effort to perform code optimizations.

Acknowledgments

We thank Adam Doupé for serving as our shepherd, and the anonymous reviewers for their constructive comments and suggestions for improvement.

This material is based in part upon work supported by the National Science Foundation, under grant No. CNS-1149051, and by the United States Air Force and Defense Advanced Research Agency (DARPA), under Contract No. FA8650-15-C-7562. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or DARPA.

References

[1] A. Bates, W. U. Hassan, K. Butler, A. Dobra, B. Reaves, P. Cable, T. Moyer, and N. Schear, "Transparent web service auditing via network provenance functions," in *International Conference on World Wide Web*, ser. WWW '17, 2017.

[2] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian, "Run-time monitoring and formal analysis of information flows in Chromium," in *Annual Network and Distributed System Security Symposium*, 2015.

[3] K. D. Bowers, C. Hart, A. Juels, and N. Triandopoulos, "Pillarbox: Combating next-generation malware with fast forward-secure logging," in *Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[4] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive record/replay for web application debugging," in *ACM symposium on User interface software and technology*. ACM, 2013, pp. 473–484.

[5] Chromium Project, "Adding traces to chromium/webkit/javascript," https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/tracing-event-instrumentation.

[6] ——, “Content module,” https://www.chromium.org/developers/content-module.

[7] ——, “Web idl in blink,” https://www.chromium.org/blink/webidl.

[8] M. Cova, C. Kruegel, and G. Vigna, “Detection and analysis of drive-by-download attacks and malicious javascript code,” in *International Conference on World Wide Web*, ser. WWW ’10, 2010.

[9] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, “Zozzle: Fast and precise in-browser javascript malware detection,” in *USENIX Conference on Security*, ser. SEC’11. Berkeley, CA, USA: USENIX Association, 2011, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028070

[10] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, “Eidetic systems,” in *USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’14. Berkeley, CA, USA: USENIX Association, 2014, pp. 525–540. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685090

[11] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, “Repeatable reverse engineering with panda,” in *Program Protection and Reverse Engineering Workshop*, ser. PPREW-5, 2015.

[12] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, “Revirt: Enabling intrusion analysis through virtual-machine logging and replay,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, Dec. 2002.

[13] S. Egger, P. Reichl, T. Hoßfeld, and R. Schatz, “”time is bandwidth”? narrowing the gap between subjective time perception and quality of experience,” in *IEEE International Conference on Communications*, 2012.

[14] B. Feinstein and D. Peck, “Caffeine monkey: Automated collection, detection and analysis of malicious javascript,” ser. BlackHat’07, 2007.

[15] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, “The taser intrusion recovery system,” in *ACM Symposium on Operating Systems Principles*, ser. SOSP ’05, 2005.

[16] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker, “Manufacturing compromise: The emergence of exploit-as-a-service,” in *ACM Conference on Computer and Communications Security*, ser. CCS ’12, 2012.

[17] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis, “A general approach for effciently accelerating software-based dynamicdata flow tracking on commodity hardware,” in *USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI, 2012.

[18] JosS. (2009) Phpecho cms 2.0-rc3 - (forum) cross-site scripting cookie stealing/blind. [Online]. Available: https://www.exploit-db.com/exploits/9014/

[19] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna, “Escape from monkey island: Evading high-interaction honeyclients,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 124–143. [Online]. Available: http://dl.acm.org/citation.cfm?id=2026647.2026658

[20] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, “Revolver: An automated approach to the detection of evasive web-based malware,” in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 637–652. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/kapravelos

[21] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, “libdft: practical dynamic data flow tracking for commodity systems,” in *ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012.

[22] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, “J-force: Forced execution on javascript,” in *International Conference on World Wide Web*, ser. WWW ’17, 2017.

[23] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, “Intrusion recovery using selective re-execution,” in *USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’10, 2010.

[24] S. T. King and P. M. Chen, “Backtracking intrusions,” in *ACM Symposium on operating systems principles*, ser. SOSP ’03. ACM, 2003.

[25] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, “Enriching intrusion alerts through multi-host causality,” in *Network and Distributed System Security Symposium*, ser. NDSS’05, 2005.

[26] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, “Rozzle: De-cloaking internet malware,” in *IEEE Symposium on Security and Privacy*, 2012.

[27] S. Krishnan, K. Z. Snow, and F. Monrose, “Trail of bytes: efficient support for forensic analysis,” in *ACM conference on Computer and communications security*, ser. CCS ’10. ACM, 2010.

[28] K. H. Lee, X. Zhang, and D. Xu, “High accuracy attack provenance via binary-based execution partition,” in *Network and Distributed System Security Symposium*, ser. NDSS, 2013.

[29] 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.: USENIX Association, 2015, pp. 723–735. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies

[30] G. Li, E. Andreasen, and I. Ghosh, “Symjs: Automatic symbolic testing of javascript web applications,” in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[31] Linux Man Pages, “Chattr,” http://man7.org/linux/man-pages/man1/chattr.1.html.

[32] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, “MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning,” in *USENIX Conference on Security Symposium*, ser. Usenix Security, 2017.

[33] S. Ma, X. Zhang, and D. Xu, “Protracer: Towards practical provenance tracing by alternating between logging and tainting,” in *Network and Distributed System Security Symposium*, ser. NDSS, 2016.

[34] G. A. Marson and B. Poettering, “Even more practical secure logging: Tree-based seekable sequential key generators,” in *19th European Symposium on Research in Computer Security - Volume 8713*, ser. ESORICS 2014, 2014.

[35] J. Mickens, J. Elson, and J. Howell, “Mugshot: Deterministic capture and replay for javascript applications,” in *USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855722

[36] Mozilla, “Record and replay framework,” http://rr-project.org/.

[37] C. Neasbitt, B. Li, R. Perdisci, L. Lu, K. Singh, and K. Li, “Webcapsule: Towards a lightweight forensic engine for web browsers,” in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15, 2015.

[38] C. Neasbitt, R. Perdisci, K. Li, and T. Nelms, “Clickminer: Towards forensic reconstruction of user-browser interactions from network traces,” in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14, 2014.

[39] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad, “Towards measuring and mitigating social engineering software download attacks,” in *USENIX Conference on Security Symposium*, ser. SEC’16, 2016.

[40] J. Newsome and D. X. Song, “Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software,” in *Network and Distributed System Security Symposium*, ser. NDSS ’05, 2005.

[41] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for javascript,” in *IEEE Symposium on Security and Privacy*, 2010.

[42] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, “Jalangi: A selective record-replay and dynamic analysis framework for javascript,” in *Joint Meeting on Foundations of Software Engineering*, 2013.

[43] P. Vadrevu, J. Liu, B. Li, B. Rahbarinia, K. H. Lee, and R. Perdisci, “Enabling reconstruction of attacks on users via efficient browsing snapshots,” in *Network and Distributed System Security Symposium*, ser. NDSS, 2017.

[44] J. Vilk, J. Mickens, and M. Marron, “ReJS: Time-travel debugging for browser-based applications,” in *Microsoft Research – Technical Report*, 2016.

[45] B. Wu and B. D. Davison, “Detecting semantic cloaking on the web,” in *International Conference on World Wide Web*, ser. WWW ’06, 2006.