ScriptChecker: To Tame Third-party Script Execution With Task Capabilities

Wu Luo*, Xuhua Ding†, Pengfei Wu‡, Xiaolei Zhang*, Qingni Shen* and Zhonghai Wu*
*Peking University
Email: {lwyeluo,xl_zhang,qingnishen,wuzh}@pku.edu.cn
†Singapore Management University
Email: xhding@smu.edu.sg
‡School of Computing, National University of Singapore
Email: pfwu@nus.edu.sg

***Abstract*—We present ScriptChecker, a novel browser-based framework to effectively and efficiently restrict third-party script execution according to the host web page's directives. Different from all existing schemes functioning at the JavaScript layer, ScriptChecker holistically harnesses context separation and the browser's security monitors to enforce on-demand access controls upon tasks executing untrusted code. The host page can flexibly assign resource-access capabilities to tasks upon their creation. Reaping the benefits of the task capability approach, ScriptChecker outperforms existing techniques in security, usability and performance. We have implemented a prototype of ScriptChecker on Chrome and rigorously evaluated its security against 1373 malicious scripts and its usability with empirical studies upon top-1000 sites. The experimental results show that its strong security strength and ease-of-use are attained at the cost of unnoticeable performance loss. It incurs about 0.2 microseconds overhead to mediate a DOM access, and 5% delay when loading popular JS graphics and utility libraries.**

I. Introduction

Third-party scripts provide ready-to-use functions for specialized utilities (e.g., `jQuery` [15] and `CryptoJS` [13]) and Web feature enrichment (e.g., Yahoo Partner Ads [19] for advertisement and `userreport` [18] for user tracking). According to Nikiforakis et al. [44], 88.45% of web sites include at least one third-party script. They are greatly attractive to web page developers thanks to the remarkable saving in implementation. However, it has been known for years that invoking a third-party script function potentially invites unintended or even malicious code execution with the full privileges to access all resources [44], [37], [36], [33], [53].

The third-party script security problem is challenging for two reasons. First, a host script and a third-party script are co-located in the same host frame and therefore have the same origin and frame identity. Thus, frame oriented or origin oriented policies and mechanisms, e.g., the same-origin policy [16], iframe isolation [6], the feature-policy header [20], and CSP [11], are unable to differentiate and handle them separately. It is hence necessary for a solution to draw the boundary within a frame between the host script and the third-party script. Second, there is no single policy to uniformly govern all third-party scripts for all websites. It is the host page, not the browser, that is entitled to decide what resources are accessible to a third-party script and what are not. Moreover, a script needs different permissions for different functionalities. Thus, a desirable third-party script restriction scheme should allow the host to dynamically specify the security demand.

Earlier work [42], [52] hooks function calls to third-party code and uses filters to decide whether the requested invocation is allowed. More recent work isolates the script with specialized contexts [50], [43], [34], [29] or JS scope [23], [51], [2]. A reference across the isolation boundary is trapped and checked by security functions to block harmful references. The idea underpinning both approaches is to have a JavaScript layer reference monitor intervening in the untrusted execution and then granting or denying object references properly. Centering on JS code, existing schemes face limitations in performance, usability and more notably security. We briefly explain their security drawbacks below with an elaboration in Section II.

Security of existing schemes is undermined by two issues. In order to attain watertight security, they require security checks on *each* and *all* JS APIs/objects relevant to concerned resources. Although theoretically feasible, the requirement is highly demanding for developers. An unattended API can be used by the adversary to bypass the restriction. Hence, the schemes are error-prone in practice. Moreover, to detect malicious accesses only at the juncture of JS object referencing results in false negatives, i.e., security breaches such as cookie leakage. A seemingly innocuous JS function call from a third-party script function may turn out to be a Confused Deputy Attack [31], because the malicious caller can influence (or even manipulate) the subsequent control/data flow of the callee function.

In this paper, we propose **ScriptChecker**, a novel browser-based discretionary access control framework targeting third-party scripts. At the core of ScriptChecker is that a host script invokes third-party script code in a separate *unit for execution scheduling*[1] and properly assigns it with a capability in the form of a permission set. The browser's security monitor treats these tasks as subjects and allows/denies their

Network and Distributed Systems Security (NDSS) Symposium 2022
27 February - 3 March 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24382
www.ndss-symposium.org

[1]In Chrome, it is called a task which is the most primitive unit for scheduling [10].

resource accesses according to their attached capabilities. Since capability enforcement is undertaken by the browser instead of the JS engine, ScriptChecker incurs insignificant performance loss and has the built-in immunity against the confused deputy attack. Web developers only need to articulate their security policies concerning the host's sensitive resources, with the flexibility to dynamically assign permissions. They are relieved from the burden of meticulously going through each relevant JS API. The host-policy-browser-enforcement paradigm used in ScriptChecker resembles CSP. However, CSP policies are static while the host using ScriptChecker can dynamically set policies. Moreover, CSP policies are against scripts while policies in ScriptChecker are against tasks.

We have implemented ScriptChecker in Chromium with case studies explaining how to dynamically specify permissions for different runtime needs. We test ScriptChecker against 1373 malicious scripts in a dataset. It successfully blocks all illegal accesses including those from event listeners. Our empirical study on Alexa top-1000 websites reveals how real-life third-party scripts access host resources and whether they are broken by ScriptChecker with a conservative policy. The performance benchmark experiment reports 2 µs overhead (about 9%) in one DOM object access from a third-party script execution. In average, Chrome retrofitted with ScriptChecker drops around 0.3% in JS performance and sees 5% delay in average when loading popular JS graphics and utility libraries.

ORGANIZATION. Section II further sheds light on the problem to explain limitations of existing techniques. Section III overviews the design of ScriptChecker. The three key components of ScriptChecker are elaborated in Section IV, V and VI, respectively. Section VII describes the prototype implementation and case studies. We evaluate its performance in Section VIII and make a detailed comparison with existing work in Section IX. Finally, we discuss several issues in Section X and conclude this paper in Section XI.

II. THE PROBLEM IN CLOSE-UP

A. Limitation of Default Policy Enforcement in Chrome

In Chrome's multi-process architecture [9], [25], the *renderer process* parses web documents, interprets JavaScript and renders web content using multiple threads. Except the Web Worker [21], all JavaScript code are executed in its main thread. The main thread schedules tasks [10] following FIFO to accomplish designated work. The *kernel* process is responsible for providing system services to the renderer processes, including networking, IPC, and disk-resident data such as cookies and bookmarks [25].

Like other popular browsers, Chrome has *security monitors* [25], [35] safeguarding critical and sensitive resources against all scripts. Residing in the renderer process, the *local security monitor* enforces system-wide policies such as Same Origin Policy (SOP) [16], Cross-Origin Resource Sharing (CORS) [12] and Content Security Policy (CSP) [11] when DOM objects, JS objects and network responses are accessed. The *kernel security monitor* resides in the kernel process and mediates accesses to system resources such as cookies, bookmarks and sensors.

These origin- and frame-based policies as well as their enforcement are ill-suited for third-party restriction. Firstly, there is no one-size-for-all policy governing all websites' third-party scripts. For instance, one website may use a third-party script to evaluate the strength of user passwords while most sites would view such password accesses are intrusive. Hence, it is more desirable for the host pages to define the desired restrictions, instead of a universal policy mandatorily enforced. Secondly, the access control framework in Chrome considers the scripts accessing resources as subjects. It is thus susceptible to the confused deputy attack. Moreover, since third-party scripts and host scripts are co-located in the same frame, the security monitors do not have sufficient contextual information to differentiate them based on their origins or frame identities.

However, we observe that the ultimate objective of third-script execution restriction is to prevent their resource misuses, which is congruent to the goal of SOP and frame-based policies. By design, the security monitors have the ability to mediate resource accesses from all scripts including third-party scripts. Its working is independent of the JavaScript APIs. Thus, the security monitor provides stronger security and better performance than JS layer solutions.

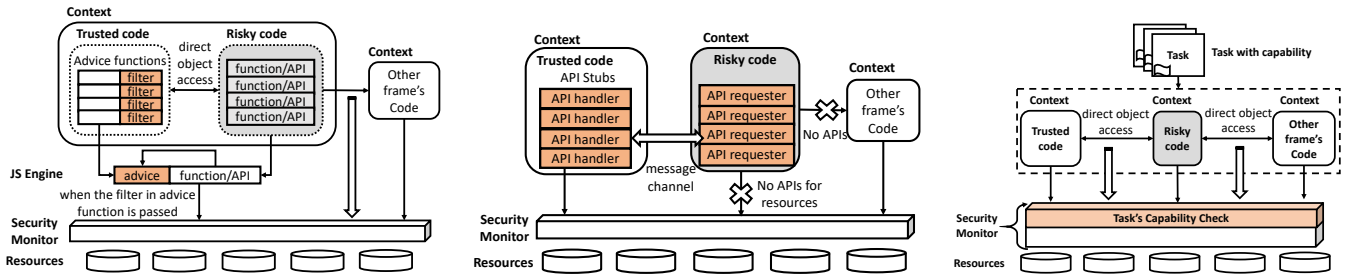
B. Existing Approaches Against Third-Party Scripts

By and large, two approaches are used for third-party script restriction, both functioning at the JavaScript layer. The first, termed the *function filter approach* in this paper, is to hook each security-critical function with a special filter called the *advice* function in ConScript [42] and WebJail [52]. As shown in Figure 1(a), the host page's trusted code registers advice functions in the JS engine. Before a sensitive function is called, the JS engine executes the corresponding advice function to decide whether the function call is allowed.

We term the other approach the *reference restriction approach* as it restricts the third-party code's ability to reference JS objects. Two methods are used for restriction in the literature. One is to relocate the untrusted code to a special context, for instance, a dedicated cross-origin frame in AdJail [50] and Pivot [43], a Web Worker context in TreeHouse [34], and a shadow JS engine in AdSentry [29]. Depending on the context in use, the JS engine controls the code's behavior accordingly. For instance, code in a Worker context is not provisioned with browser APIs for system resource accesses. As depicted in Figure 1(b), due to the change of the context, the third-party functions cannot reference the JS objects in the host frame, and must use the `postMessage` API to send its request [34]. The host page script then uses API stubs to examine the requests and grants/denies them. The other reference restriction method is to declare a dedicated JS scope for third-party code by leveraging JavaScript features [51], [2], [23]. For instance, Jate [51] uses the `with` keyword to define a new JS scope of the third-party script and supplies it with the `proxy` objects only. Without the capability of referencing the real objects in the host script, the confined script only accesses the proxy objects and the JS engine is triggered to run the call-back functions in the host script for policy checking.

C. Limitations of Existing Approaches

Both the function filter and reference restriction approaches have limitations in security and adaptability. The discussion below also reveals the challenges in handling third-party scripts.



(a) Function Filter Schemes. (b) Reference Restriction Schemes (using contexts). (c) ScriptChecker.

Fig. 1. Comparison with *function filter* and *reference restriction* schemes for confining third-party scripts considered as risky code.

Security. Although it is the critical DOM and system resources that demand protection, existing approaches are aimed to stall the attacks from the JavaScript layer. Two issues arise from this strategy of anchoring the countermeasure at the attack vectors. First, the coverage on the attack vectors must be complete. The solution must assess *all* relevant JS functions in the function filter approach and JS objects in the reference restriction approach. For instance, the host page developer must provide filters/call-backs for the long list of APIs – *document.getElementsByTagName*, *document.getElementById*, *document.getElementsByName* and *document.getElementsByClassName*, just to name a few. It is therefore highly demanding and error-prone to come up with a proper list and treat every item therein. An update of the JS engine with new and/or deprecated APIs and objects would entail a re-examination of the attack vectors.

Secondly, these approaches are susceptible to the confused deputy attack [31]. A malicious script can leverage its "legitimate" executions to misuse the host script's code and built-in JS APIs as a deputy to access resources. This problem has been acknowledged by Acker et al. who remarked that the implementation should "restrict access to sensitive operations in other execution contexts" [52]. It is infeasible to block a third-party script's all references to the host JS objects universally, since it impedes the intended functionality. Functions rightfully accessible to the third-party script can become the confused deputy without being caught by existing schemes.

The issue may deteriorate when the host frame applies different policies on multiple third-party scripts. Scripts can collude to complement their respective privileges by calling another's functions. For instance, one script is granted with a cookie permission only and another is granted with network permissions only. A collusion between them essentially gives both to the accesses to cookies and the network.

Adaptability. Existing approaches lack adaptability to meet changing demands at runtime. A third-party script has various ways to interact with the host frame, e.g., to reference a dynamically generated DOM or an event object. As the behavior may not be derived from the static code, existing systems have to create new API stubs or new proxy objects on the fly. Moreover, in order to follow the well-known least-privilege-principle, the security policy governing the third-party scripts may evolve throughout the lifetime of the web page. Different invocations of the same script could be subject to different restraints. However, existing schemes rigidly implement the policies in the form of code (i.e., filters) and objects, which are not easy to be adapted accordingly.

III. Sketch of ScriptChecker

We propose ScriptChecker, a framework that reins third-party script execution at the host page's command. ScriptChecker pivots on Chrome security monitors, with the assistance of the JS engine, to enforce on-demand permission checks upon tasks executing third-party scripts considered as risky by the host. It does not rely on attack vector elimination at the JavaScript layer, and thus avoids the pitfalls in existing schemes.

Trust Model. We trust Chrome sandbox and V8 JS engine including the just-in-time compiler, and do not consider attacks on software implementation vulnerabilities, such as memory corruption exploits [35], [46] and control flow attacks [24], [40], [41]. The adversary in our model is rogue third-party scripts in a host frame with the goal to illegally access sensitive resources such as the host frame's JS/DOM objects and system resources (e.g., network, persistent storage and sensors).

A. Overview

The basic idea of ScriptChecker is to start a third-party script execution in a new Chrome task with a host-specified capability stipulating access control permissions for the browser to enforce. The working of ScriptChecker is illustrated in Figure 1(c). The architecture consists of three key components: (i) a sandbox context for third-party scripts for the purpose of context segregation; (ii) a capability-based discretionary access control system against tasks; and (iii) a set of APIs that asynchronously execute the callee function.

Sandbox Context. We introduce the *sandbox context* for the risky code to run within while other code remains in the default main context. The sandbox context has the full set of JS APIs to access DOM objects and system resources.

The sandbox context serves for a different purpose from the special contexts in by Jate [51] and TreeHouse [34]. The latter two are used to restrict the script's access to JS objects and/or APIs as they are potential attack vectors. Hence, those special context are meticulously designed for a full-scale isolation. In contrast, our sandbox context is for context separation only, in the sense that the third-party's accesses to host JS objects become cross-context and are subsequently checked by the security monitor. It therefore protects the host against the

environment poisoning attack [42], [52] in which the malicious script modifies the legitimate JS functions or variables to gain the advantage of evading policy checking. The host can declare which JS objects to be shared and which not. More details of the sandbox context are in Section IV.

Task Capability System. ScriptChecker provides a permission dictionary for the host script to dynamically compose policies in the form of capabilities regarding accesses to cross-context JS objects, DOM objects and system resources. Capabilities are created and attached to a task during its creation according to either the explicit specification in the host script or inheritance from its parent. Permissions specified in a task's capability are enforced by ScriptChecker via the security monitors at the juncture of Chrome's default policy enforcement (e.g., SOP checking) for all JS scripts. Tasks without capabilities are trustworthy because they do not execute any untrusted JS code. The details of the task capability system are presented in Section V.

Asynchronous Execution. To leverage the capability system to restrict third-party scripts, the web developer must change the way to invoke those risky functions so that they run in a separated task with tailored permissions. For this purpose, ScriptChecker provides a suite of APIs that create a new task for the callee function. Moreover, ScriptChecker also retrofits Chrome to use these APIs to execute risky code during script inclusion and listener execution. The details of asynchronous execution are presented in Section VI.

In short, ScriptChecker differentiates trusted and risky tasks. A trusted one does not run any risky code. While a risk task may execute both risky code and host script code, it is always governed by the assigned capability.

B. An Example

Suppose that the frame developer wants to prevent a third-party script `utility.js` from accessing cookies. With ScriptChecker, the frame developer simply uses the code in Listing 1.

```
1<script src="https://.../utlity.js" risky
     task_capabitity="No_Access" />
2<script>
3   function accessCookie() { return document.cookie;
        }
4   var secret = "user_info";
5   function shareData() { ...}
6   ...
7   // invoke untrusted code with restriction
8   var permissions = "JS_WL:shareData";
9   var data_to_hash = "data_to_hash";
10  var result = "";
11  asynCallCap(function() {
12    result = windowRisky[0].UtilityJS.Processing(
          data_to_hash);
13  }, permissions);
14
15  //followup computation
16  ...
17</script>
```

Listing 1. Example of restricting a third-party script

ScriptChecker modifies Chrome's frame parser to support the `risky` and `task_capability` attributes on the `script` tag. With the `risky` attribute, the parser task places `utility.js` into the sandbox context and creates a risky child task with the capability generated according to the `task_capability` attribute. Function `UtilityJS.Processing` from the utility library is called asynchronously through `asynCallCap()`, one of the APIs provided by ScriptChecker. It creates a child task T to run the callee function with a capability generated according to the second parameter. In this example, the child task is disallowed to access cookies of the parent task's origin, but it can reference `shareData` function.

Like in normal tasks, a resource access from T is trapped into Chrome's security monitor. Before performing the default policy, the security monitor examines whether the access is compliant with its capability. Suppose that the downloaded `utility.js` in Listing 1 is malicious. It will never manage to read the frame's cookie regardless the functions it calls.

Among all JS objects in the host script, T can only reference `shareData`. Note the even if the developer mistakenly includes `JS_WL:accessCookie` in Line 8, it still cannot access the cookie even if calling `accessCookie` function. This is because the security monitor is the final gatekeeper for resource access and denies the access based on T's capability.

The example already shows advantages of ScriptChecker over existing schemes from the developer's perspective. The developer specifies the resources to protect without considering how the adversary may access them. The restriction can be flexibly set for the host script's each invocation of the utility function according to the runtime security demand.

IV. Context Separation

In the following, we describe the sandbox context and explain how cross-context references are supported and checked. The main purpose of placing a risky script to the sandbox context is to cope with the environment poisoning attack [42], [52] wherein the adversary modifies trusted code and/or its dependent JS objects to bypass security checks.

A. Sandbox Context

ScriptChecker introduces the `risky` attribute in the `script` tag and the *sandbox context* as a new type of context. When encountering the `risky` attribute, the parser creates a new sandbox context and loads the script to it instead of the default main context. Risky scripts do not share the sandbox context since they may execute with different permissions.

The sandbox context is initialized with the default built-in JS objects and functions as shown in Figure 2. Unlike the Web Worker context used in [34] with reduced APIs, the sandbox context has the *full* suit of APIs to reference DOM and system resources as the main context does. Chrome's JS binding module provides the same view of DOM and system resources to both the sandbox context and the main context. Hence, *no* change is needed for the third-party script to access DOM objects, system resources, and built-in JS objects.

Frame Structure. The risky script uses objects relevant to the frame structure (e.g., `parent` and `frames`) to reference other frames or send messages in the same way as in the main context. It still references its hosting frame

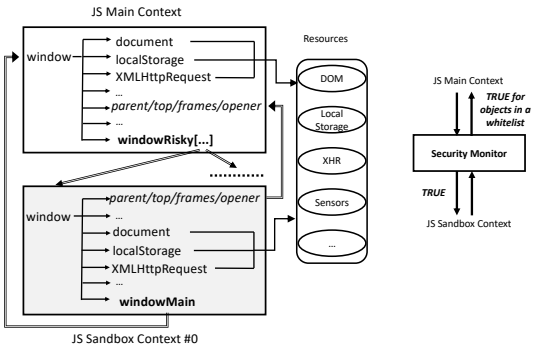


Fig. 2. A frame's main context v.s. sandbox context. The **windowRisky** and the **windowMain** are two new fields introduced to the contexts.

using `document.defaultView` for which Chrome returns the sandbox context's own `window` object whose associated Blink objects (e.g., `LocalDOMWindow`) are cloned from their counterparts of the main context respectively.

Event Listeners. A risky script can register its event listeners to DOM elements. When the event is triggered, listeners run in the respective contexts which they are registered from. The JS `window` object supports event binding as well (e.g., the `load` event). Chrome locates and executes both listeners in the main context and sandbox context.

Code Extension. A risky script in its sandbox context may extend its code by embedding another script or using an inline script. We modify Chrome to ensure that the added code remains in the sandbox context. For script embedding, the `risky` attribute is attached to the newly created `script` tag. Once the `risky` attribute is set, ScriptChecker tags the script element to be risky and never removes this tag in the further processing. Hence, restricted tasks cannot reset the script to be non-risky. For inline script and code sinks such as `eval`, ScriptChecker utilizes the same strategy by tagging the new code with proper non-removable attributes.

Cross-Context Reference. Since ScriptChecker relocates the third-party script from the main context to the sandbox context, the references between the two scripts become cross-context. To support it, we make minor changes on both of them as shown in Figure 2. We add a `windowRisky[]` object to the main context and respectively. Each element in the `windowRisky[]` array points to a sandbox context confining a third-party script. Thus, the host script can enclose multiple third-party scripts and isolate each of them with different permissions. We add the `windowMain` object to the sandbox context. A third-party script uses it to reference JS objects in the main context. Note that for compatibility and modularity reasons, it is seldom for a third-party script to directly reference JS objects created by the host script.

B. Oneway Cross-Context Isolation

If a risky script alters its environment objects and functions in its own context, there is no adverse effects on scripts in other contexts. Hence, the adversary has to make cross-context references to launch the environment poisoning attack [42], [52]. By default, Chrome's local security monitor checks all such cross-context references.

ScriptChecker imposes one-way isolation between the two types of contexts. One the one hand, non-risky tasks that execute from code in the main context without involving risky code are always allowed by the security monitor to access objects in the sandbox context. As explained in the next section, they do not hold any capabilities and are not subject to restrictions imposed by ScriptChecker. On the other hand, risky tasks, namely those executing third-party code within sandbox contexts, are not allowed to reference objects in the main context by default, unless the referenced objects are explicitly whitelisted in their capabilities.

One-way isolation effectively prevents a third-party script from accidentally or maliciously tampering with the host script's objects and code. At the same time, the host script has the flexibility to specify objects to share with tasks running third-party functions. Thus, context separation protects the integrity of the JS environment the host script execution depends on.

V. Capability System

ScriptChecker outfits Chrome's security monitors with a new capability system to enforce policies against untrusted tasks. In the following, we begin with definitions of key elements in ScriptChecker's capability system, followed by descriptions of capability management and enforcement.

A. Definitions

Task. By default, Chrome assigns each task generated in the renderer process a unique sequence number as its identifier. We use T_n to denote the task with identifier n. T_n is defined as the *parent* task of T_m if the first function executed in T_m is chosen by T_n. At runtime, T_m can be created in one of the following ways: (a) T_n explicitly creates T_m by using an asynchronous API; (b) T_n designates a JS function as an event listener or as an IPC handler for processing a requested resource (e.g., a network request) and T_m is the task created to execute that function upon the corresponding event.

The purpose of introducing the parent-child relation is to capture runtime scenarios that one task can determine the code executed in another task, considering that a sly script may hide its malicious activities through asynchronous execution. Since we define the relation from the security perspective, it is not always consistent with the conventional notion of parent-child from the task generation perspective. For instance, in the second scenario above, it is the Chrome system task that creates T_m instead of T_n.

A task is defined as *risky* if it is either created by a non-risky task through one of ScriptChecker's asynchronous APIs, or a child task of a risky task. Hence, all descendants from a risky task are risky as well, even if these descendant tasks may execute JavaScript code from the host script. It is security critical to maintain the parent-child relation among risky tasks as it provides the basis for capability assignment and ensures that no untrusted code execution evades ScriptChecker's restriction.

Permissions. ScriptChecker's permission dictionary consists of three groups. Firstly, the system resource permission set, denoted by $\mathcal{P}_S$, govern accesses to system resources including the persistent storage (e.g., cookie and local storage),

network, inter-frame communication (e.g., postMessage) and user resources (e.g, sensors, clipboard and files). Secondly, the DOM resource permission set, denoted by $\mathcal{P}_D$, dictate whether a task can read or write DOM elements or register event listeners. Lastly, the cross-context reference permission set, denoted by $\mathcal{P}_J$, determine whether the task can reference JS functions and objects in the main context. For example, the `JS_WL:shareData` permission in Listing 1 is a permission in this category.

Capability and Provenance Label. A task's capability is essentially a set of permissions granted with respect to the aforementioned three categories. We use Ω_{T_n} to denote T_n's capability. It is expressed as $\Omega_{T_n} = \langle P_S, P_D, P_J \rangle$, where $P_S \subseteq \mathcal{P}_S$, $P_D \subseteq \mathcal{P}_D$, and $P_J \subseteq \mathcal{P}_J$ are the set of permissions in respective categories.

Since the tasks are within the renderer process, the kernel security monitor is unable to retrieve their capabilities to make access control decisions regarding IPC messages. To address this issue, ScriptChecker introduces the *provenance label* attached to IPC messages. A provenance label is essentially a copy of P_S in the sending task's capability. (Note that P_D and P_J are irrelevant to the kernel security monitor's access control.) The kernel security monitor uses the label in an IPC message to decide whether it is allowed to access a system resource such as cookies.

Note that all capabilities and provenance labels are created, stored and maintained by ScriptChecker. They are inaccessible to *any* JavaScript code, including the host scripts. As a result, no fake capability or label is used in policy enforcement and no risky task can transfer or delegate its capability to others.

B. Task Creation and Capability Assignment

Capabilities are assigned to *risky* tasks only. Tasks executing the host scripts do not have capabilities. According to our definition of risky tasks, a capability is created and assigned to T_n in the following three cases.

Case I. T_n is created by a host script using ScriptChecker's asynchronous API. For example, the host script in Listing 1 uses `asynCallCap()` to execute a third-party function in a newly created child task that is considered as risky. ScriptChecker generates the capability for it according to the parameter in the API during task creation.

Case II. T_n is created by a risky task using the standard JavaScript asynchronous APIs such as `setTimeout` and `setInterval`. ScriptChecker checks every child task generation. If a parent task is assigned with a capability, the task it creates is also assigned by ScriptChecker with the same capability by default. A risky task is also allowed to use ScriptChecker asynchronous API to assign a new capability to its child task. ScriptChecker ensures that the child task's capability is a *subset* of the parent's. This prevents a risky task from escalating its permissions.

Case III T_n is created by Chrome's system task to execute code which is labeled as risky using the metadata (e.g., `risky`) introduced by ScriptChecker. There are two subcases: script inclusion and listener execution. When a script with `risky` in its tag is downloaded and executed by Chrome in a new task, ScriptChecker assigns to the task the capability generated according to the associated `task_capability` attribute as exemplified in Listing 1. ScriptChecker modifies the listener registration procedure in Chrome. If a listener function is registered by a risky task T_n, ScriptChecker attaches it with the task's identifier n. When such a listener is executed, ScriptChecker assigns Ω_{T_n} to the generated listener task since it is the child task of T_n.

We highlight that ScriptChecker does *not* determine whether a listener task is risky or not based on its code source. In a confused deputy attack, the malicious third-party script can register a host script's function as an event listener for its malicious purpose. While not being able to tamper with the listener execution, the attacker control the data flow to or from it, which could result in undesirable consequences such as cookie deletion.

C. Capability Enforcement

A task's capability is applied at runtime by Chrome's security monitors to make access control decisions. The scope of permission checking comprises: cross-frame and in-frame DOM objects, system resources, cross-context JS objects, all of which are trapped to the local/kernel security monitor in Chrome by default. Namely, ScriptChecker does not introduce new trappings for enforcement but an additional policy in existing junctures of enforcement.

To support ScriptChecker, both the local and the kernel security monitors are retrofitted to make capability enforcement with respect to tasks. For the kernel security monitor, it just retrieves P_S from the provenance label attached to the IPC message and then evaluates whether the system resource request conforms with permissions stipulated therein.

We equip the local security monitor with a reference to the `current` task, i.e., the one accessing the resource. If the current task is bound with a capability, the local security monitor first checks permissions in the capability before applying the default JS access control policies such as SOP. Otherwise, the default policies are enforced only. Hence, ScriptChecker has no effect upon non-risky tasks. As compared with executions without ScriptChecker, a risky task has only one extra occasion for permission checking: cross-context references, since ScriptChecker places the third-party script code in the sandbox context instead of the main context.

VI. Asynchronous Execution of Risky Code

In this section, we elaborate how ScriptChecker allows a risky code invocation to begin in a newly generated task. We have categorized task creation scenarios to three cases in Section V-B. Since this section focuses on the details of asynchronous execution instead capability assignment, we regroup the scenarios into function call, script inclusion and listener execution.

A host script's call to a function in a third-party script is arguably the most common among them, as third-party scripts are often used as libraries providing utility functions. In script inclusion, the risky script is automatically executed after being loaded by Chrome. Listener functions are invoked through the callback mechanism which requires the listener function to be priorly registered. Note that listeners are called by Chrome

instead of the host script. In the following, we explain the details for each scenario and also deal with a side-effect due to asynchronously calling a risky function which itself makes asynchronous executions.

A. Restricted Risky Function Call

The key issue of converting a synchronous function call to an asynchronous one is to preserve the execution logic. Namely, the ensuing execution flow must be the same as the synchronous one. Thus, two new tasks are created: one for the callee function and the other for the caller's code following the function call. To minimize the incurred delay due to scheduling, the first task needs to be immediately executed after the function call.

ScriptChecker introduces two asynchronous APIs: `asynCallCap(f,c)` for third-party functions and `asynCallNoCap(f,tid)` for host functions. The former creates a child task to execute function `f` with a new capability generated based on `c` and returns the task's identity. The new task is given the top priority and inserted into the *head* of the task queue. Hence, it is the next one to execute after the completion of the current caller task. The second API creates a child task to execute function `f` without any capability. The new task is inserted next to task T_{tid} in the task queue so that it executes immediately after T_{tid}. Supposing that a host script function needs to invoke a third party function, it uses `asynCallCap` to asynchronously to run the callee and uses `asynCallNoCap` to asynchronously execute the rest of the host script following the third-party function.

We use an example below to illustrate how a synchronous call is converted into an asynchronous execution using the two APIs above. Listing 2 shows the existing way to synchronously call `foo` from a third-party script, while Listing 3 shows the corresponding asynchronous invocation with customized permission restriction. (Note that `foo` is referenced using `windowRisky` since the third-party script is loaded in the sandbox context as explained in Section IV.)

```
1function myfunc(params) {
2  // Call an untrusted function foo with parameters
3  var results = windowRisky.foo(params);
4  // Handle the return value for the untrusted
       function
5  rest_of_mycode(results);}
```

Listing 2. To synchronously call a third-party function `foo()`

```
1function asyn_myfunc(params) {
2  var results;
3  ...
4  // create a child task to run untrusted function
       foo
5  tid= asynCallCap(function () {
6    results = windowRisky.foo(params);
7  }, "No_Cookie_Access");
8  // create another child task to run the rest
9  asynCallNoCap(function() {rest_of_mycode(results)
       ;}, tid); }
```

Listing 3. To asynchronously call `foo()` in ScriptChecker

The combined use of the two APIs produces the effect illustrated in Figure 3. Let T_n be the running task executing `asyn_myfunc` and T_m be the supposed next-to-run task when T_n starts. The invocation of `asynCallCap` results in T_{tid} to head the task queue and the invocation of `asynCallNoCap` places T_{tid+1} next to T_{tid}. Essentially, the asynchronous execution involves three tasks T_n, T_{id} and $T_{id}+1$ which are executed *consecutively*. Since Chrome's task scheduling is non-preemptive and strictly follows the order in the queue, Listing 3 has exactly the same execution semantics as Listing 2.

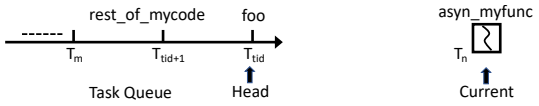


Fig. 3. Two child tasks are created after asyn_myfunc().

Due to the parent-child task relation introduced in Section V-A, all child tasks spinning off from T_{tid} and their descendants are constrained by ScriptChecker under the same capability of T_{tid}. It is unnecessary for the host script to deal with scenarios wherein `foo` might make its own asynchronous calls such as `setTimeout`, or register an event listener.

However, there is one special case in which the asynchronous function call has a side effect. A host script may call a third-party function to generate a network request and specifies the response function. For instance, as shown in Listing 4, the host calls `axios` to download `data.json` and designates its own function `h_handler` to process the network response. When `axios` is invoked asynchronously using the method described above, the task executing `h_handler` inherits the capability from its parent task, i.e, the one executing `axios.get`. This inherited restriction is actually undesired, because the task starts its execution from the host script function and is not controlled by `axios`.

```
axios.get("data.json").then(response=>{h_handler(
    response)});
```

Listing 4. Host-specified handler in third-party execution

The host script can have an alternative way to handle the network response to avoid the side effect. The method, as illustrated in Figure 4, is to convert the network response handler into an event listener registered directly by the host script. As a result, `h_handler` runs in the child task of the host task with full privileges and is thus not subject to restrictions.

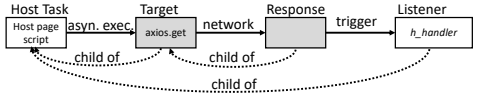


Fig. 4. Restricting asynchronous function `axios.get`. Grey boxes denote tasks with restricted privileges.

Listing 5 shows the details in the host script. It binds an event to a DOM element O set as inaccessible to any risky task, and registers its handler function as the event handler (Line 6). Next it asynchronously calls `axios.get` using `asynCallCap`. When the network operation is completed, the response function dispatches an event to trigger the handler execution. Note that the third-party function cannot directly dispatch the event by itself, due to denied accesses to DOM element O and the reference in the main context.

```
1  var results;
2  var ele_o = document.getElementById('O_id');
3  var event_name = "h_handler";
4  function h_handler() {...}
5  // register an event listener for DOM element 'O'
6  ele_o.addEventListener(event_name, h_handle);
7  asynCallCap(function(){
8    windowRisky.axios.get("data.json").then(response
         => {
9      results = response;
10     ele_o.dispatchEvent(new Event(event_name));
11   });}, capability);
12 ...
```

Listing 5. Run $axios.get$ in the asynchronous manner

B. Restricted Risky Script Inclusion

A third-party script can be included into the host frame by direct script inclusion through a `script` tag or dynamic inclusion using JS APIs such as *document.createElement*. In both cases, a network request is sent to the kernel process and the downloaded script is automatically executed without using a caller. It is therefore necessary to restrict risky execution incurred by script inclusion.

ScriptChecker's approach is to constrain the network request task which is the child task of either the parser in the case of direct inclusion or the host script task in the case of dynamic inclusion as illustrated in Figure 5. ScriptChecker introduces the `risky` and the `task_capability` attributes to mandate that the downloaded script be placed to a sandbox context and the network request task be restricted with the capability. If the script has no `async` attribute, it is executed within the request task; otherwise it is within the network response task which is a child of the network request task (as shown in Figure 5). In both cases, the script execution is restricted by the capability specified by the host script.

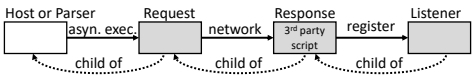


Fig. 5. Direct or dynamic inclusion of a script with `async` attribute. Grey boxes denote tasks with a capability.

An example of direct script inclusion has been shown in Listing 1 wherein the frame parser creates the network request task with the capability. For dynamic inclusion, the host script uses ScriptChecker's asynchronous APIs presented in Section VI-A.

C. Restricted Risky Listener Execution

Event listeners registered by a third-party script are risky as well. For instance, a faked `Modernizr` script collects user information when a submit button is clicked [7]. They deserve permission restriction as much as a third-party script does.

ScriptChecker copes with risky listener functions by hinging on the parent-child relation between tasks. According to Section V-A, the task executing a listener is the child of its registration task and inherits the capability from it (as illustrated in Figure 5). ScriptChecker modifies how listeners are invoked in order to run a risky listener in a separate task. It supports two types of listeners below.

Chrome Listener. When a risky task registers a listener function to a Chrome event, ScriptChecker records its task identifier n as the metadata stored together with the listener. When the event is triggered, the renderer process' handler task checks each listeners. If a listener carries with a task identifier n, the handler creates a new task to run it and assigns it with the capability Ω_{T_n}, i.e., its registration task's capability.

jQuery Listener. As a popular third-party library for DOM access and manipulation, jQuery [15] can bind a custom event with a DOM object without involving the browser. Scripts accessing the object can register their listeners to the event. When one task calls the event-triggering function, all registered listeners, including those registered by a third-party script, are executed in the *same* task. ScriptChecker handles this situation without modifying jQuery. The idea is to separates safe listeners and risky ones using two events which are always triggered together.

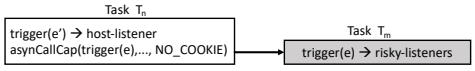


Fig. 6. Host listeners on e' and risky listeners on e

Figure 6 illustrates our approach. Suppose that jQuery event e is created for DOM object O. The host page script creates an additional DOM object O' and a jQuery custom event e' associated with O'. Host listeners are registered to event e' while risky listeners are registered to event e, which preserves the third-party script's view of the DOM. The host page script applies the techniques in Section VI-A and VI-B to disallow any third-party script to access the object O' along with other desired restrictions. This protects listener separation as no third party script can register its listener to e' due to prohibited access to O'.

When the host script needs to activate e, it first triggers e' to run its trusted listeners, and then uses `asynCallCap` to asynchronously trigger e in a child task (T_m in Figure 6) whose capability is properly set to restrict the listeners. Security is not breached even if e is triggered by a risky script, because the triggering task has already assigned with a proper capability and the restriction is automatically applied to the listeners' execution as well.

VII. Implementation

A. ScriptChecker Prototype on Chromium

We implement ScriptChecker on Chromium 67.0.3391.0. The prototype consists of around 2200 SLOC in C++. The source code is available for downloading at https://github.com/lwyeluo/ScriptChecker.

Code & Data Modification. As illustrated in Figure 7, ScriptChecker is a new component in Chromium's renderer process. The ScriptChecker component maintains capability for each risky task. Various Chromium functions in different modules are hooked for ScriptChecker. The involved core functions are listed in Table I below.

Specifically, ScriptChecker creates a capability for a newborn risky task by hooking the Task Generator and tracks task switches by hooking the Task Scheduler in order to acquire

	Major Functions	Module
Task Scheduler	*TaskAnnotator::RunTask*,	base
Task Generator	*IncomingTaskQueue::PostPendingTaskLockRequired*, *ThreadControllerImpl::DidQueueTask*,	base
Frame Parser	*ScriptLoader::ExecuteScriptBlock*, *LocalWindowProxy::Initialize*, *html_attribute_names.json5*,	blink
IPC Client	*InterfaceEndpointClient::Accept*, *RenderFrameImpl::OnMessageReceived*,	mojo, content
Security Monitor	*DocumentThreadableLoader::StartBlinkCORS*, *BindingSecurity::ShouldAllowAccessTo*, *TreeOrderedMap::Get*, *blink::Cookies*, *blink::SetCookie*,	blink, content
Events Handler	*EventTarget::SetAttributeEventListener*, *EventTarget::FireEventListener*, *blink::AddListenerToVector*,	blink
JS API	*DOMTimer::Install*, *window_timers.idl*, *window.idl*,	blink

TABLE I. CORE FUNCTIONS HOOKED TO INVOKE SCRIPTCHECKER.

the running task's identity. The Frame Parser is modified to recognize the `risky` attribute and requests the JS engine to create the sandbox context. For risky script execution (e.g., in `ScriptLoader::ExecuteScriptBlock`) or risky event listener triggering (e.g., in `EventTarget::FireEventListener`), the modified Chromium's renderer process runs the target JS code in restricted tasks. The JS engine is extended with two built-in objects (i.e. `windowMain` and `windowRisky`) and two functions (i.e. `asynCallCap` and `asynCallNoCap`).

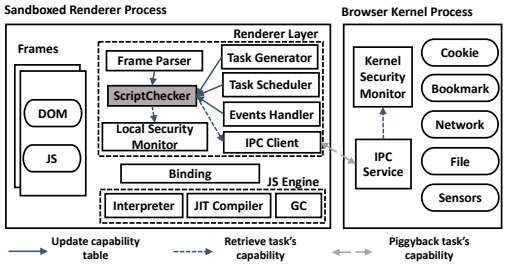


Fig. 7. The architecture for the implemented prototype.

We also modify the security monitors to support ScriptChecker. The IPC Client is hooked to retrieve the current task's capability, and piggybacks them as the provenance label to the `source frame identity` field in Chromium's IPC structures. We add capability checking logics within the local security monitor's access control functions, such as `ShouldAllowAccessTo` for cross-context JS and DOM objects accesses and `StartBlinkCORS` for network accesses. Functions including `TreeOrderedMap::Get` are also modified to check in-frame DOM objects accesses from a sandbox context.

Permission Expression. In our prototype, $\mathcal{P}_S$ consists of two permissions for system resource accesses: `Network_Access` and `Cookie_Access`. These two permissions representing the privilege of accessing all network or cookie resources respectively. However, a risky task granting these permissions cannot access network response or cookie from other origin/domain except the host frame's, since Chrome's default policy, e.g., SOP (and CSP), is still enforced in ScriptChecker. The permission setup can be extended for a wider coverage, such as sensors and inter-frame communications. It can also be enhanced with finer granularity, for example, to separate privileges for `GET` and `POST` requests, or allow network requesting for a allowlist of network servers.

Our prototype supports two methods to specify DOM access permissions. One is to use the permission macros: `No_DOM_Access` and `DOM_Access_Protective`. The former denies accesses to *all* DOM objects while the latter grants accesses to all DOM objects except those attached with the non-removable `task_sensitive` label. Essentially, the developers uses the label to group DOM objects into two classes: the sensitive one under protection and the other open to accesses. The other method uses the `DOM_Access(Type, List)` macro. All selected DOM objects in `List` are inaccessible and the remaining ones are accessible if `Type` indicates a blocklist; or the opposite if `Type` is an allowlist.

We have also implemented $\mathcal{P}_J$, i.e., the permissions for cross-context JS object references initiated from a sandbox context. The host script uses `JS_WL:List` to define $\mathcal{P}_J$, where `List` are JS objects in the main context. Since a third-party script typically does not access a large number of host script's objects, `JS_WL:List` produces a allowlist permissions. Namely, JS objects in the `List` are accessible to the sandbox context while others are not.

B. Security and Usability Evaluation

We run several experiments to demonstrate security and usability of ScriptChecker.

1) National Baseball Hall web page: We build a test page based on an archived National Baseball Hall web page[2] which was reported [22] to be attacked by a malicious analytics script (named `gbh.js`) from `googletagstorage.com`. We modify the test page to further include two malicious scripts: the fake Modernizr script that stole data from sensitive forms in `marveloptics.com` [38] by registering event listeners on button clicks for form submission, and a modified CryptoJS [13] script that steals the cookie by a direct access the cookie and by calling the test page's `accessCookie` function (forming as a confused deputy attack (CDA)). More details of the test page are in Figure 10(a) in Appendix A. All attacks succeed on the stock Chrome.

Experiment with ScriptChecker. We show how the web developer safeguards the page by using ScriptChecker. Table II below summarizes the capabilities assigned to each script.

3rd-party script description	Attacks	Capability
gbh.js for analytics	billing form (direct access)	`DOM_Access_Protective`, `Network_Access`, `Cookie_Access`
modernizr.js for network feature checking	billing form (event tracking)	`DOM_Access(Deny, {Tag:form,textarea,input})`, `Network_Access`
fake-crypto.js for SHA256 computation	cookie (direct access, CDA)	`NO_Cookie_Access`

TABLE II. TO RESTRICT THREE SCRIPTS IN THE TEST PAGE

The capability assignment is decided in accordance with the expected functionality of each script and the protection of the billing form against all of them. In total, we make 9 lines of changes including adding the sensitive tag to the billing form. Figure 10(b) in Appendix A presents more details.

[2]https://web.archive.org/web/20190205100422/https:/shop.baseballhall.org/

Chrome with ScriptChecker successfully blocks all attacks, including the CDA attack from `fake-crypto.js`. The test case illustrates that the web developer does not need to consider how the third-party script runs and what JavaScript API it calls. The developer can use ScriptChecker APIs to flexibly compose permissions for a third-party script execution.

2) Adaptive Capability Assignment: Our second experiment uses a web page with `jQuery` to show adaptability of ScriptChecker. In other words, the host can dynamically assign the capability to an upcoming execution of the third-party script, according to the security needs. The JS code of the test page is shown in Listing 6 below.

```
<script src="https://.../jquery.min.js" risky
    task_capabitity="DOM_Access_Protective" />
<script>
  asynCallCap(function () {
    windowRisky.$.ajax({url:"config.json",async:
        false}).responseJSON;
  }, "Network_Access", 0);
  // protect sensitive DOM after the user login
  asynCallCap(function () {
    windowRisky.$("#status").html();
  }, "DOM_Access_Protective", 0);
</script>
```

Listing 6. Dynamic permission setting for jQuery

The host script imposes two different permissions on `jQuery` in three scenarios. For script loading, it assigns `DOM_Access_Protective` permission because the library needs to access certain DOM elements e.g., `slot`, but is disallowed to access sensitive ones. The host page makes two invocations to `jQuery` in two tasks with different capabilities. Its first `asynCallCap` call allows `jQuery` to access the network only (i.e., `Network_Access`). Hence, the task cannot access any DOM objects or cookies. The second invocation of `asynCallCap` takes place after the user logs in. In order to protect sensitive user information, the second invocation assigns `DOM_Access_Protective` to restrict `jQuery`'s DOM access, without networking permission.

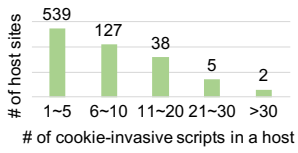
3) Malicious Script Benchmark: We create a simple webpage with a few DOM objects (including a password submission form) and a cookie. We test ScriptChecker against 1373 malicious scripts in a Github repository [14], using the most restrictive policy, namely to deny all accesses to DOM, cookie and network. The results show that 1021 scripts are blocked and 332 scripts end up with a crash (e.g., referencing non-exist APIs in Chrome). The remaining 20 scripts are not caught. Among them, twelve inhibit their attacks due to environmental errors and four use a popup which is out of the permission scope of our prototype[3]. Among the attacks, 1003 scripts inject arbitrary DOM/script using *document.write*, and 20 scripts create new DOM through *document.createElement*.

Many scripts use obfuscation techniques to evade code analysis. Some convert their code into `String` and decode them through `unescape` and `eval`. We also find 16 scripts using nested `eval` and 19 scripts registering event listeners. There are 8 scripts attempting to poison the environment by redefining the built-in objects, e.g., `Array.from` and `Array.prototype.indexOf`. Despite various ways of attack and obfuscation, ScriptChecker successfully blocks them according to the policy, by virtue of using task-capability based access control and sandbox context isolation.

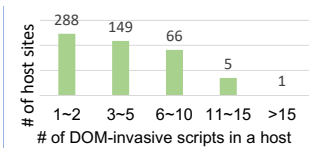
[3]We could not identify the reason manually for the remaining four because of their strong obfuscation.

4) Empirical Study on Popular Websites: We run an empirical study to assess how ScriptChecker may impact real-life websites. The first experiment is to understand how existing third-party scripts "invade" the hosts' sensitive resources. We modify Chrome to automatically label all third-party scripts as *risky* and any DOM object as `sensitive` if it contains a keyword from the dictionary {**passwd, password, secret, credential, card, privacy, private, security, input, form, button**}. ScriptChecker applies the same capability to the scripts so that accesses to sensitive DOM objects, network, and cookie are considered as policy violation. Nonetheless, to fully reveal the script behaviours, ScriptChecker is modified not to block illegal accesses, but to *record* them. The experiment with 4356 pages in Alexa top-1000 sites shows that 760 domains with either DOM or cookie invasive scripts. The details are reported below.

Cookie Access. There are 711 sites having third-party scripts accessing their cookies. Among them, 453 sites have the cookie-invasive scripts accessing the network, thus facing a higher risk of cookie leakage. Figure 8(a) plots the distribution of host sites according to the number of cookie-invasive scripts. The majority of them has less than five while the extreme one uses 40 such scripts. The ten sites using most cookie-invasive third-party scripts are reported on the left side of Table III. The invasive scripts in the experiment are from 551 different domains including those well-known ones under Google. The right side of Table III reports the domains owning most popular scripts with cookie accesses. As expected, analytical scripts make the most cookie accesses. Some websites share cookies with scripts from their sibling or parent domains.



(a) Usage of cookie-invasive third-party scripts



(b) Usage of DOM-invasive third-party scripts

Fig. 8. Histogram of Alexa top-1000 sites using risky third-party scripts (in logarithmic scale)

Host Site	# of Scripts
squarespace.com	40
td.com	33
trustpilot.com	27
uol.com.br	23
etrade.com	22
latimes.com	21
freee.co.jp	20
aliexpress.ru	18
linkedin.com	17
iqbroker.com	17

Script Domain	# of Hosts
googletagmanager	451
doubleclick.net	116
google-analytics.com	102
cookielaw.org	70
googletagservices.com	62
go-mpulse.net	41
googlesyndication.com	41
licdn.com	41
facebook.net	37
baidu.com	32

TABLE III. WEBSITES USING MOST COOKIE-INVASIVE THIRD-PARTY SCRIPTS AND PROVIDERS OF MOST POPULAR COOKIE-INVASIVE SCRIPTS.

Sensitive DOM Access. There are 509 sites with third-party scripts accessing their DOM objects containing keywords in

our dictionary. Figure 8(b) plots the distribution of host sites according to the number of third-party scripts accessing to sensitive DOM objects (in our definition). The majority of the websites have one or two such scripts while the extreme case has seventeen. These third-party scripts are from 538 different web domains. As shown in Figure 9 below, DOM objects with "input" and "form" keywords are read much more often than others, possibly because they are more used user-activity analysis. The figure also shows many scripts accesses multiple sensitive DOM objects. Accesses to objects with "password" and "secret" may require more prudent examination. Note that only the website owners can determine whether the accesses are desirable or not.

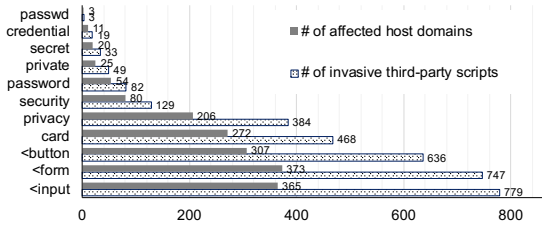


Fig. 9. Distribution of DOM accesses by third-party scripts and affected host domains

Our second experiment is to assess how ScriptChecker's enforcement breaks third-party script executions. We use the same policy and browser setup in the first experiments, except that ScriptChecker blocks all illegal accesses. For each domain having cookie or DOM invasive scripts, we randomly choose one page for testing. In total, the dataset consists of 760 pages. The results show that 676 pages are rendered normally (i.e., the `document.readyState` being set as complete), 66 pages have their loadings timeout and 18 pages result in browser crashes. Among the crashed pages, four are due to the Chromium version in use and other crashes are potentially due to bugs in our modifications. The data shows that websites seem robust against third-party script execution failure. The specific consequence varies from case to case, e.g., terminating the script code snippet interpreted by JS engine due to referencing null pointer. Since the host script's execution is in separate context from third-party script execution, termination of an offending script execution does not disrupt the rendering of the host pages.

In short, our experiments show that it is not uncommon to find cookie or DOM invasive third-party scripts among popular sites. Although we do not have the knowledge to determine whether those accesses are justifiable, the current practice is clearly against the least-of-privilege principle and induces security risks. ScriptChecker can help hosts to flexibly and selectively grant or deny accesses to sensitive resources.

C. Webpage Development & ScriptChecker

Like dealing with supply chain attacks, security problems due to third-party scripts cannot be fully resolved without efforts from the webpage developers. Legacy websites are oblivious to the deployment of ScriptChecker in Chrome and are rendered normally without any setback. We show below how a web developer can benefit from ScriptChecker.

Grant or Not-Grant. With ScriptChecker, a developer no longer treats a third-party script in the all-or-nothing fashion: either not using at all or using with the same privileges as host scripts. She can follow the least-privilege principle and grant permissions discreetly. Nonetheless, there are two thorny problems: (i) how to know the permissions demanded by a third-party script and (ii) how to assign permissions with no or little functionality disruption. Note the developer also face similar questions when using the existing JS based schemes. ScriptChecker helps developers to better answer them.

A developer can test her webpages with third-party scripts using the log-only variant of ScriptChecker used in our empirical study. The testing outcome reveals the whole set of permissions demanded by the scripts. The developer can make proper decisions for each prospective execution scenario. In general, we recommend that developers use an allowlist approach to explicitly grant system resource permissions in $\mathcal{P}_S$ and cross-context reference permissions in $\mathcal{P}_J$, since those accesses often have direct and well-defined justifications. For DOM access permissions in $\mathcal{P}_D$, we recommend the blocklist approach (i.e., `DOM_Access_Protective`) to safeguard sensitive DOM objects. Security-critical webpages may need more fine-grained considerations.

Thanks to its fine-granularity and flexibility, ScriptChecker is more advantageous to defuse the conflict between security and functionality as compared with exiting schemes. It allows customised permission combos for different executions of the script including loading, function invocation. It also allows object-level permissions with different operation parameters. Hence, it is easier to accommodate the script's execution needs. Note that our prototype can be extended to support more permissions, including endorsement.

Host Script Modifications. The developer needs to make a few changes to her current JS code to use ScriptChecker. For a third-party script, she needs to make one line change for script loading to add the risky label and the loading-time capability. For each function call made by her code to the script, she needs to change three places: to create the capability for the callee execution, to asynchronously call the script, and to resume her own code execution. Depending on the permissions in use, she may also add the `tasks_sensitive` label to DOM objects deserving protection. In short, the amount of required efforts is limited and locating the code for change is easy.

Limitations of ScriptChecker. Access control is arguably the last line of defence against resource misuses. It does not neutralise malware attacks by itself. Like its counterparts in file systems and databases, ScriptChecker has inherent limitations. After a task is granted with permissions, ScriptChecker cannot discern whether the granted access is for good or bad purposes. Moreover, it follows the DAC model. The host's policies may have loopholes leading to data leakage. Section IX-B elaborates how ScriptChecker can be reinforced by information flow checking.

VIII. Performance Evaluation

Our performance evaluation experiments run on a desktop with 16 Intel Core i7-10700K 2.9 GHz processor and 64 GB DRAM installed with Ubuntu 16.04 x64 with Linux kernel 4.4.0. We first measure the main component overheads of ScriptChecker and then evaluate its performance impact to

DOM accesses followed by an overall assessment on third-party script loading and usage. In the end, we measure the performance impact on JavaScript performance on Chrome.

A. Component Overheads

As compared with third-party script execution without restriction, the main cost of ScriptChecker is contributed by asynchronous function execution, sandbox context creation and capability-based checking. We build a simple webpage with a home-made third-party script that accesses the host's DOM, cookie and network interface. The purpose is to measure the CPU time taken by individual steps in ScriptChecker. Note that the sum of them does not represent the total cost of ScriptChecker, as it depends on behaviors of both the host and the third-party scripts.

Asynchronous Execution. We run experiments to evaluate the overhead due to converting synchronous function calls to asynchronous execution. To avoid measurement noise due to JavaScript engine's code optimization for hot JS code, we measure the time usage at Chrome's binding layer which is programmed using C++. The binding layer intercepts the API call from the JS engine and invokes the API implementation code in Chrome's Blink.

According to our results, the task creation using `setTimeout` costs about 162 µs in average while task creation using `asynCallCap` takes about 197 µs due to capability generation and assignment. We also measure costs due to task scheduling as the gap between the binding layer finishing the `asynCallCap` call and the new task beginning to run the DOM timer. The experiments show that it takes 21 µs in average.

Hence, the cost of asynchronous execution in Listing 1 comprises two task creation and two task scheduling, which amounts to approximately 0.44 ms. Note that the two child tasks are inserted to the head of the task queue, which avoids the waiting time for other tasks in the queue to complete.

Sandbox Context Creation. We also build a page to measure the time cost for initializing the sandbox context. Chrome creates a main context for each frame when the frame begins to loading. Since the creation of a sandbox context always follows the main context creation, ScriptChecker copies those objects instead of creating new one. As a result, creating a sandbox context requires only 2.6 ms, less than the main context creation time (4.2 ms).

In ScriptChecker, the sandbox context creation is the only step needed for setup. Hence, we compare it with the setup time with other schemes in Table IV. ScriptChecker needs much less time to setup than AdSentry [29], TreeHouse [34] and Jate [51], though it probably takes more than ConScript [42] whose policy file loading time is not reported.

Cross-Context Permission Checking. We also use the aforementioned web page to measure the overhead of cross-context permission checking. The page allows the risky JS code in the sandbox context to access a selected object in main context. In the experiment, we measure the execution time of the security monitor *BindingSecurity::ShouldAllowAccessTo*. An access from the main context to the sandbox context which incurs no checking costs 0.9 µs, while an access from the other direction costs about 2.6 µs in average.

Schemes	Setup Overhead
AdSentry [29]	**31 ms** (9×10^7 cycles), script loading to shadow JS engine
TreeHouse [34]	**350 ms** (9×10^8 cycles), script loading to Web Workers
Jate [51]	**58 ms**, script rewriting growing drastically with dynamic code (No CPU frequency reported)
ConScript [42]	**2 µs** (7.2×10^4 cycles) for environment creation plus time for policy loading
ScriptChecker	**2.6 ms** (7.5×10^6 cycles) for sandbox context creation

TABLE IV. COMPARISON OF THE SETUP COST. THE NUMBER OF CPU CYCLES IS CALCULATED BASED ON THE RESPECTIVE CPU FREQUENCY.

Resource Access Permission Checking. We also evaluate the overheads for DOM, cookie and network accesses after script loading. The time is measured in Chrome's binding layer to exclude the time taken by JS engine. Table V reports the results. Note that the network access incurs the highest CPU time for checking due to two child tasks being generated and creation of provenance labels. However, it is the least significant since the access itself costs longer time than others.

	DOM	Cookie	Network
Baseline	2.1	346	6690
With ScriptChecker	2.3	387	6818
Overhead	0.2 (9%)	41 (12%)	128 (2%)

TABLE V. PERMISSION CHECKING IN RESOURCE ACCESSES (IN µS).

In short, the overall overhead incurred by ScriptChecker on a webpage can roughly be expressed as $2.6c_s + 0.44c_i + 0.128c_n$ milliseconds, where c_s, c_i, c_n refer to the number of third-party scripts with `risky` attributes loaded in the page, the number of asynchronous executions, and the number of network accesses during all those invocations, respectively. Note that the overhead due to DOM accesses is negligible since it is four orders of magnitude smaller than the loading overhead. The formula shows that the overhead for a specific website is more likely to be dominated by the number of third-party scripts, unless there exist intensive asynchronous invocations. As compared with the time delay of average webpage loading, the added delay appears unnoticeable to human users.

B. Macro-benchmark Experiments

Performance of Restricted Third-party Scripts. The overall impact on third-party script depends not only on ScriptChecker, but also on the script's own behaviors. Since there is no common benchmarks, we choose top 10 graphic libraries [17] to represent heavy-duty scripts which make intensive DOM accesses and 5 utility libraries [17] to represent light-duty ones. We build the webpage by cloning the JS libraries' website and measure the time interval between the start of page loading and the moment when the library functionality is completed (e.g., when the draw is finished for graphic libraries), with and without ScriptChecker. To avoid the networking impact on the results, all scripts are stored locally. For each library, ScriptChecker blocks its access to a private form in the web page. Table VI reports the experiment results. The average performance tolls upon the graphic libraries and the utility libraries are 3% and 9%, respectively.

Graphic I	chartjs	highcharts	particlesjs	raphael	d3
Baseline	1218.37	415.87	237.69	248.69	625.04
Script-Checker	1223.17 (4.8)	427.57 (11.7)	248.19 (10.5)	258.95 (10.26)	670.68 (45.64)
Graphic II	threejs	mathjax	amcharts	supersized	googlecharts
Baseline	3211.22	3081.13	5922.22	188.65	2118.10
Script-Checker	3227.09 (15.87)	3130.03 (48.9)	5996.25 (74.03)	200.30 (11.65)	2154.44 (36.34)
Utility	jquery	lodash	modernizr	moment	underscore
Baseline	102.88	85.28	200.40	68.47	55.86
Script-Checker	111.64 (8.76)	93.75 (8.47)	206.55 (6.15)	75.36 (6.89)	63.62 (7.76)

TABLE VI. LATENCY (IN MS) ON LOADING WEB PAGES AND INVOKING THE FUNCTIONALITIES OF JS LIBRARIES REPORTED IN [17].

In these experiments, we do not visually notice the performance difference incurred by ScriptChecker. The experiment results show that the performance impact varies from case to case. As explained in our micro-benchmark experiments, the overhead is by and large decided by the number of scripts to load. The `amcharts` test suits has the highest overhead because its loads 5 scripts and registers 1757 listeners. Hence, webpages with more third-party scripts and frequent invocations of light-weight third-party script tasks suffer a higher relative overhead.

Performance of Host Scripts. Three benchmark suites are used to evaluate the speed and smoothness of executing the host's JS code which does not involve any third-party scripts. We run the Kraken [5] benchmark with and without ScriptChecker. Kraken uses a myriad of test cases, including audio processing, image filtering, JSON parsing and cryptographic routines. For each test, the average time usage is reported in Table VII. The total overhead is about 0.1%.

	ai	audio	imaging	json	stanford	Total
Baseline	249	398	426	783	1409	3266
ScriptChecker	249	397	428	782	1413	3269

TABLE VII. JAVASCRIPT PERFORMANCE FOR KRAKEN (IN MS).

The JetStream2 [8] benchmark suite scores 31.88 and 31.75 in the stock Chromium and Chromium with ScriptChecker, respectively. It amounts to roughly 0.4% drop. To assess whether the host's DOM access is affected, we choose the Dromaeo DOM Benchmark [4]. The experiment reports 0.304% drop in average.

These experiments show that ScriptChecker inflicts negligible collateral performance loss on the host scripts not using third-party scripts. Because these host tasks do not have a capability, ScriptChecker neither converts them into asynchronous execution nor checks their permissions at runtime. The reported loss is mainly attributed to the checking whether a task carries a capability.

IX. COMPARISON WITH EXISTING SCHEMES

A. ScriptChecker vs. Third-party Script Restriction Schemes

Table VIII summarizes the comparisons between ScriptChecker and existing schemes from three perspectives: performance, usability and security.

Performance. Reference restrictions schemes take a heavier performance toll than other approaches, mainly due to the overhead to cross the restriction boundary. Attributed to messaging passing, AdSentry reports 590 times overhead in one DOM access[4] and Treehouse incurs about 15 times overhead to DOMTRIS [3]. Although Jate [51] uses proxy objects instead of messages, it still incurs 219% overhead in loading jQuery because of dynamic script-rewriting and object creation. The runtime overhead of function filtering schemes (e.g., ConScript [42] and WebJail [52]) depends on the advice functions executions. It is especially efficient when a JS function makes a batch of resource accesses and can be arbitrated as a whole. Its main performance drawbacks are code expansion and costs of protection over the advice functions.

Usability. Because all existing schemes work at the JS layer, they place a heavy burden on web developers who must express the desired policies into JS functions, i.e., advise functions in function filtering schemes and the call-back functions in resource restriction schemes. These functions are additional security logics which require security expertises and prudence to develop. In contrast, with ScriptChecker in the browsers, developers only need to specify the resources to protect and leave the enforcement details to the browser. Another remarkable advantage of ScriptChecker is its adaptability to change policies at runtime as evidenced by the JQuery restriction case in Section VII-B. None of the existing scheme supports dynamic policy changes. In terms of deployability, ScriptChecker involves non-negligible changes on Chrome while Treehouse [34] does not require any client end support.

Security. ScriptChecker provides the strongest security. Existing schemes are susceptible to the confused deputy attack, including collusion attacks where a script gets another script's privilege by calling its function. Moreover, existing schemes requires a manual effort to cover the entire attack surface at the JS layer, which is error prone and must be kept consistent with the latest JavaScript engine. In contrast, ScriptChecker's assurance is independent of the JS layer attack surface.

B. ScriptChecker v.s. Information Flow Checking

Information Flow Checking (IFC) schemes [39], [28], [45], [27] track and regulate runtime data flows to stall sensitive data leakage to unauthorized parties. These schemes enforce a system-wide policy governing all DOM objects and scripts including third-party ones. Its ability of restricting third-party scripts depends on whether it is fine-grained enough to differentiate them from the host scripts. Since third-party scripts are often co-located with host scripts in the same frame, fine-grained IFC incurs a remarkably high overhead. Moreover, IFC imposes a rigid and uniform policy and hence is ill-suited to the divergent use of third-party scripts. As explained in Section II, the least-privilege principle requires that third-party scripts used in one webpage may need different permissions for their respective purposes and the same for different executions of the same script. It is hard to attain an elastic IFC for dynamic policy changes.

[4]AdSentry reports only 4.08% overhead in confining advertisement scripts because, according to its authors, the the network cost is "typically much more significant than local computation time in web browsers" [29].

Main Approach	Schemes	Runtime Overhead		Usability			Security	
		DOM Access	Page Loading	Developer's work	Policy Enforcer	Adaptable Policy	JS Attack Surface Coverage	Confused Deputy Attack & Collusion
Resource restriction	AdSentry [29]	590x	4.08% on Ads restriction	write call-back functions	Host JS code	No support	i) Manually done by developers; ii) error prone; iii) Need to update with JS;	Vulnerable
	TreeHouse [34]	N/A	15x on DOMTRIS [3]					
	Jate [51]	11.97%	19.5% on Ads restriction; 219% on JQuery loading;					
Function filtering	ConScript [42]	15%	N/A	write advice functions				
	WebJail [52]	N/A	**42%** on self-made pages					
Task capability	**ScriptChecker**	**9%**	**9%** on utility libraries [17] **8%** on JQuery **7.67%** on DOMTRIS [3]	replace function call interface	Browser	Easy	Orthogonal	Secure against

TABLE VIII. COMPARISON WITH EXISTING SCHEMES RESTRICTING RISKY SCRIPS.

In contrast to IFC, ScriptChecker is a machinery enforcing the host-declared policies for third-party scripts. The strength or correctness of the policies is beyond its scope. For instance, the host page copies its cookie to a DOM object and then invokes a third-party script without cookie access permission. Although ScriptChecker prevents the script's execution from accessing the cookie, it cannot stop the leakage from the aforementioned DOM object which is unprotected.

The methodologies behind ScriptChecker and IFC can be combined together to reinforce each other. On the one hand, an IFC scheme using tasks as subjects do not update data labels along the instruction flow. It greatly reduces tracking occasions and simplifies rules so that a coarse-grained IFC scheme can meet the security demands. On the other hand, ScriptChecker can be augmented by a Mandatory Access Control (MAC) and by a data tainting mechanism to trace sensitive data flows so that the aforementioned leakage can be caught. It releases the developer's from the onus of defining a watertight policy.

X. DISCUSSIONS

A. Confining Code with Untrusted Inputs

Besides risky scripts, the host frame's own code receiving untrusted inputs poses threats as well, especially when the untrusted inputs are converted to code using `eval`. Existing defenses sanitize inputs based on the known attack patterns and vulnerabilities [26], [32], [48], [11]. These approaches may not be able to detect zero-day attacks.

ScriptChecker can be applied in tandem with input sanitization to restrict those risky executions. Even if a malicious input is missed by the sanitizer, its execution is still restricted by the assigned capability which can stall the attack or mitigate the damage. ScriptChecker hence provides the last line of defense while existing techniques at the JS layer form the front line defense. Furthermore, the existing defense can benefit from ScriptChecker as it provisions more knowledge to examine untrusted inputs. For instance, if a message is sent from a restricted task, the receiver frame may need to apply more rigorous checking with the relevant contextual data.

For instance, a special case of untrusted input is the so called permission re-delegation attack [30]. A malicious third-party function may "invoke" its deputy code by delivering messages using `postMessage`. The task that processes the message does not inherit the sender's capability, and its execution could be influenced by the adversary due to its own vulnerability. To counter the attack, the message handler can spin off a new task with downgraded privilege to process the incoming message. The new task can be deprived of accessing any sensitive DOM or critical JS objects.

B. Third-party Script Modification

As described in Section IV, ScriptChecker places a third-party script into a sandbox context to counter environment poisoning attacks. The relocation the script to use `windowMain` of the sandbox context to directly reference JS objects created by the host script. There are two approaches to automatically handling the change. One approach is loading-time script rewriting. The third-party script is screened upon loading so that direct references to JS objects of the host script are replaced by an indirect reference through the `windowMain` object. The other approach is to modify the JS engine to detect such scenarios at runtime. When executing a script in the sandbox context, the JS engine always try to locate the JS object in references from the sandbox context by default. If the target object is not found, it searches the corresponding main context for the object under the same name.

The first approach is more lightweight and incurs less overall runtime overhead since the script is only screened once. However, it may not able to replace all the references. The second approach is more complete and yet incurs heavier overhead especially when the script is executed frequently. A hybrid approach may strike a good balance between efficiency and completeness.

C. JavaScript Access Control: Code vs. Task

Access control against JavaScript code is one of the fundamental security responsibilities the browser undertakes. By default, Chrome's local/kernel security monitor enforces SOP [16] for all JS scripts, regardless whether they are from the host or a third-party, though various frame based policies are proposed in the literature, e.g., COWL [47].

From the access control perspective, the security monitor considers the JS code as the subject and determines whether its origin and/or frame attributes conform to the prevailing policy. Unfortunately, it is known to be susceptible to the confused deputy attacks in the same way to attack existing third-party script restriction schemes as shown in Section II-C. Malicious code can evade the access control restrictions by calling legitimate JS code. Several attacks have been discovered including cross-site request forgery [49], clickjacking [1] and permission re-delegation [30], all of which are embodiments of the confused deputy attack in different settings.

ScriptChecker can be easily extended to harden SOP and other frame-based policies against the confused deputy attack, in that they are all enforced by the local/kernel security monitor. To strengthen SOP and frame-based policies, the capability scheme of ScriptChecker as described in Section V needs to be extended to attach frame related metadata for each task. At the moment of SOP checking, the security monitor consults the metadata of the requesting task, instead of the origin of the requesting JS code. We remark that the task oriented access control resembles process oriented permission checking in modern operating systems.

D. Deployment Considerations

Backward Compatibility. With ScriptChecker, web applications depend on the browser to restrict their third-party scripts. If a web application does not find ScriptChecker APIs in the browser, it then falls back to a JS layer scheme to confine third-party scripts. Note that ScriptChecker and JS layer schemes can co-exist in the same browser without conflicting each other. We believe that ScriptChecker can be deployed in an evolutionary fashion. Since ScriptChecker is fully transparent to legacy webpages, web browsers can take the lead to deploy it without breaking existing webpages.

Deployment on Other Browsers. The design principle of ScriptChecker hinges on two fundamental functions of a browser: task scheduling and security monitors, which are components in other mainstream browsers. The former is mandated by the HTML standard. Browsers should use an even-loop to coordinate various executions including scripts. Moreover, mainstream browsers have their respective security monitors enforcing SOP and CSP. Hence, ScriptChecker can be built in browsers other than Chrome, although the implementations details, such as APIs for task generation are different from our prototype.

XI. Summary

We propose and implement ScriptChecker, a browser-based capability system enforcing policies defined by the host script against risky third-party script executions. Unlike in the existing work, web developers can focus on resources to protect, instead of covering all attack vectors. By combining tasks with capabilities, ScriptChecker has the built-in resistance against the confused deputy attacks. Our experimental results show that the security and usability advantages are gained at the cost of modest performance loss.

Acknowledgement

We are grateful to anonymous reviewers for their constructive suggestions and comments.

References

[1] "The confused deputy rides again," http://waterken.sourceforge.net/clickjacking/, 2008.

[2] "Adsafe," https://www.crockford.com/adsafe/index.html, 2011.

[3] "Test tetris in treehouse," https://github.com/lawnsea/TreeHouse/blob/master/demos/domtris-pageload.html, 2012.

[4] "DOM core tests in dromaeo," http://dromaeo.com/?dom, 2013.

[5] "The kraken benchmark for javascript," https://wiki.mozilla.org/Kraken, 2013.

[6] "Play safely in sandboxed iframes," https://www.html5rocks.com/en/tutorials/security/sandboxed-iframes/, 2013.

[7] "Inside the magecart breach of british airways: How 22 lines of code claimed 380,000 victims," https://www.riskiq.com/blog/external-threat-management/magecart-british-airways-breach/, 2018.

[8] "Introducing the jetstream 2 benchmark suite," https://webkit.org/blog/8685/introducing-the-jetstream-2-benchmark-suite/, 2019.

[9] "Chromium's multi-process architecture," https://www.chromium.org/developers/design-documents/multi-process-architecture, 2020.

[10] "Chromium's task," https://chromium.googlesource.com/chromium/src/+/lkgr/docs/threading_and_tasks.md, 2020.

[11] "Content security policy (CSP)," https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP, 2020.

[12] "Cross-origin resource sharing (CORS)," https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS, 2020.

[13] "crypto-js," https://github.com/brix/crypto-js, 2020.

[14] "Geeksonsecurity," https://github.com/geeksonsecurity/js-malicious-dataset, 2020.

[15] "jquery," https://jquery.com/, 2020.

[16] "Same-origin policy (SOP)," https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, 2020.

[17] "Top javascript graphics detected by scanning websites," https://hexometer.com/most-popular-tech-stacks/JavaScript+Graphics, 2020.

[18] "Userreport," https://www.userreport.com, 2020.

[19] "Yahoo partner ads (YPA)," https://developer.yahoo.com/ypa/, 2020.

[20] "Featurepolicy," https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Feature-Policy, 2021.

[21] "Using web workers," https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers, 2021.

[22] L. Abrams, "National baseball hall of fame hit by payment card stealing attack," https://www.bleepingcomputer.com/news/security/national-baseball-hall-of-fame-hit-by-payment-card-stealing-attack/, 2019.

[23] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "Jsand: complete client-side sandboxing of third-party javascript without browser modifications," in *Proceedings of the 2012 Annual Computer Security Applications Conference (ACSAC)*. ACM, 2012, pp. 1–10.

[24] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "The devil is in the constants: Bypassing defenses in browser JIT engines." in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[25] A. Barth, C. Jackson, C. Reis, T. Team *et al.*, "The security architecture of the chromium browser," *Technical report*, 2008.

[26] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side xss filters," in *Proceedings of the 2010 international conference on World Wide Web (WWW)*, 2010, pp. 91–100.

[27] 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 *Proceedings of Network and Distributed System Security (NDSS)*, 2015.

[28] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, "Information flow control in webkit's javascript bytecode," in *Proceedings of International Conference on Principles of Security and Trust (POST)*, 2014.

[29] X. Dong, M. Tran, Z. Liang, and X. Jiang, "Adsentry: comprehensive and flexible confinement of javascript-based advertisements," in *Proceedings of the 2011 Annual Computer Security Applications Conference (ACSAC)*. ACM, 2011, pp. 297–306.

[30] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses." in *Proceedings of USENIX Security Symposium*, vol. 30, 2011, p. 88.

[31] N. Hardy, "The confused deputy:(or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[32] M. Heiderich, C. Späth, and J. Schwenk, "Dompurify: Client-side protection against xss and markup injection," in *Proceedings of European Symposium on Research in Computer Security (ESORICS)*. Springer, 2017, pp. 116–134.

[33] M. Ikram, R. Masood, G. Tyson, M. A. Kaafar, N. Loizon, and R. Ensafi, "The chain of implicit trust: An analysis of the web third-party resources loading," in *Proceedings of the 2019 international conference on World Wide Web (WWW)*. ACM, 2019, pp. 2851–2857.

[34] L. Ingram and M. Walfish, "Treehouse: Javascript sandboxes to help web developers help themselves," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference (ATC)*. USENIX Association, 2012, pp. 13–13.

[35] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang, "The web/local boundary is fuzzy: A security study of chrome's process-based sandboxing," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 791–804.

[36] D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey, "Security challenges in an increasingly tangled web," in *Proceedings of the 2017 International Conference on World Wide Web (WWW)*. International World Wide Web Conferences Steering Committee, 2017, pp. 677–684.

[37] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[38] J. Li, "Reversing js malware from marveloptics.com," https://blog.jse.li/posts/marveloptics-malware/, 2018.

[39] S. Maffeis, J. C. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2010.

[40] G. Maisuradze, M. Backes, and C. Rossow, "What cannot be read, cannot be leveraged? revisiting assumptions of jit-rop defenses," in *Proceedings of USENIX Security Symposium*, 2016, pp. 139–156.

[41] ——, "Dachshund: digging for and securing against (non-) blinded constants in JIT code," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[42] L. A. Meyerovich and B. Livshits, "Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2010, pp. 481–496.

[43] J. Mickens, "Pivot: Fast, synchronous mashup isolation using generator chains," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014, pp. 261–275.

[44] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote javascript inclusions," in *Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 736–747.

[45] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer, "Information flow control for event handling and the dom in web browsers," in *Proceedings of Computer Security Foundation Workshop (CSF)*, 2015.

[46] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis, "Revisiting browser security in the modern era: New data-only attacks and defenses," in *Proceedings of IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 366–381.

[47] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazieres, "Protecting users by confining javascript with COWL," in *Proceedings of 2014 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 131–146.

[48] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against DOM-based cross-site scripting," in *Proceedings of USENIX Security Symposium*, 2014, pp. 655–670.

[49] A. Sudhodanan, R. Carbone, L. Compagna, N. Dolgin, A. Armando, and U. Morelli, "Large-scale analysis & detection of authentication cross-site request forgeries," in *Proceedings of IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 350–365.

[50] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan, "Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements." in *Proceedings of USENIX Security Symposium*, 2010, pp. 371–388.

[51] T. Tran, R. Pelizzi, and R. Sekar, "Jate: Transparent and efficient javascript confinement," in *Proceedings of the 2015 Annual Computer Security Applications Conference (ACSAC)*. ACM, 2015, pp. 151–160.

[52] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen, "Webjail: least-privilege integration of third-party components in web mashups," in *Proceedings of the 2011 Annual Computer Security Applications Conference (ACSAC)*, 2011, pp. 307–316.

[53] Y. Zhou and D. Evans, "Understanding and monitoring embedded web scripts," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 850–865.

APPENDIX A

The case study shown in Section VII-B1 examines the effectiveness and adaptability of ScriptChecker. The original web page and the modified one with ScriptChecker are depicted in Figure 10(a) and Figure 10(b), respectively.

```
<!DOCTYPE html>
<head>
...
<script>

    (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;
    i[r]=i[r]||function(){(i[r].q=i[r].q||[]).push(arguments)},
    i[r].l=1*new Date();a=s.createElement(o),
    m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;
    m.parentNode.insertBefore(a,m)
    })(window,document,'script', https://.../.../gbh.js' ,'gadw');

</script>
...
<script type="text/javascript" src="https://.../.../modernizr.js">

</script>

<script type="text/javascript" src="fake-crypto.js">

</script>                                                    Three untrusted scripts
...
</head>
<body>
...
<form action="" method="get" id="co-billing-form">
    <p>Email:
        <input type="text" name="email" value="xxx@xxx.com"/>
    </p>
    <p>Password:
        <input type="text" name="pwd" value="123456"/>
        </p>
    <input type="submit" value="Submit" />                   DOM elements
</form>
<textarea rows="10" cols="30">A textarea</textarea>
...
<script type="text/javascript">
    function accessCookie() { return document.cookie; }
</script>

<script type="text/javascript">
    // test feature API using Modernizr                      Invoke Modernizr
    supportSocket = Modernizr.websockets;

    // the business logic according the result of feature testing
    runWithFeature(supportSocket);
</script>


<script type="text/javascript">
    // randomString generates string with given size
    data = randomString(64*1024);
    var hashes;                                              Invoke CryptoJS
    hashes = CryptoJS.SHA256(data).toString(data);

    Handle(hashes); // handle the hash results
</script>
...

</body>
</html>
```

(a) The original web page.

```
<!DOCTYPE html>
<head>
...
<script
  risky task_capability=
"DOM_Access_Protective;Network_Access;Cookie_Access;">
    (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;
    i[r]=i[r]||function(){(i[r].q=i[r].q||[]).push(arguments)},
    i[r].l=1*new Date();a=s.createElement(o),
    m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;
    m.parentNode.insertBefore(a,m)
    })(window,document,'script','https://.../.../gbh.js','gadw');
</script>
...
<script type="text/javascript" src="https://.../.../modernizr.js"
    risky task_capability=
    "Network_Access;DOM_Access(Deny,{Tag:form,textarea,input});">
</script>
<script type="text/javascript" src="fake-crypto.js"
    risky task_capability="No_Cookie_Access;">
</script>
...
</head>
<body>
...
<form action="" method="get" id="co-billing-form" task_sensitive>
    <p>Email:
        <input type="text" name="email" value="xxx@xxx.com"/>
    </p>
    <p>Password:
        <input type="text" name="pwd" value="123456"/>
        </p>
    <input type="submit" value="Submit" />
</form>
<textarea rows="10" cols="30">A textarea</textarea>
...
<script type="text/javascript">
    function accessCookie() { return document.cookie; }
</script>

<script type="text/javascript">
    var supportSocket;
    var capability =
      "Network_Access;DOM_Access(Deny,{Tag:form,textarea,input});";
    tid = asynCallCap(function () {
        supportSocket = windowRisky[1].Modernizr.websockets;
    }, capability);
    asynCallNoCap(function() { runWithFeature(supportSocket); }, tid);
</script>

<script type="text/javascript">
    data = randomString(64*1024);
    var hashes, capability = "No_Cookie_Access;";
    tid = asynCallCap(function () {
        hashes = windowRisky[2].CryptoJS.SHA256(data).toString();
    }, capability);
    asynCallNoCap(function() { Handle(hashes); }, tid);
</script>
...
</body>
</html>
```

(b) The modified web page for Chromium with ScriptChecker.

Fig. 10. The modified National Baseball Hall web page used in Section VII-B1. It embeds three malicious scripts with different types of attacks.