FIRM: Capability-based Inline Mediation of Flash Behaviors

Zhou Li, XiaoFeng Wang
Indiana University, Bloomington
{lizho,xw7}@indiana.edu

ABSTRACT

The wide use of Flash technologies makes the security risks posed by Flash content an increasingly serious issue. Such risks cannot be effectively addressed by the Flash player, which either completely blocks Flash content's access to web resources or grants it unconstrained access. Efforts to mitigate this threat have to face the practical challenges that Adobe Flash player is closed source, and any changes to it need to be distributed to a large number of web clients. We demonstrate in this paper, however, that it is completely feasible to avoid these hurdles while still achieving fine-grained control of the interactions between Flash content and its hosting page. Our solution is *FIRM*, a system that embeds an inline reference monitor (IRM) within the web page hosting Flash content. The IRM effectively mediates the interactions between the content and DOM objects, and those between different Flash applications, using the capability tokens assigned by the web designer. FIRM can effectively protect the integrity of its IRM and the confidentiality of capability tokens. It can be deployed without making any changes to browsers. Our evaluation based upon real-world web applications and Flash applications demonstrates that FIRM effectively protects valuable user information and incurs small overhead.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

Security

Keywords

Adobe Flash, Cross-site scripting, Inline Reference Monitor

1. INTRODUCTION

Flash, a multimedia platform first introduced in 1996, has been extensively used today to deliver dynamic web contents, including animations, advertisements, movies and others. Websites such as YouTube serve hundreds of millions of videos every day. Popular portals, such as CNN, Yahoo, etc., broadcast news and host advertisements through Flash contents. The pervasiveness of Flash, however, brings in new security and privacy concerns. Adobe Flash player offers interfaces for a Flash application (Shock-Wave-Flash file or SWF) to operate on the DOM (document object model) objects of its hosting page, other SWF files and file systems. Through ActionScript functions such as `getURL`, Flash content can even inject JavaScript code into web content. Without proper control, it is conceivable that malicious Flash code can wreak such havoc as stealing sensitive information (e.g., cookies, passwords) and modifying high-integrity data (e.g., account balances). Such threats can also come from legitimate yet vulnerable Flash content: as discovered recently [29, 34, 18], a large number of existing Flash applications contain serious security flaws that can be exploited to launch attacks like cross-site scripting (XSS), cross-site request forgery (XSRF), and others. As an example, last year, an XSS flaw was found in the Flash content hosted by the SSL e-banking site of Marfin Egnatia Bank [23], through which an attacker can inject malicious scripts to steal credentials from the bank's customers.

Adobe Flash player provides security mechanisms to control interactions between Flash code and its hosting page: web developers can determine whether a Flash application should be allowed to operate on DOM objects, for example, through script injection. Specifically, when embedding the Flash application, the web developer declares a property named `allowScriptAccess` with one of the values: `always`(scripting allowed), `sameOrigin`(scripting allowed only when the hosting page and Flash code are from same origin) or `never`(scripting prohibited). Once scripting is allowed, the injected code automatically acquires unlimited access to DOM objects. Adobe also controls the interactions between different SWFs according to Same Origin Policy [38]. SWF files and other files are grouped into sandboxes by virtue of the domains they originate from. A Flash application can directly access the resources within its sandbox but needs mediation for any cross-domain access. This security control, again, is black-and-white, which either grants a Flash application and other SWF files it downloads full access or completely denies their access. Such a treatment turns out to be too coarse-grained to be useful for real-world Flash serving websites. Many legitimate Flash applications need script injection. Examples include CNN [5] that lets Flash advertisements utilize JavaScript to enrich their visual effects, and Yahoo [15] that allows such advertisements to track user clicks and profile through scripts. Overly restricting Flash/DOM interactions can significantly reduce the utility of Flash and is often suggested against [17]. As

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

a result, many websites are forced to give Flash code unlimited access, which exposes valuable information assets on the web client to the threat of malicious or vulnerable Flash content.

Our approach. A practical solution to this problem is by no means trivial. Modifying the security mechanism of Adobe Flash player is not feasible, as the software is closed source. Even if this can be done, deployment of a new mechanism requires changes to every client's browser, a slow and painful process. In this paper, we present an effective and convenient alternative. Our techniques allow Flash hosting sites to offer immediate protection to their customers' valuable web contents. This is achieved through an *Inline Reference Monitor (IRM)* system, called *FIRM*, that embeds an access control mechanism entirely into web pages. Through FIRM, the website designer can assign *capability* tokens to different Flash applications the site hosts. Each token is associated with a set of security policies that specify an application's privileges over web contents, including DOM objects and other SWF files. Such a policy is enforced by an *IRM* that wraps both ActionScript functions within SWF files and DOM functions. As an example, consider a Flash advertisement that needs to run a script to track viewers' clicks. Our approach first analyzes the binary code of the Flash, instruments it with a *Flash wrapper* and also grants it a capability token. The new Flash code is served within a web page that also includes a *DOM wrapper* and a set of security policies. When a user is browsing the page, the *Flash wrapper* intercepts the `getURL` call from the advertisement and works with the DOM wrapper to decide whether to let the call proceed based upon the capability.

FIRM offers flexible and fine-grained control over Flash / DOM and Inter-Flash access. It can be completely embedded into a web page by web developers, and therefore avoids any browser-side changes, which makes its deployment instant. Our technique is also reliable: the design of FIRM prevents unauthorized dynamic contents, such as scripts and SWF files, from stealing authorized parties' capability tokens or modifying the IRM and its policy data. We also built a tool for automatic analysis and instrumentation of Flash code. We evaluated our approach on phpBB [11], WordPress [14] and Drupal [6], 3 extremely popular web-design systems, and 9 real-world Flash applications. Our study shows that FIRM effectively mediates Flash behaviors, incurs small overheads and is convenient to use.

Contributions. We summarize the contributions of the paper as follows:

• *Novel Flash mediation techniques.* To the best of our knowledge, FIRM is among the first attempts to enforce inline mediation of Flash/DOM interactions. This is achieved through a novel capability mechanism, which employs randomized, unpredictable tokens to differentiate the access requests that come directly or indirectly (through JavaScript) from the Flash applications with different privileges. Our mechanism can also effectively protect itself from malicious web contents, and automatically instrument Flash applications and web pages.

• *Capability-based inlined mediation of JavaScripts.* The behaviors of Flash applications cannot be effectively mediated without proper control of the scripts they spawn. Different from the prior approach [36] that uses the same set of policies to control all scripts within a web page, FIRM can enforce different policies on different scripts, according to the privileges associated with their capability tokens. This finer-grained access control mechanism is enforced with the collaborations between the DOM wrapper and the Flash wrapper.

• *Implementation and Evaluation.* We implemented a prototype of FIRM and evaluated it on popular web applications and real-world Flash. The outcomes of this study demonstrate the efficacy of our techniques.

Roadmap. The rest of the paper is organized as follows. Section 2 introduces the attack techniques in Flash malware. Section 3 surveys the design of FIRM and the adversary model. Section 4 elaborates the techniques for Flash inline mediation. Section 5 documents our Flash analysis and instrumentation techniques. Section 6 reports our experimental study. Section 7 discusses the limitations of our techniques and future research. Section 8 compares our approach with prior work, and Section 9 concludes the paper.

2. FLASH THREATS

In this section, we briefly review the threats FIRM is designed to mitigate. These threats come from malicious or vulnerable Flash applications.

Illegitimate operations on DOM objects. Through Adobe Flash player, Flash code can directly access DOM objects. Such access, if unmediated, could cause leak of sensitive user data (e.g., cookie) as well as unexpected change of browser behavior. A prominent example is the re-direction attack [21]: a malicious Flash application can redirect the user's browser to a malicious website, where subsequent attacks like drive-by download, phishing, etc. can happen.

Script injection. A Flash application can inject JavaScript code into its hosting web page through ActionScript calls such as `getURL`, which can be exploited to launch an XSS attack. This threat has been widely reported [18, 8, 34]. An example in Figure 1 shows how it happens through vulnerable Flash code. Flash applications (particularly advertisements) often use the variable `clickTag` to receive URLs from its hosting page and redirect the user to these links. This feature can be exploited by the attacker, who can create a link that invokes the Flash code with `clickTag` involving JavaScript code. Once the victim clicks on that link, the Flash injects the script into her web page, using the origin of its hosting page. To eliminate such a threat, a website needs to detect and fix the flaws in every SWF file it stores or links to. This can introduce considerable overhead, particularly for the websites such as Yahoo that host hundreds of thousands of Flash advertisements from other sites. Moreover, malicious scripts can also be injected by malicious Flash applications, which allow them to indirectly access the victim's data.

Figure 1: Vulnerable Flash with XSS flaw

Other threats. Though less known, other attack avenues do exist during Flash/DOM interactions and inter-Flash interactions. For example, once a Flash application shares a single function to another Flash, the Flash player automatically exposes all its functions to the latter. As another example, Flash can export function interfaces to its hosting page, which can be invoked by any script in the page. This channel can be used to bypass the security policy enforced by the Flash player: consider that a Flash application is not

allowed to be touched by another Flash but needs to share its functions to the hosting page; the latter can then inject scripts into the page to gain access to those functions.

Assumptions. Our approach protects the user's valuable web content, such as cookies, passwords and account numbers, from unauthorized access by malicious Flash code or the vulnerable Flash exploited by the adversary. We assume that the website hosting SWF files is not under the control of the adversary and implements FIRM correctly. Also, though FIRM can protect itself against malicious web contents, it is not resilient to a compromised browser or operating system. For example, a malware plug-in can certainly bypass the mediation of our IRM. Such a threat is out of the scope of this work. Finally, FIRM is designed to regulate Flash/DOM and inter-Flash interactions. Other Flash-related attacks, like filling web surfer's clipboard with malicious hyperlinks [2], are not the focus of our approach and left to our future research.

3. OVERVIEW

FIRM includes a *Flash wrapper* for mediating Flash actions and a *DOM wrapper* for controlling the activities of the scripts. Both wrappers interact with a *capability manager* that bootstraps the reference monitor with randomly generated capability tokens, and maintains a *policy base* to map these tokens to the security policies set by the web designer. The wrappers and the capability manager constitute the IRM part of FIRM. The other FIRM component is the tool that automatically embeds the Flash wrapper into a SWF file through analyzing and instrumenting its binary code. Figure 2 illustrates this design.

3.1 Design

The website that uses FIRM first embeds our IRM into the web page that needs protection. Whenever the page is requested by a web client, the site automatically parameterizes it with a set of randomized capability tokens, which are associated with pre-determined security policies (Section 4.1). These policies grant different privileges to different SWF files, as determined by the web designer *a priori*. The token of each Flash application is checked by the wrappers against the policies to control the Flash's access to the web contents (Section 4.2). This idea can be explained with an example in Figure 3, which describes a Flash advertisement with FIRM instrumentation. In the example, FIRM permits the Ad script to get the data necessary for counting a viewer's clicks, but denies its requests to read cookies, passwords and other sensitive information. Our approach can also protect the Flash code through mediating the access of scripts and other SWF files to the call interfaces it exposes to its hosting page (Section 4.2).

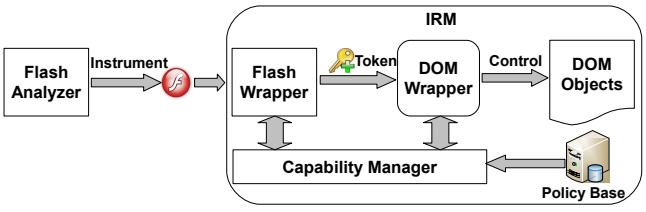

Figure 2: Overview.

To protect the IRM, our approach prohibits other scripts to wrap DOM functions (Section 4.3). This is enforced through regulating the methods (e.g., `__defineGetter__`, `__defineSetter__`) necessary for performing such an operation. The IRM also forbids unauthorized parties to read or write its code and the policy base, or access sensitive FIRM data, such as capability tokens.

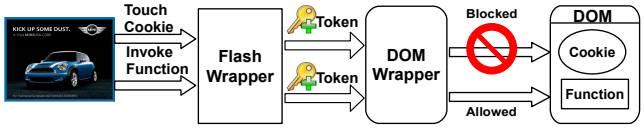

Figure 3: An example demonstrates the design of FIRM.

Though the DOM wrapper and the capability manager can be manually built into a web page by the web developer, an automatic tool is necessary for instrumenting SWF files with the Flash wrapper, as they are often developed by third parties and can be updated frequently. FIRM therefore provides a tool that automatically disassembles the binary code of a SWF file, identifies its access-related function calls (e.g., `getURL`) and internal functions exposed to other domains or JavaScript, and then wraps these functions with mediation code (Section 5).

4. INLINE MEDIATION

Inline Reference Monitor was proposed in [24] as a mechanism to mediate access to Operating System resources. It is built directly into an application's code to control program behaviors, and therefore does not need OS or hardware level supports. Similarly, the IRM used in our research is embedded in web contents, which avoids any changes to the client's browser. In this section, we elaborate our design and implementation of the IRM, which is composed of the capability manager, the Flash and DOM wrappers, and show how these components work together to mediate Flash behaviors and safeguard their own integrity and data confidentiality.

4.1 Capability

To mediate access to DOM and Flash functions, the IRM needs to know the privilege that the caller of these functions possesses. To this end, FIRM adopts a capability mechanism that requires each caller to produce a capability token to gain access. A capability [33] is a token that indicates a set of access rights a subject (e.g., Flash, scripts) has on objects (e.g., cookie, text item, functions). When the subject is about to access a protected object, this operation is checked against the capability: it is allowed to proceed only when the subject has the access right, as specified by a security policy associated with the capability. In FIRM, such capability tokens are maintained by the capability manager, which was implemented as a JavaScript program in our research. Following we elaborate how the mechanism works.

Capability management. Every subject with access policies specified by the web designer is assigned a capability token. An instrumented Flash application acquires its token from the capability manager when it is initialized. Specifically, the Flash exposes a callback function to the JavaScript, through `ExternalInterface.addCallback`. The function is called by the capability manager to parameterize the Flash with its token and the related security policies, and disabled by the IRM afterwards to prevent unauthorized invocations. This treatment avoids hard-coding tokens into these applications, a process that requires recompiling the applications each time their hosting web page is requested by a web client.

The capability token used in FIRM is a random string. It is designed to be sufficiently long ($\geq$ 10 bytes) to defeat a brute-force attack in which the adversary tries to use random guesses to produce a correct token. Each capabil-

ity token is associated with a set of security policies that specify a subject's access rights to different objects. The capability manager organizes those tokens and their policies into a *policy base*, which is stored in a local variable within a function called `Checker`. `Checker` encapsulates the policy base to mediate the access to its content. To retrieve policies, one has to call the function with a capability token. The IRM also hides its own capability in local variables and controls all the channels to read the code of `Checker` and its other functions. This technique is elaborated in Section 4.3. Another measure FIRM takes to prevent leaks of tokens to unauthorized parties is prohibiting a SWF file to share its capability with others. To this end, the mediation code our Flash analyzer injects into the SWF utilizes randomized, unpredictable names for the variables involved in capability-related operations to preclude any references to them from other part of the Flash code (see Section 5).

Table 1: Protected objects and properties

Type	Objects	Properties
DOM	Document	cookie, domain, lastModified referrer, title, URL
	Window	defaultStatus, status
	Location and Link	hash, host, hostname, href, search, port, protocol, pathname, toString
	History	current, next, previous, toString
	Navigator	appCodeName, appName, systemLanguage userAgent, userLanguage, platform
	Form	action
	Form Elements	checked, defaultChecked, defaultValue name, selectedIndex, toString, value
	Text	innerHTML, innerText
Flash	Functions	-

Security policies. Security policies are specified by the web designer for controlling subjects' access rights. A policy can be described by a 4-tuple $< s, o, a, c >$, where s, o and c denotes a subject, an object and the capability of the subject respectively, and a is the action s requests to perform on o. In FIRM, a subject is either a Flash application or the JavaScript code from a specific domain; an object describes DOM objects, or the functions or variables of JavaScript code and SWF files (see Table 1 for examples); the action element in the tuple can be "*read*", "*write*", "*execute*" or left blank to indicate denial of access.

Whenever a capability token appears with a request from a subject s' with an operation a' on the object o', the IRM searches the policy base to retrieve all the policies containing c'. The request is permitted if one of the policies contains s', o', a' and c', and denied otherwise. FIRM also includes a set of default policies specified by the web developer, which use a wildcard symbol '*' to match any subject, object, action or capability token. When the symbol is applied to c in the tuple, the policy is used on a subject that does not carry any capability. The default policies can be used to define a set of basic operations open to even untrusted Flash content or scripts, for example, a `read` on a nonsensitive text item, or avoid verbose specifications of the permissions for every subject/object pair. They are overruled by other policies once a conflict happens. For example, given two policies $< s, cookie, , c >$ and $< s, *, read, c >$, a Flash code s is denied the access to the cookie.

4.2 Mediation

The objective of inline mediation is to ensure that the web content under our protection can only be accessed by subjects with sufficient privileges, as indicated by their capability tokens. To this end, we designed and implemented a DOM wrapper and a Flash wrapper that work together to control both Flash/DOM interactions and inter-Flash interactions. We elaborate this approach below.

Flash and DOM wrappers. To mediate Flash code and its script's access to web contents, we need to control DOM and ActionScript functions used in such an access. This is achieved in our research through wrapping these functions with mediation code. Specifically, we developed two wrappers, the DOM wrapper that controls DOM functions and the operations of scripts, and the Flash wrapper that mediates the use of ActionScript functions.

Sample code for wrapping the getter of document.cookie

```
//1. get pointer to the old getter
var oldGetter = document.__lookupGetter__("cookie");
//2. define the new getter
function newGetter() {
   if(Checker(currentToken))
      return oldGetter();
   else
      throw "unauthorized access";
}
//3. replace the old getter with new getter
document.__defineGetter__("cookie", newGetter);
```

Figure 4: An example of redefining getter.

The DOM wrapper redefines the `get` and `set` methods of the DOM objects that need to be protected. Most DOM objects, such as document, window, forms, and the input box offer these methods for scripts to read or write their properties such as cookies, locations and others. In Mozilla Firefox 3.5, FIRM wraps these methods through `__defineGetter__` and `__defineSetter__`, two methods specified under `Object.prototype`. Other browsers, including IE8, Google Chrome, Safari and Opera, use different methods, as described in Table 2. The web server that implements FIRM can use scripts to identify the type of the browser a client is running (from `Navigator.appName`) before rendering a web page that wraps its methods. Figure 4 illustrates an example in which the `get` method of `document.cookie` is supplemented with the code that mediates the access to the property. Different from most properties, `document.location` and `window.location` do not have `get` and `set`. On the other hand, these two properties need protection because otherwise, untrusted scripts can modify them to redirect the browser to malicious websites. Our solution is to make use of the method `Object.watch` to monitor these properties: once the method detects that the properties are about to be changed by the party without a proper capability[1], the IRM simply aborts the redirection operation if the target is not permitted. Our approach also wraps DOM functions like `document.alert`, which pops up windows (e.g., an alert). These functions could be used in social engineering, and therefore need mediation. Other part of the hosting page that the DOM wrapper modifies includes the JavaScript code for accessing DOM objects or Flash resources (through the call interfaces exposed by SWF files). Such code is instrumented to add in the mechanism that checks the capability tokens of the party invoking it or attaches its token to every access request it makes.

The Flash wrapper controls ActionScript functions like `getURL`, `navigateToURL`, `ExternalInterface.call` and `fscommand`. These functions can be used to invoke the script

[1] `watch` can intercept the operations that modify the objects it is monitoring. An authorized party who wants to change the object needs to place its capability token to a "mark" variable, which is discussed later in this section.

Table 2: API variations in different browsers

Browser	Define Getter	Define Setter	Get Setter	Get Setter	Watch
Mozilla Firefox 3.5	__defineGetter__	__defineSetter__	__lookupGetter__	__lookupGetter__	watch
IE 8	defineProperty	defineProperty	getOwnPropertyDescriptor	getOwnPropertyDescriptor	onPropertyChange
Google Chrome 4	__defineGetter__	__defineSetter__	__lookupGetter__	__lookupGetter__	-
Safari 4	__defineGetter__	__defineSetter__	__lookupGetter__	__lookupGetter__	-
Opera 10	__defineGetter__	__defineSetter__	__lookupGetter__	__lookupGetter__	-

already in the hosting web page or inject new scripts. Individual SWF files stored in the hosting website are also instrumented with the code for acquiring capability tokens and their policies from the capability manager as soon as they are bootstrapped. This is achieved by exposing an ActionScript function to JavaScript, through which the capability manager parameterizes the SWF file. The mission of the Flash wrapper includes letting a Flash program use its capability to execute JavaScript code, and protecting its functions from being misused by scripts or other Flash applications.

Mediating access to DOM objects. A Flash application relies on JavaScript code to access DOM objects. To mediate the access, FIRM wraps all the ActionScript calls related to JavaScript, as discussed above. Whenever the Flash makes such a call, the Flash wrapper supplies the capability token of the Flash to the call, and the mediation code inside the JavaScript functions to be invoked calls `Checker` to look up the security policies regarding the token and makes access decision based upon the policies.

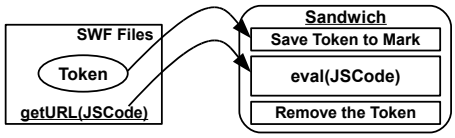

Figure 5: Sandwiching the injected script.

A challenging problem is how to let the JavaScript code injected by a Flash application run at the Flash's privilege. The IRM may not have access to such code until the runtime: for example, the code can be downloaded by the Flash from another site. Automatic analysis of JavaScript code is well known to be hard [27], which makes it difficult to instrument the code on the fly. We tackled this problem by leveraging a special feature of JavaScript: JavaScript code in a web page actually runs in a single thread, and as a result, its execution is sequential [42][2]. This feature allows us to develop a "marking" mechanism that labels the script code running on a Flash program's behalf. Specifically, the DOM wrapper maintains a "*mark*" variable, which is initialized to zero and later used to keep the capability token of the running script. After the Flash wrapper identifies a script injection operation in a Flash program, for example, from the prefix `javascript:` within the input content of `getURL`, it sets the script code as the input string to an `eval` command, and inserts one JavaScript command before the `eval` to set the mark to the Flash's capability and one after to zero out the mark. This transformation, which we call "sandwiching", is illustrated in Figure 5. When the script is running, the IRM refers to the mark for the script's privilege. Note that other scripts cannot read the mark before the sandwiched code runs to completion, due to the sequential execution of JavaScript [42]. On the other hand, the code cannot escalate its own privilege by changing the mark, as it does not know other capability tokens.

As stated in Section 2, a Flash application can redirect a visitor to a malicious site and install malware. To defend against this attack, FIRM mediates functions like `getURL`: if the input parameters of these functions are found to contain URLs (started with http, for example), they are used to check against a whitelist; only redirection to the URLs on the list are allowed.

Mediating access to Flash. A Flash application can choose to expose some of its functions (through Actionscript calls such as `ExternalInterface.addCallback`) to let JavaScript code access their resources. A problem here is that there is no restriction on who can call these functions. For example, a malicious Flash program can take advantage of these functions to gain access to another Flash that it is not allowed to access within the Flash player.

Our solution to this problem is instrumentation of the exposed functions. Mediation code, as part of the Flash wrapper, is inserted to the beginning of such functions after static analysis of Flash code. Once an exposed function is invoked, our code checks the capability token supplied by the caller, and decides whether to let the call go through according to the security policies tied to the token.

Inter-Flash access control. Adobe Flash player maintains a boundary between different Flash applications. Such a boundary, however, can be crossed if one Flash shares its functions to another Flash through a `LocalConnection` object. The problem here, again, comes from the "black-and-white" strategy adopted by the Flash player: a Flash program shares either *all* its functions or none at all. A serious consequence of this treatment is that untrusted Flash code can call the function of privileged Flash code to gain access to the resources it is not entitled to, once the latter inadvertently exposes its functions. Our solution to this problem, again, is based upon code instrumentation and call wrapping: for the Flash application that is found to build a `LocalConnection` with others, our analysis tool instruments all its functions with mediate code; the code checks the caller's capability token once a function is invoked, and aborts the call if the token does not carry a sufficient privilege.

A Flash application can load another Flash with `load` or `loadBytes` as a resource and then use `addChild` to make the latter its child Flash. When this happens, the child acquires the full access to the father's resources, including functions, variables and others, and is able to leak them out. FIRM mitigates such a threat by automatically reducing a Flash's privilege once it is found to have downloaded untrusted child Flash.

4.3 Protecting FIRM

Since an IRM works on the same layer as the subjects it controls, it is under the threats these subjects pose. Without proper protection, FIRM can be subjected to various

[2]The registered user events are triggered sequentially: they cannot be executed until the script stops running. Similarly, delayed execution with function `setTimeout` is also sequential.

attacks from malicious scripts [25] or Flash applications, including compromising the integrity of its code and policies, and stealing its capability tokens. In this section, we elaborate the measures our approach takes to mitigate these threats.

Integrity Protection. The obvious targets of attacks are the DOM and Flash wrappers. As Flash content is not able to alter its code in runtime (See Section 5), malicious Flash code cannot get rid of the Flash wrapper after it is instrumented. This feature automatically ensures the integrity of the Flash wrapper. Hence, our integrity protection is focused on the DOM wrapper.

As discussed before, the DOM wrapper mediates the `get` and `set` functions of important DOM objects. The adversary may try to replace the wrapper with his own functions. To eliminate this threat, our IRM has been designed to wrap these important objects before any other subject, and block any request without a proper privilege to change the getters and setters of the objects. This is achieved through mediating the methods `__defineGetter__` and `__defineSetter__`. To prevent malicious scripts from tampering with the wrappers for these methods, we employ `Object.watch`[3] (`onPropertyChange` in IE8) to monitor the operations on the methods: any change to their function pointers will be detected by `watch` and aborted by the IRM before it happens. The `watch` method itself is protected in the same way: it is watching itself and interrupts any attempts to replace it. The IRM also mediates all the methods of `prototype`, a property under `Object`, `Array` and `Function`. This is necessary for protecting the functions associated with these global variables, such as `toString`, which could also be modified by the adversary [19].

Prior research [36] discovered that a malicious script can delete all wrapped objects from the memory, which could lead to the restoration of the original, unwrapped objects. This threat, however, is limited to Firefox, and can be eliminated by setting constraints on the deletion operation, which is permitted under Standard ECMA-262 5 [12], the next generation JavaScript specification. FIRM also takes measures to mitigate the threat: once the IRM finds itself in Firefox and a Flash program is about to execute the scripts within the "sandwich" (See Section 4.2 and Figure 5), the Flash wrapper works with the DOM wrapper to calculate the hash values of the instrumented DOM methods and their function pointers and save them to the variables within the Flash wrapper. They also move all the valuable data, such as cookies, into the Flash wrapper. After the execution of injected scripts, the Flash wrapper verifies the integrity of these function pointers and methods. If no foul play is found, the valuable data is restored. Otherwise, it aborts its operation and warns the user through a pop-up window. The Flash variables used to save such data assume randomized names to protect them from being accessed by the original, uninstrumented Flash content. They are also beyond the reach of injected scripts, as they are located within the Flash.

Confidentiality Protection. The most sensitive FIRM data are capability tokens, which, once seized by unauthorized parties, can be used to escalate their privileges. During the operations of FIRM, these tokens are stored in the local variables of JavaScript and the variables of SWF files. Since these local variables cannot be referred by the script code outside their related functions, the only way an unauthorized party can access the capabilities is to read the code of these functions. This path is also blocked by the IRM, which is configured to allow none but itself to read its code, through mediating the `get` methods for the `innerHTML` and `innerText` properties under its `script` object, and the `toString`, `toSource` and `valueOf` methods under the prototype of `Function` object.

To protect the capability tokens stored in the instrumented Flash code, the names of the variables that accommodate these tokens are randomized, making them unpredictable to the adversary. Such an operation only needs to be performed once when instrumenting the code. As a result, a malicious Flash program is unable to access the capability tokens and other data stored in the Flash wrapper. Note that an instrumented Flash does not carry hard-coded capability token. Instead, it gets the token from the capability manager once it is bootstrapped by the Flash player.

As discussed in Section 4.2, the variables and functions of a Flash program are completely exposed to the child Flash it downloads. This lets the child inherit the father's capability, which can be risky in some circumstances. For example, a Flash-based video player could run an untrusted Flash Ad as its child. It is evidently undesired to grant the Ad the privilege of the video player. We solve this problem by instrumenting the ActionScript calls for downloading and creating a child Flash: once a child Flash is found to come from an untrusted domain, our mediation code automatically lowers down the father's privilege.

5. FLASH ANALYSIS AND INSTRUMENTATION

To embed the Flash wrapper into a Flash, we need to analyze its binary code and instrument the code with mediation mechanisms. As Flash contents are often submitted by the third party and in binary forms, manual analysis of such contents can be time consuming and even unrealistic for the Flash serving sites like Yahoo that receive a large number of uploads every day. In our research, we developed an automatic Flash analyzer to work on Flash code. Our analyzer can decompile a binary SWF file, identify the functions related to resource access and wrap them with FIRM instrumentation. In this section, we elaborate the design and implementation of this tool.

ActionScript. ActionScript is a scripting language based on ECMAScript [7], which is designed to control the behavior of Flash. Compared with JavaScript, ActionScript code is easier to analyze and instrument: 1) a Flash program is not allowed to modify its code during the runtime, which makes a malicious Flash impossible to get rid of our instrumentations; (2) Flash cannot parse and execute an input string as `eval` does, which avoids the complication in statically analyzing such a string; (3) a Flash program cannot access the code and data within another Flash without permission. These features allow us to perform a static analysis of Flash code to add mediation to the code.

Static analysis of Flash. The prototype we built first utilizes SWFScan [13], a free decompiler, to convert SWF binaries into the ActionScript code. Then, it identifies the program locations from the code where instrumentations need to be done. Specifically, our implementation looks for four types of ActionScript APIs: `getURL` and `navigateToURL` that

[3]According to ECMA-262 5 [12], any property of a JavaScript object has an attribute named `Configurable`. When set to `false`, assigning new value to the property will throw an exception, which achieves the same goal as `Object.watch`. Though the current versions of Google Chrome, Safari and Opera do not support the function, they will certainly move towards this standard.

allow a Flash to inject scripts into its hosting page, `ExternalInterface.call` and `fscommand` that enable the Flash to call a JavaScript function defined in the page, `ExternalInterface.addcallback` that lets Javascript call ActionScript functions, and `LocalConnection` that shares the functions of one Flash program with others.

To accurately locate these functions, we parsed the Flash code into a grammar tree. The parser implemented in our prototype was generated by ANTLR (or ANother Tool for Language Recognition) [3], a popular parser generator. We manually translated ActionScript grammar into the form accepted by ANTLR, which then converted it into an LL parser. From the grammar tree the parser creates, our analyzer can identify both the direct use of script-related APIs, for example, a call to `getURL()`, and the indirect use, for example, `var a = getURL; a();`.

Instrumentation. After relevant program locations are identified, our tool automatically instruments the code at these locations with mediation mechanisms. The mediation code allows the caller to supply its capability token for privilege checking, and invokes the original function if the caller is authorized. After instrumentation, the new Flash program is compiled into a SWF binary using the compiler of Adobe Flash CS [1].

Discussion. Though ActionScript is easier to analyze than JavaScript, it does include some language features that can be used to obfuscate its code. Particularly, ActionScript 2.0 allows the `_root` object to invoke a function through parsing an input string. For example, `_root['getURL']()` will be interpreted as `getURL()` during an execution. This technique, however, seems no longer supported by ActionScript 3.0. Moreover, some API functions like `asFunction` can be exploited to inject scripts [8]. To mitigate the threat, we can mediate their operations using the Flash wrapper. Note that all these language features are not frequently used by legitimate Flash code. On the other hand, a malicious Flash that uses them to obfuscate its code could end up decreasing its privileges, as uninstrumented calls cannot use any capability tokens. They are ensured by FIRM to have nothing but the lowest privilege.

6. EVALUATION

We evaluated our implementation of FIRM on real web applications and Flash contents. Our objective is to understand the effectiveness of our technique in mediating Flash activities, and the performance impacts it could bring to web services. In this section, we first explain our experiment settings (Section 6.1), and then elaborate on this experimental study and its outcomes (Section 6.2 and 6.3).

6.1 Experiment Settings

Here we describe the web applications, Flash and computation platforms used in our study.

Web applications. We utilized three extremely popular open source web-design systems in our study:

• *phpBB* is one of the mostly used open source forum solutions, with millions of installations worldwide [11]. It serves as a template, which one can customize, e.g., adding plugins, to build her own forum. The forums based upon phpBB allow users to post Flash contents through the tag `[Flash]` in *BBCode* [4].

• *WordPress* is a blog publishing application known as the largest self-hosted blogging tool, with millions of users worldwide [14]. Through the application, a blog author can publish Flash content, which is handled by a plugin called *Kimili Flash Embed* [9].

• *Drupal* is an open source content management system (CMS) that supports a variety of websites ranging from personal weblogs to large community-driven websites [6]. The system can be used to publish different types of web contents, including Flash.

Flash. Also used in our experiments were 9 real-world Flash applications, as illustrated in Table 3. Specifically, we utilized 3 vulnerable Flash advertisements, a malicious Flash game and a Flash player to understand whether our technique can effectively control the scripts they invoke. To study our protection of the call interfaces Flash exposes to scripts, an experiment was conducted to let malicious Flash code spawn scripts that attempted to access the functions another Flash exported to its hosting page. Inter-Flash access control was evaluated with another pair of Flash applications: one attempted to share only some of its functions, whereas the other tried to access other functions. All these Flash applications were instrumented with our analysis tool and executed within the aforementioned web services.

Computation platforms. All our experiments were conducted on a laptop with 3G memory and 2G Dual-core CPU. The laptop ran Windows Vista, with Apache 2.2.9/PHP 5.2.6 as web server and MySQL 5 as database server. Our experiments were conducted in Firefox 3.5 and IE 8.

6.2 Effectiveness

Installing FIRM. We modified these web applications to install the DOM wrapper and the policy manager. A phpBB-based forum could receive posts with Flash contents, for example, `[flash]a.swf[/flash]`, which will be activated in a viewer's browser[4]. In the absence of mediation, such a Flash can spawn scripts to compromise the integrity and confidentiality of the viewer's information assets. To embed our IRM into the web pages generated by the forum, we changed a PHP file `tpl/prosilver/viewtopic/body.html.php`, making the program inject the JavaScript code of the IRM into every web page it created. Similarly, a blog publishing system built upon WordPress can be used by the malicious blogger to spread Flash malware. In our experiment, we inserted code to `index.php` under the folder `wp-content/themes/classic` so as to embed our IRM scripts into the web content produced by the system. A website developed using Drupal can also inadvertently include malicious Flash contents, for example, an advertisement Flash that picks a web surfer's cookie. Like the other two applications, the system includes a PHP program `page.tpl.php` under `themes/garland` for dynamic page generation. This program was also modified in our research to build our IRM into its pages. We also ran our Flash analyzer to instrument the aforementioned Flash applications.

Experiment outcomes. To evaluate the effectiveness of FIRM, we first attacked the unprotected web applications through the Flash contents they hosted, and then made the same attempts on these applications when they were under the protection of FIRM. Following we elaborate our findings.

The mediation on Flash's access to DOM objects was studied using the five Flash applications described in Section 6.1 and Table 3. Without mediation, we found that the malicious Flash game could steal the cookie from the hosting page and the three Flash advertisements could be exploited to launch XSS attacks. Once the IRM was activated, all

[4]We changed a PHP file `bbCode.html` under `styles/prosilver/template` of phpBB3 to allow Flash executing JavaScript code.

Table 3: Effectiveness

Type	No	Flash	Operations	Result		
				phpBB	WordPress	Drupal
Flash to DOM	1	Puzzle Game	read cookie	Reject	Reject	Reject
	2	Adobe Demo	read cookie	Reject	Reject	Reject
	3	CNN Ad	change Location	Reject	Reject	Reject
	4	CNET Ad	read user account text	Reject	Reject	Reject
	5	Flow Player	O_1: read Location	O_1 Reject	O_1 Reject	Allow
			O_2: call script function	O_2 Allowed	O_2 Allow	
DOM to Flash	6	Color Widget	expose functions	-	-	-
	7	Invoker	call functions of Color Widget	Reject	Reject	Reject
Flash to Flash	8	Sender	call functions of Receiver	Reject	Reject	Reject
	9	Receiver	receive message	-	-	-

these attacks were found to be successfully deflected. Specifically, the instrumented phpBB and WordPress adopted the security policies that disallowed Flash contents to access any document objects. As a result, we observed that the JavaScript calls for accessing cookies, initiated by the Flash game and the Flash advertisements, were all blocked. A side effect was that the legitimate Flash, the Flow player, could not access the URL of the hosting page either. The policies specified for Drupal differentiated the legitimate Flash, Flow Player, from the other Flash applications: the player was allowed to access document objects, except the cookie, whereas the Flash games and advertisements were denied the access. We found that these policies were faithfully enforced by our implementation, which defeated all our attacks without interfering with the legitimate Flash's operations.

We employed two Flash applications to evaluate the protection of the call interfaces a Flash exposed to JavaScript. One of these applications share its functions with the hosting page, which could be accessed by the other Flash from a different domain through injecting script. After instrumenting the exposed functions with mediation code, we observed that the second Flash was no longer able to use the functions without legitimate capability tokens.

We also studied the situation when two Flash applications attempted to share resources between them. One Flash tried to let the other Flash call some but not all of its functions. In the absence of FIRM, this could only be done through establishing a `LocalConnection`, which unfortunately made all functions within the first Flash available to the other. After the IRM was installed, we could assign the second Flash a capability token that only granted it access to some of these functions. Such security policies were found to be successfully enforced by our prototype.

6.3 Performance

The performance of FIRM was evaluated in our research from three perspectives: (1) the performance impact of FIRM on page loading, (2) runtime overheads for mediating DOM and Flash operations and (3) the cost for performing static analysis on SWF files. We elaborate this study below.

Table 4: Performance of page loading

	phpBB3 (s)	WordPress (s)	Drupal (s)
No FIRM	3.927	1.66	3.555
FIRM	4.117	1.923	3.96
Overhead	4.80%	15.80%	11.40%

Bootstrapping FIRM could add further delay to a page loading process. To understand such a performance impact, we measured the page loading time of the three web applications described in Section 6.1 using a plugin (e.g., Firebug [35] in Firefox). The experiment was designed to compare the loading time of unprotected pages with that of instrumented pages. We collected the data from 10 independent tests under the settings when FIRM was installed and when it was not to compute averages, which are reported in Table 4. As we can see from the table, the overhead incurred by FIRM was reasonable: it was kept below 20% in the worst case (WordPress), and acceptable for other web applications.

Table 5: Mediation overheads

	Flash to DOM(ms)		DOM to Flash(ms)		Flash to Flash(ms)	
Tasks	Read cookie		Call func		Call func	
Browsers	IE	FF	IE	FF	IE	FF
No FIRM	0.641	0.799	6.52	4.92	0.675	0.67
FIRM	0.995	1.87	6.81	4.96	0.686	0.698
Overhead	55.20%	134.00%	4.40%	1%	1.60%	4%

We further studied the overheads incurred by instrumented operations, which include the delay caused by mediating the interactions between Flash and JavaScript, as well as those between different Flash applications. We collected data from 1000 independent tests of individual operations, with or without mediation. Averaged delays computed from such data are displayed in Table 5. From the table, we can observe that mediation was lightweight, incurring an overhead of 5%. A more significant delay appeared when the IRM was controlling the JavaScript code invoked by a Flash, which went up to 134%. A closer look at the overhead revealed that it was caused by `eval` the IRM employed to wrap the injected code (Section 4.2). Running JavaScript within the function turned out to be more time-consuming than a direct execution of the code in a hosting page. However, given the small execution time of the code, the delay introduced thereby was actually hard to notice in practice.

The overhead of analyzing and instrumenting Flash consists of the latencies incurred by decompiling binary code, analyzing, instrumenting and compiling the source code. We measured these latencies from the 9 Flash applications used in our study, each of which was run 10 times to get the average. The outcomes are presented in Table 6. The table shows that in most cases, the whole analysis took less than 10 seconds on the low-end laptop used in our experiment. Analyzing and instrumenting Flow player takes over one minute as it contained over 20,000 lines of code, while most Flash programs, particularly advertisements, are much smaller, typically below 1,000 lines.

Table 6: Perfomance of static analysis

Flash	Decompile (s)	Analysis (s)	Compile (s)	Total (s)
Puzzle Game	3.46	0.665	2.66	6.785
Adobe Demo	2.3	0.563	2.62	5.483
CNN Ad	4.9	0.723	2.8	8.423
CNET Ad	6.08	0.865	2.9	9.845
Flow player	14.6	3.582	49	67.182
Color Widget	3.6	0.848	3.98	8.428
Invoker	2.88	0.571	2.3	5.751
Sender	2.82	0.717	2.1	5.637
Receiver	2.5	0.631	2.74	5.871

7. DISCUSSION

FIRM is designed to be the first inline policy enforcement system that mediates Flash/DOM and Flash/Flash interactions. Also of great importance to Flash security, naturally, is well-designed security policies. The current design of FIRM can support simple policies, as described in Section 4.1. These policies seem to be sufficient for mitigating traditional threats such as XSS [29, 34, 18]. However, questions remain whether they offer enough protection against the new threats posed by malicious Flash, for example, seizure of the clipboard [2]. Further study is needed to understand this problem and improve FIRM to support more complicated policies, if necessary.

FIRM instruments the dynamic contents including Flash and JavaScript located at the websites that adopt our technique. For the Flash or scripts downloaded to the client's browser from other domains during the runtime, the control we could achieve is still coarse-grained: our current treatment just grants them the lowest privilege. A more desirable approach could be applying different policies to the dynamic contents from different domains. This could requires establishing certain trust relations between websites. Alternatively, our IRM could pass the scripts and Flash acquired during the runtime to its website (the one that offers the hosting page) for analysis and instrumentation. Study of these approaches is left to our future research.

As discussed in Section 4.3, a Flash can download and run another Flash as its child. The child Flash, which can be untrusted, inherits the privilege of its father. Our current solution is de-escalation of the father's privilege, which results in rather coarse-grained control. In the follow-up research, we plan to look into the possible approaches that can be used to mediate the child's activities without demoting the father.

The techniques we propose can be applied more generally: for example, they can be extended to mediate JavaScript code from different domains. On the other hand, our approach cannot protect a web service from a denial of service attack: for example, a malicious Flash or script can delete DOM objects to disrupt the normal operations of the service. Further research is needed to understand the feasibility of making IRM more resilient to the attack.

8. RELATED WORK

Inline reference monitor. The idea of moving a reference monitor into an application has been applied to protect binary executables [45, 16, 39] and Java applications [24, 20]. Compared with other access control mechanisms, an IRM is often more efficient and has more information about an application's internal states, but can also be more prone to the attacks that aim at its integrity and data confidentiality. Concurrently with this research and independently, Phung et. al. [36] proposed a JavaScript IRM that mediates accesses to sensitive DOM objects and properties. A problem with this approach is that all the scripts within a web page are granted the same privilege. In contrast, FIRM offers a fine-grained control of the scripts and Flash applications with different privileges, according to their capabilities. Another concurrent work from Meera et. al. [41] devised a Flash IRM to verify if certain functions violate pre-defined policies. To mitigate XSS attack, their framework sanitizes the input of the functions like `getURL`. Nevertheless, this approach is black-and-white which only allows or prohibits the whole script from input. Conversely, our framework can allow the legitimate script code while prohibits the malicious one.

Access control in web contents. The rapid development of new web services and applications, such as Mashup [10], makes the classic Same Origin Policy [38] increasingly insufficient for mediating dynamic web contents. New policy models and enforcement platforms, for example, MashupOS [44], OMash [22], xBook [40] and BFlow [46], are proposed to achieve finer-grained control of web activities, particularly those involving JavaScript. FIRM is designed to control Flash applications and the scripts they spawn, which has not been done before. Moreover, all these existing approaches require installing browser plug-ins. This raises the bar for their practical deployment. Our approach, however, does not need the web client to do anything: all the policies and enforcement mechanisms are completely embedded in the web pages delivered to the browser, and therefore can be deployed easily. Grier et. al. proposed a new browser named OP Browser [26] which embeds security policy in browser kernel to mediate the access from plugins like Adobe Flash player. However, their approach does not differentiate the security demands of different Flash contents and turns out to be too coarse-grained.

XSS defense. As a well-recognized threat to integrity and confidentiality of valuable web contents, XSS has received great attentions from security researchers. Prominent countermeasures include Beep [30], BrowserShield [37], Noxes [32], and BluePrint [43]. Different from the prior work, FIRM focuses on the XSS caused by vulnerable or malicious Flash applications. Controlling such a threat needs effective mediation of the interactions between Flash contents and JavaScript, which has not been explored by the prior research.

Instruction set randomization. FIRM protects its IRM through randomizing capability tokens, and the JavaScript and ActionScript variables that maintain those tokens and their related policies, which makes these critical resources out of the reach of malicious web contents. This idea has been inspired by previous research on Instruction Set Randomization (ISR) [31]. ISR was designed to defeat code-injection attack, through creating process-specific randomized instruction set. Recently, researchers move towards utilization of the technique to protect web applications. A prominent example is Noncespaces [28], which randomizes the namespace prefixes within a document to eliminate the scripts not created by the server. However, such a control can be coarse-grained: for example, the reference monitor either permits or denies execution of script code, but cannot decide what resources a running script can access.

9. CONCLUSION

Flash contents have been increasingly utilized for video

playing, advertising and other purposes. However, it is revealed [29] that Flash can be exploited by an adversary to launch various attacks, including XSS and XSRF. The intrinsic protection of Adobe Flash player is not sufficient in that it either denies a Flash's access to web resources or gives it unconstrained access. Patching such a security mechanism turns out to be nontrivial: Adobe Flash player is closed source, and deploying the patch on every browser cannot be accomplished easily. In this paper, we present FIRM, a novel solution that avoids these hurdles while still achieving effective mediation of Flash activities. FIRM builds an inline reference monitor into the web page hosting Flash contents. The IRM effectively mediates the interactions between Flash and DOM objects, and between different Flash applications, according to the capability token possessed by the Flash. Our approach protects the IRM through controlling DOM methods and randomizing the names of the variables that hold sensitive data, such as capability tokens. We implemented a prototype of FIRM and evaluated it on popular web applications, including phpBB, WordPress and Drupal, and 9 real-world Flash applications. Our study shows that the technique effectively protects user data, incurs small overheads and is convenient to deploy.

10. ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful comments. This work was supported in part by the NSF under Grant No.CNS-0716292 and CNS-1017782.

11. REFERENCES

[1] Adobe flash cs4. http://www.adobe.com/products/flash/.
[2] Adobe flash player clipboard security weakness. http://www.securityfocus.com/bid/31117.
[3] Antlr parser generator. http://www.antlr.org/.
[4] Bbcode. http://www.bbcode.org/.
[5] Cnn. http://http://www.cnn.com.
[6] drupal community pluminbing. http://drupal.org.
[7] Ecmascript. http://www.ecmascript.org.
[8] Flash url parameter attacks. http://code.google.com/p/doctype/wiki/ArticleFlashSecurityURL.
[9] Kimili flash embed. http://kimili.com/plugins/kml_flashembed/.
[10] Mashup dashboard - programmableweb. http://www.programmableweb.com/mashups.
[11] phpbb - creating communities worldwide. http://www.phpBB.com.
[12] Standard ecma-262. http://www.ecma-international.org/publications/standards/Ecma-262.htm.
[13] Swfscan. https://h30406.www3.hp.com/campaigns/2009/wwcampaign/1-5TUVE/index.php?key=swf.
[14] Wordpress - blog tool and publishing platform. http://wordpress.org.
[15] Yahoo! http://www.yahoo.com.
[16] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, pages 340–353, 2005.
[17] Adobe. Flash player security - controlling outbound url access. http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3/WS5b3ccc516d4fbf351e63e3d118a9b90204-7c9b.html, 2009.
[18] Y. Baror, A. Yogev, and A. Sharabani. Flash parameter injection. Technical report, IBM, As of September 2008.
[19] A. Barth, C. Jackson, and W. Li. Attacks on javascript mashup communication. In *Proceedings of Web 2.0 Security and Privacy 2009 (W2SP 2009)*, 2009.
[20] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM.
[21] S. Chenette. Malicious flash redirectors - security labs blog. http://securitylabs.websense.com/content/Blogs/3165.aspx, 2008.
[22] S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *Proceedings of the 15th ACM conference on Computer and communications security table of contents*, pages 99–108. ACM New York, NY, USA, 2008.
[23] DP. Flash clicktag parameter xss. banks, e-shops, adobe and others vulnerable. http://xssed.org/news/98/Flash_clickTAG_parameter_XSS._Banks_e-shops_Adobe_and_others_vulnerable/, 2009.
[24] U. Erlingsson and F. B. Schneider. Irm enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.
[25] Google. Attackvectors. http://code.google.com/p/google-caja/wiki/AttackVectors, 2010.
[26] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 402–416, Washington, DC, USA, 2008. IEEE Computer Society.
[27] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.
[28] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *NDSS'09: Proceedings of the 16th Network and Distributed System Security Symposium*, 2009.
[29] P. Jagdale. Blinded by flash: Widespread security risks flash developers don't see. In *Black Hat DC 2009*. Hewlett-Packard, 2009.
[30] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.
[31] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM.
[32] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337, New York, NY, USA, 2006. ACM.
[33] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.
[34] S. D. Paola. Testing flash applications. In *6th OWASP AppSec Conference*, 2007.
[35] I. Parakey. Firebug - web development evolved. http://getfirebug.com/, 2009.
[36] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, New York, NY, USA, 2009. ACM.
[37] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *Proc. OSDI*, 2006.
[38] J. Ruderman. The same origin policy. http://www.mozilla.org/projects/security/components/same-origin.html, 2008.
[39] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
[40] K. Singh, S. Bhola, and W. Lee. xbook: Redesigning privacy control in social networking platforms. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.
[41] M. Sridhar and K. W. Hamlen. Actionscript in-lined reference monitoring in prolog. In *Proceedings of the Twelfth Symposium on Practical Aspects of Declarative Languages (PADL)*, 2010.
[42] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in javascript. In *25th Annual Computer Security Applications Conference (ACSAC)*, 2009.
[43] M. Ter Louw and V. Venkatakrishnan. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *30th IEEE Symposium on Security and Privacy*, May 2009.
[44] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashupos. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, pages 1–16, 2007.
[45] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.
[46] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with bflow. In *EuroSys'09*, 2009.