JavaScript: Bringing Object-Level Security to the Browser

Charlie Meyer and Maurice Rabb
University of Illinois at Urbana-Champaign, {cemeyer2, m3rabb}@illinois.edu

***Abstract* – JavaScript has evolved from a simple language intended to give web browsers basic interaction into a fully featured dynamic language that allows the browser to become an application delivery platform. With innovations such as asynchronous JavaScript and XML (AJAX) and JavaScript Object Notation (JSON), JavaScript has become the de facto standard for creating interactive web applications. With its new found power and popularity, JavaScript has been the target of many attacks. In this paper, we present a framework that allows programmers to define secure properties of JavaScript objects such that they are more immune to malicious activity and require a smaller footprint that existing solutions. We then use our framework and apply it to an already built JavaScript system to analyze its properties and effectiveness.**

INTRODUCTION

JavaScript is an object-oriented scripting language that is primarily used in the form of client-side JavaScript, implemented as an integrated component of the web browser, allowing the development of enhanced user interfaces and dynamic websites. JavaScript is a dialect of the ECMAScript standard and is characterized as a dynamic, weakly typed, prototype-based language with first-class functions [1]. JavaScript is one of the world's most popular programming languages due primarily to the fact that almost every modern personal computer has a web browser with a JavaScript interpreter installed on it. JavaScript's popularity is due entirely to its role as the scripting language of the internet [2].

Given JavaScript's ubiquity and popularity, it has proven to be a common target for malicious activity. Modern web browsers provide a Document Object Model (DOM) API to JavaScript that allows script to interact with the browser and the pages that it renders to the user, but the DOM provides the potential for malicious authors to easily deliver code to clients. Malicious authors can easily deliver scripts that invoke DOM functionality such as redirecting the user's browser, possibly to a malicious web site. Most web browsers attempt to mitigate this risk using two restrictions. First, all scripts run in a sandbox environment where they only have access to the web browser and not the system itself. Second, scripts are constrained by the same origin policy, where loaded in one document do not have access to data and scripts loaded in another document. Most JavaScript-related security problems are breaches of one of the two restrictions [1]. In addition, insecure JavaScript engineering practices often open new attack vectors. In many cases, security does not receive sufficient attention due to the complexity of web based applications, the ad hoc process of development, and the fact that many web designers do not have the necessary security knowledge on web development techniques [3]. It comes as no surprise that website security breaches are common and web-based applications are more susceptible to attacks than other traditional applications [4].

One of the main vulnerabilities in web applications is that one document may load scripts from many different sources. Each script loaded into a particular document by default has access to all of the data and functionality of each of the other scripts loaded into that document. This is often necessary for activities such as the inclusion of advertising [5] and analytics tracking [6]. A study of 6,805 popular web sites in 15 different categories revealed that at least 66.4% of analyzed sites have insecure practices such as including scripts from external sources into the top-level documents of their homepages [3].

```
<script type="text/javascript">
var gaJsHost = (("https:" == document.location.protocol) ?
"https://ssl." : "http://www.");
document.write(unescape("%3Cscript src='" + gaJsHost + "google-
analytics.com/ga.js' type='text/javascript'%3E%3C/script%3E"));
</script>
```

FIGURE 1
A SINPPET OF JAVASCRIPT BASED ON WWW.CS.UIUC.EDU WHERE A SCRIPT FROM A DIFFERENT SOURCE IS LOADED INTO THE TOP LEVEL DOCUMENT OF THE SITE

Another common JavaScript vulnerability is its ability to dynamically generate and execute source code. The `eval()` function is provided by the JavaScript engine and evaluates a string containing JavaScript code as if it was part of the source loaded with the document. Malicious authors often use this ability to deliver code at runtime that adversely impacts the document. The same study mentioned above found that over 74.9% of analyzed web sites use one or more types of JavaScript dynamic generation techniques, and calls to the `eval()` function exist in 44.4% of analyzed web sites. Based on running several popular web sites and JavaScript tests through an instrumented JavaScript engine, a study found that changes to live objects are not uncommon, with adding fields to objects being the most frequent modification. While deletion of fields was not detected in the tests, it was found to occur on several web sites [7].

```
<script type="text/javascript">
eval("alert('Hello, world!");
</script>
```

FIGURE 2
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING THE USE OF THE EVAL() FUNCTION

To help programmers secure their scripts from attacks, we present the HotSausage JavaScript framework [8]. The HotSausage framework contains several modules to allow JavaScript programmers to richer sets of functionality than what the environment provides by default, such as privacy, collections, and templates. For this paper, we focus on the privacy module, which allows programmers to secure properties of objects in secure locations and only grant designated functions of objects access to those protected properties. By including the HotSausage framework in a document, an author has the ability to write code that is more immune to attack than it would be alone.

This paper will explore the functional abilities of the privacy module, how it interacts under various possible attacks, performance penalties incurred by using it, and the next steps that we are taking to enhance the framework. Lastly, we will investigate a case study where the HotSausage framework was applied to a working JavaScript system to demonstrate its abilities in a typical setting. We aim to show that by being conscientious of security when writing JavaScript source code, authors can mitigate many common attack vectors targeted at their applications.

MOTIVATION

Web browsers define the origin of a document to include the protocol used to load it and the domain name and port that it was loaded from. Although a document can only be loaded from one source, there is no such restriction on the origin of scripts that are included in a document. Under the same-origin policy, JavaScript cannot access another document that has a different origin, but it does have full access the document in which it was included even if its origin is different from its parent's document [9].

It is possible then for a document author to unknowingly include a JavaScript script in his document that has malicious intent and full access to all other data in the document. It is common practice for authors to include scripts for functionality such as libraries and frameworks in their document from external script hosting sites to reduce the burden of updating local copies when new versions of the script are released or to reduce load on their hosting servers. Providers such as Yahoo! offer scripts in this manner [10]. Such script hosting sites may become popular targets of attack, due to the ease of distributing malicious code to a large user base if the hosting site was compromised. There are alternative methods to mitigate this risk, but often times they are not compatible with the document author's design intentions or are overlooked [3].

In addition, the `eval()` function is especially dangerous. It takes a single string parameter and evaluates it as JavaScript source without reference to a particular object, but with the same privileges as the function's caller [11]. If the `eval()` function is present in a script, an attacker could try to leverage its power to evaluate a string of malicious code with the privileges of the caller. In addition, since often times `eval()` is used to run dynamically generated scripts, it is near impossible to effectively filter out malicious code from valid code [3].

We aim to reduce the chances that a malicious script could alter data of JavaScript objects that have strict security and integrity requirements. By utilizing our framework, authors can be confident that their data will remain secure and immune to attack.

HOTSAUSAGE

The HotSausage JavaScript framework aims to provide JavaScript programmers with a rich set of functionality that is not provided by the standard library. The base HotSausage module contains general settings for the framework along with setup and structure for all the sub-modules it contains. One example setting is the ability to handle errors quietly. If enabled, the framework will not throw errors on error conditions but rather silently ignore them. The framework is further decomposed into several sub-modules, each with its own distinct purpose.

The collections module provides authors with a set of data structures that can be used to store collections of data. It features predefined objects such as lists and spans of elements that feature rich APIs similar to what a programmer would expect from other languages, such as Java or C#.

The purpose of the Templates module is to enable programmers to program using pure prototypal semantics. While JavaScript inherently prototypal, its inconsistent object creation syntax obscures its prototypal basis, leading many programs to try to write programs from a classical class-based approach. Furthermore, this schizophrenic nature around object creation creates usability issues for programmers, which causes some common and serious bugs which are tricky to debug and are visually difficult to locate in source code. Using the HotSausage Templates module enables programmers to avoid these headaches, and use a pure and consistent approach to object creation and build object types.

The final module in the HotSausage framework is the privacy module. Its responsibility is to provide objects with the capability to store protected properties in a secure "purse" that is only accessible by author designated functions. Accesses to the purse of an object are secured with a one-time session key that is only valid when invoked in the scope of the framework. In addition, there are several layers of checks that are performed when accessing protected properties that ensure that only authorized access is allowed.

I. Getting Started with HotSausage

To use the HotSausage JavaScript framework, script authors must first include the compiled source file containing the framework. Currently, we only support loading the framework from the same host as the origin of the document where it will be used, but in the future we plan to implement a script hosting service for the code. To mitigate risk of malicious authors compromising the script host, we plan to distribute hash checksums of the code to authors and include a built-in hashing mechanism in the framework that will authenticate itself when it is loaded.

As soon as the source code is loaded in the document, the framework automatically registers itself as a global object available to all other scripts called `HotSausage`. All sub-modules of HotSausage, such as collections, templates, and privacy automatically register themselves as `HotSausage.Collections`, `HotSausage.Templates`, and `HotSausage.Privacy` respectively. It is then the responsibility of the document author to call the `HotSausage.installCoreMethods()` function to finish initializing the framework. This function augments several of the JavaScript global objects, such as Object and Function to include extra functionality that the framework provides. It accomplishes this task by adding fields to the prototypes of these objects.

```
<script type="text/javascript">
HotSausage.installCoreMethods();
</script>
```

FIGURE 3
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING THE INITIALIZATION OF THE FRAMEWORK

HOTSAUSAGE PRIVACY MODULE

The privacy module provides authors the ability to secure protected properties of JavaScript objects in secure containers. Once the HotSausage framework is loaded into a document and is initialized, all of its features are available to script authors, including privacy, but privacy features are not extended to any objects unless an author explicitly enables them for a type of object. This requirement reduces overhead since only objects requiring privacy features will require the extra processing demanded by HotSausage when they are instantiated and evaluated.

II. Getting Started With the Privacy Module

When an object has privacy featured enabled, a container that we call a purse is attached to it, which is used to store properties of the object that authors wish to protect. The purse is only available in the functional block in which privacy was enabled and in any functions that the author designates as privileged. We made a design decision to reveal a direct reference to the purse when privacy is enabled hoping that script authors will have the foresight not to leave access to that reference available outside of the constructor of an object.

```
<script type="text/javascript">
var Person = function (ssn) {
     var purse = this.enablePrivacy();
     purse.ssn = ssn;
     //it is the author's responsibility to not disclose the
     //the reference to the purse here
};
</script>
```

FIGURE 4
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING HOW TO ENABLE PRIVACY ON AN OBJECT AND STORE PROPERTIES IN ITS PURSE

As seen in (4) and steps p1-p2 in (19), once privacy is enabled, the programmer gets direct access to the purse for that object where he can store whatever properties he would like. The reference to the purse is only valid inside of the functional block in which privacy is enabled, which in this case is the constructor of the object. The security and integrity of the privacy for the `Person` object would be compromised if the author assigned the purse reference to a variable declared in a different scope than the constructor or defined a function that returned the purse reference inside the constructor. As stated earlier, we will count on script authors to be mindful of these restrictions and write their scripts accordingly. We also to plan to include these restrictions in the documentation that is shipped with the framework and in the wiki for the project to raise awareness of this potential attack vector to users of the framework.

The reference to the purse that is received when calling `enablePrivacy()` is the only direct reference that objects will have to their purses. Any other accesses to the purse can only come from functions which the script author as designated as privileged.

Lastly, after `enablePrivacy()` is called, a new function called `_purse()` is added to that object as seen in p5 in (19). This function takes a session key and if the key provided is correct, stores the purse for that object in a location that is only accessible by the framework. The use of this function will be described below.

III. Adding Privileged Methods to Objects

When `installCoreMethods()` is called to initialize the framework, several methods are added to global JavaScript objects. The two methods that the privacy module installs are `Object.prototype.privilegedMethod` and `Function.prototype.privilegedMethod`. Both of these functions have the same purpose, but act slightly different. Since `Function` inherits from `Object`, objects of type `Function` will inherit `Function`'s version of `privilegedMethod` while all other JavaScript objects will inherit `Object`'s version.

Both of these functions serve the same purpose, to install a function on the object on which it was called that has access to the purse of that object. `Function`'s version of `privilegedMethod` adds a privileged method to the prototype of the object on which it was called, while Object's version adds the privileged method to the object itself that it was called on.

The function privilegedMethod takes two parameters, a string containing the name that the author wishes the new function to have and a function containing the implementation of the new method as seen m1 in (19). When writing the implementation of the new privileged method, the author must make a special consideration. The reference to this in the implementation will refer not to the object that the method is being added to, but rather to the purse of that object. By default, the purse is populated with two properties:

- owner – a reference to the object which owns the purse
- _hspv – which is reserved for future use by the framework

Any other properties that the programmer added to the purse at any other time are also available.

```
<script type="text/javascript">
Peron.privilegedMethod(“getSSN”, function () {
    return this.ssn;
});
var personImpl = new Person(“123-45-6789”);
personImpl.getSSN(); //returns “123-45-6789”
</script>
```

FIGURE 5
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING HOW TO PROPERLY ADD A PRIVILEGED METHOD TO AN OBJECT

IV. Locking the Framework

After the script author has defined all of the objects that are required for his script, it is necessary to lock the framework to prevent an attacker from simply adding a new privileged method to an object at run time and accessing an object's purse. To accomplish this task, HotSausage provides a lock() function that prevents this from happening. By calling HotSausage.lock(), the author prevents any further addition of privileged methods to objects, but the functions to enable privacy on an object remain so that objects constructed after the framework was locked will still succeed.

```
<script type="text/javascript">
HotSausage.lock();
</script>
```

FIGURE 6
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING HOW TO PROPERLY LOCK THE FRAMEWORK

After HotSausage.lock() is called, the properties Object.prototype.privilegedMethod and Function.prototype.privilegedMethod are deleted. Any attempts to add them back by an attacker will not have access to any object's purse because the new functions will not be defined within the scope of the framework, which is necessary to access purses. A potential problem with our approach is that if an attacker does in fact redefine these functions, they will appear to work properly for the script author with no warning given, but the worst possible outcome is that the functions that are added to the object requested will not work as expected; there will be no accesses to the purse of that object.

V. Privileged Method Internals

When an author calls the privilegedMethod function, the framework does much more than simply adding the implementation function as a property of the object it was called on.

First, the implementation function is wrapped in another function that gains access to the purse of the object to which the method is being added as seen in m2 and m3 in (19). The details of this access will be described below. After the wrapper function has successfully gained access to a reference to the purse, it uses JavaScript's Function.prototype.apply method to execute the implementation function, passing the reference to the purse as the context to which the implementation function is evaluated in as seen in m4 in (19). This is how the reference in the implementation function references the purse and not the behavior object to which the implementation function belongs. The result of evaluating the implementation function is stored in the wrapper and is compared to the purse. If the result of the implementation function is not the purse, that result is returned, otherwise the purse's owner is returned. This last check mitigates the attack where a malicious author could write a privileged method using the framework to try to expose a reference to an object's purse by returning it.

A current limitation is that an attacker could create an additional privileged method that assigns its own reference to the purse to a variable declared in a larger scope. We currently have no defense against this type of attack, so it is imperative that programmers write their privileged method implementations carefully and lock the framework once they have finished.

VI. Purse Access Internals

As mentioned earlier, when adding privileged methods to objects, the framework wraps their implementation with another function that is responsible for mediating the execution of the implementation function, including gaining it a reference to the purse of the object that it is attached to.

The framework internally has a function defined called __purseOf() which takes a single parameter, an object, and returns the purse of that object as seen in (19). The wrapper function mentioned above calls this function to retrieve a reference to an object's purse. It does this by first generating a one-time session key. This key is only visible within the framework and is only valid for the time in which __purseOf() is executing. Inside of the privacy module, there exists an object that is simply a holder for properties called the _activeTransporter as seen in (19) labeled aHashMap. The __purseOf() function stores a dummy reference in the slot of the `_activeTransporter` designated by the generated key, then calls the object's `_purse()` method, supplying it the session key as seen in e4 in (19). The `_purse()` function takes the reference to the purse of its object and stores it in the `_activeTransporter` in the slot where the dummy reference was as seen in e5 in (19).

If an attacker was to call the `_purse()` function with an invalid session key, all that would happen is that the `_purse()` function would store a reference to the purse inside the framework's `_activeTransporter` where it will never be read.

Once the `_purse()` function terminates, the `_purseOf()` function retrieves the reference to the purse from the `_activeTransporter` and returns it back to the privileged method's wrapper function as seen in e6 in (19).

VII. Privacy Module Security Checks

Earlier, we mentioned that the HotSausage framework made several security checks during execution and threw errors if the checks resulted in a condition that violated the security policy. A programmer can write JavaScript code that monitors for these errors and take action depending on which errors are thrown. Below we will describe several potential attacks against the framework and how the framework responds.

Given that an attacker knew that there were two types of objects, both with privacy enabled and both with a purse property of the same name. If the attacker knew that object A had a privileged method that returned the protected property of the name he was looking for in object B, but object B did not have a privileged method that revealed that property, he might try to reassign the privileged method from object A to object B.

```
<script type="text/javascript">
var Employee1 = function (ssn) {
     var purse = this.enablePrivacy();
     purse.ssn = ssn;
};

Employee1.privilegedMethod(“getLastFourOfSSN”, function () {
     return this.ssn.substring(this.ssn.length-4, 4);
});

var Employee2 = function (ssn) {
     var purse = this.enablePrivacy();
     purse.ssn = ssn;
};

HotSausage.lock();

var e1 = new Employee1(“123-45-6789”);
var e2 = new Employee2(“987-65-4321”);

var e1ssn = e1.getLastFourOfSSN() //returns “6789”

e2.prototype.getLastFourSSN = e1.prototype.getLastFourOfSSN
var e2ssn = e2.getLastFourOfSSN(); //throws error
</script>
```

FIGURE 7
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING THE PRIVILEGED METHOD REASSIGNING ATTACK

In this case, the framework would cause a JavaScript error to be thrown that would indicate that the privileged method had been moved. This check is performed inside the function that wraps the privileged method's implementation, and compares the object that it was originally attached to to the current value of `this`.

Another attack exists where two objects both have privacy enabled on them and with private properties of the same name, but one object has privileged methods to access all of its private properties which the second object does not. An attacker may try to reassign the _purse() function of the inaccessible object to the accessible object so that accessing privileged methods of the accessible object would actually reference the purse of the inaccessible object.

```
<script type="text/javascript">
var Employee1 = function (ssn) {
     var purse = this.enablePrivacy();
     purse.ssn = ssn;
};

Employee1.privilegedMethod(“getLastFourOfSSN”, function () {
     return this.ssn.substring(this.ssn.length-4, 4);
});

var Employee2 = function (ssn) {
     var purse = this.enablePrivacy();
     purse.ssn = ssn;
};

HotSausage.lock();

var e1 = new Employee1(“123-45-6789”);
var e2 = new Employee2(“987-65-4321”);

var e1._purse = e2._purse;

var e1ssn = e1.getLastFourOfSSN(); //throws error
</script>
```

FIGURE 8
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING THE PURSE ACCESSOR METHOD REASSIGNING ATTACK

This attack will cause an error to be thrown because the `__purseOf()` function mentioned earlier will successfully execute the `_purse()` function that was transplanted, but it then checks that the `owner` property of the retrieved purse matches the object on which the privileged method was called. When that check fails, an error will be thrown.

Third, an attacker might try to redefine the `_purse()` function of an object that had security enabled on it to either try to reveal the purse or break the object. Although the purse will still be secure if the `_purse()` function is redefined improperly, privileged methods will no longer function properly as the actual purse will not be accessible. The framework will detect that the invalid `_purse()` function did not properly modify the `_activeTransporter` object since it did not have access to it, so a counterfeit `_purse()` function must have existed and an error will be throw.

```
<script type="text/javascript">
var Employee = function (ssn) {
     var purse = this.enablePrivacy();
     purse.ssn = ssn;
};

Employee.privilegedMethod(“getLastFourOfSSN”, function () {
     return this.ssn.substring(this.ssn.length-4, 4);
});

HotSausage.lock();
```

```
var e1 = new Employee(“123-45-6789”);

//attacker code
e1._purse = function (key) {
     var purse = {};
     purse.ssn = “987-65-4321”;
     return purse;
};

e1.getLastFourOfSSN(); //throws error
</script>
```

FIGURE 9
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING THE PURSE ACCESSOR METHOD REDEFINING ATTACK

Fourth, a malicious author might try to call the `_purse()` function of an object with privacy enabled many times to guess the session key. Although guessing the correct session key while that key is alive will only give the framework access to the purse, it may give the attacker the ability to guess the next keys that are generated. A valid `_purse()` function will detect invalid session keys and cause an error to be thrown if errors are not set to be handled quietly.

```
<script type="text/javascript">
var Employee = function (ssn) {
     var purse = this.enablePrivacy();
     purse.ssn = ssn;
};

Employee.privilegedMethod(“getLastFourOfSSN”, function () {
     return this.ssn.substring(this.ssn.length-4, 4);
});

HotSausage.lock();

var e1 = new Employee(“123-45-6789”);

//attacker code
e1._purse(Math.random()); //throws error
</script>
```

FIGURE 10
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING THE BRUTE FORCE SESSION KEY ATTACK

Lastly, an attacker might try to access the purse of an object by trying to call `enablePrivacy()` on that object again. The framework will detect the duplicate call and will throw an error.

```
<script type="text/javascript">
var Employee = function (ssn) {
     var purse = this.enablePrivacy();
     purse.ssn = ssn;
};

HotSausage.lock();

var e1 = new Employee(“123-45-6789”);

//attacker code
var purse = HotSausage.Privacy.enableOn(e1); //throws error
</script>
```

FIGURE 11
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING THE DUPLICATE PRIVACY ENABLEMENT ATTACK

VIII. Privacy Module Vulnerabilities

Earlier we discussed several weaknesses of the privacy module. In this section, we will expand on those weaknesses and demonstrate possible attack vectors that exploit them.

The primary weakness of the framework is its dependence on script authors to be security conscientious when writing their scripts with HotSausage. We rely on them to ensure that any references to purses are not leaked to areas of code where they could potentially be exploited by an attacker.

```
<script type="text/javascript">
var currentPurse;

var Employee1 = function (ssn) {
     currentPurse = this.enablePrivacy();
     purse.ssn = ssn;
};

var Employee2 = function (ssn) {
     var purse = this.enablePrivacy();

     this.getPurse = function () {
          return purse;
     };
};

HotSausage.lock();

var e1 = new Employee(“123-45-6789”);
var e2 = new Employee2(“987-65-4321”);

</script>
```

FIGURE 12
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING A LEAK OF PRIVATE DATA

In both of the examples given in the previous figure, an attacker would have access to the private `ssn` property of both objects. In the case of e1, the attacker would be able to read `currentPurse.ssn` and in the case of `e2`, the attacker could read `e2.getPurse().ssn`. By being prudent of the limitations of the framework, an author can eliminate any attacks of this nature.

Another vulnerability exists in the reassigning of a purse to an object after it is constructed. When `enablePrivacy()` or `HotSausage.Privacy.enableOn()` is used to attach a purse to an object, the framework first checks to ensure that privacy has not already been enabled on the target object as discussed earlier. Internally, the framework does this by checking for the existence of the `_purse()` method on the target. If an attacker removes the `_purse()` method from the target and enables privacy on that object again, he then has full access to a blank purse where he can put whatever properties he would like. The owning object has no knowledge of this attack.

```
<script type="text/javascript">
//valid code
var Employee = function (ssn) {
     var purse = this.enablePrivacy();
     purse.ssn = ssn;
};
Employee.privilegedMethod("getLastFourOfSSN", function () {
     return this.ssn.substring(this.ssn.length-4, 4);
});
HotSausage.lock();
var e1 = new Employee("123-45-6789");

//attacker code
delete e1._purse;
var newPurse = HotSausage.Privacy.enableOn(e1);
newPurse.ssn = "ATTACKEDSSN";
e1.getLastFourOfSSN(); //returns "DSSN"
</script>
```

FIGURE 13
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING THE PURSE ACCESS ATTACK

We are currently implementing a solution to mitigate this attack, which will require a script author to register his objects with the framework and be supplied a unique key. When enabling privacy on those registered objects, the proper key will need to be supplied back to the framework in order for the operation to complete successfully. This strategy still is vulnerable to the primary weakness of the framework, information leaks due to author negligence. If a script author accidently makes the key for an object available, the above mentioned attack is still possible.

HOTSAUSAGE PERFORMANCE

The HotSausage framework necessarily creates overhead on objects that have privacy enabled on them. This is due to the extra function calls and data initializations required for ensuring privacy and secure generation of session keys. We created a suite of benchmarks that test the privacy module compared with several other traditional methods that have been used to protect private properties in JavaScript objects.

I. Traditional Privacy Mechanisms

There are two main ways which JavaScript programmers traditionally secure private properties of objects, closure based protection and naming convention based protection.

Closure based protection revolves around the concept of functional scope in JavaScript. Properties declared in JavaScript are only readable within the functional scope that they are declared, including in any functional blocks declared within that scope. Programmers use this property to declare variables in the constructor of an object, and then define functions within that constructor that access those variables. Since the defined functions are in the functional scope of the constructor, they have access to any variables declared in the constructor. This effectively ensures privacy for those variables, but incurs a high overhead for each object since functions declared in the constructor of an object are bound to each instance of the object created rather than to the prototype of the object itself.

```
<script type="text/javascript">

var Employee = function (ssn) {
     this.getLastFourOfSSN = function () {
          return ssn.substring(this.ssn.length-4, 4);
     );
};

</script>
```

FIGURE 14
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING CLOSURE BASED PRIVACY

As seen in the above figure, an attacker would have only the allowed read access and no write access to the variable `ssn`.

A second traditional way of securing private properties is by using a naming convention that conveys that properties are supposed to be private. Although this is a widely used technique, it is insecure since properties that are named as private are still accessible to an attacker. Commonly, JavaScript programmers will use an underscore before the name of a property that they wish to designate as private. This method has the benefit of incurring a low overhead since functions of objects that need to access private properties can be bound to that object's prototype rather than to the object itself.

```
<script type="text/javascript">

var Employee = function (ssn) {
     this._ssn = ssn;
};

Employee.prototype.getLastFourOfSSN = function () {
     return this._ssn.substring(this._ssn.length-4, 4);
);

</script>
```

FIGURE 15
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING NAMING CONVENTION BASED PRIVACY

II. Benchmarking Procedure

We wrote a series of benchmarks that test the privacy module against traditional techniques both for object construction and private property access. We measured the processing time required for many iterations of both instances. The benchmarks were run using the Rhino JavaScript engine using a variety of optimization levels [12, 13]. The Rhino engine was chosen to eliminate as many variables as possible that browser-based engines may introduce and allow for tuning the engine optimization settings.

III. Benchmark Results

For each of the tests, 1,000 iterations of 10,000 object constructions and property accesses were performed. The times listed below reflect the mean and standard deviation of the processing time for all of the 1,000 iterations. The benchmarks were run on an Intel Core 2 Duo T9600 at 2.80 GHz with 4 GB of main memory.

TABLE I
BENCHMARK RESULTS FOR OBJECT CONSTRUCTION

Engine Optimization Level	Privacy Technique	Mean (ms)	Standard Deviation (ms)
Interpreted	HotSausage	81.50	17.80
Interpreted	Closure	43.82	5.25
Interpreted	Naming Convention	22.65	0.87
None	HotSausage	32.45	17.91
None	Closure	23.69	2.68
None	Naming Convention	8.16	1.47
Full	HotSausage	33.60	19.93
Full	Closure	22.91	1.81
Full	Naming Convention	7.95	1.39

TABLE 2
BENCHMARK RESULTS FOR PROTECTED PROPERTY ACCESS

Engine Optimization Level	Privacy Technique	Mean (ms)	Standard Deviation (ms)
Interpreted	HotSausage	144.61	53.82
Interpreted	Closure	13.55	1.19
Interpreted	Naming Convention	14.61	0.75
None	HotSausage	59.42	52.00
None	Closure	3.55	26.98
None	Naming Convention	7.55	1.39
Full	HotSausage	61.49	49.47
Full	Closure	3.59	26.98
Full	Naming Convention	3.09	0.89

As can be seen by the above results, HotSausage incurs a severe performance penalty for both object construction and protected property access versus traditional methods. Even given these penalties, we still believe that HotSausage is a better solution than the traditional techniques. An object with HotSausage privacy enabled on it has a smaller memory footprint than a comparable object with closure based privacy and is more secure than a comparable object using naming convention based privacy.

It is important to note that these processing times are for many iterations of object construction and private property access, and when performing each of these tasks a small number of times, the differences in the compute times required are not measurable.

TESTING

To ensure that our framework performs as expected all throughout development, we employed an extensive testing strategy based upon the behavior driven development philosophy. Behavior driven development focuses on using natural language to describe the behavior of units of code rather than the technical details that power them. It minimizes translation between the technical language in which the code is written and the domain language spoken by the end users [14].

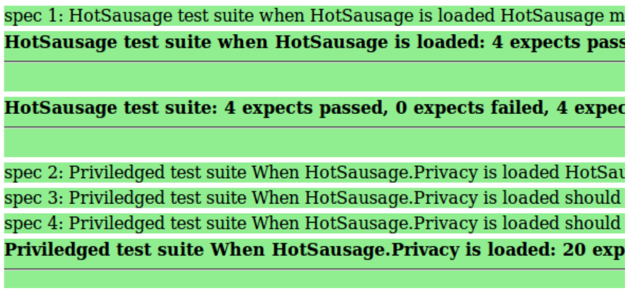
I. Testing Framework

To aid us in our testing, we employed an existing behavior driven development testing framework called Jasmine [15]. Jasmine allowed us to cleanly express the behavior of our units of code and easily write expressive tests to ensure that each unit behaved as expected. In addition, Jasmine allowed us to write a custom renderer to render the testing output, so we were able to more easily determine test results and locate problems when tests failed.

```
<script type="text/javascript">

describe('Priviledged test suite', function () {
     describe('When HotSausage.Privacy is loaded', function () {
          it('should be able to add a privileged method to an
object', function () {
               var pp;

               var Person = function (ssn) {
                    var purse = this.enablePrivacy();
                    purse.ssn = ssn;

                    //bad practice, but needed for testing
                    pp = this;
               };

               Person.privilegedMethod("getSSN", function () {
                    expect(this).toEqual(pp);                              expect
                    return this.ssn;
               });

               var ssn = "123-45-6789";
               var p = new Person(ssn);

               expect(p.getSSN).toBeDefined();
               expect(p.getSSN()).toEqual(ssn);
          });
     });
});

</script>
```

FIGURE 16
A SINPPET OF JAVASCRIPT SOURCE DEMONSTRATING AN EXAMPLE JASMINE TEST

We wrote extensive test suites for all modules of the framework, including privacy. We used these test suites to ensure that the framework functioned as we intended. We ran the test suite after any major edits to the source code to verify that the changes that we made did not adversely impact the functionality of the framework. If a test happened to fail, we investigated the cause of the failure and implemented a fix before moving on to the next step in development.



```
spec 1: HotSausage test suite when HotSausage is loaded HotSausage m
HotSausage test suite when HotSausage is loaded: 4 expects pass

HotSausage test suite: 4 expects passed, 0 expects failed, 4 expec

spec 2: Priviledged test suite When HotSausage.Privacy is loaded HotSau
spec 3: Priviledged test suite When HotSausage.Privacy is loaded should
spec 4: Priviledged test suite When HotSausage.Privacy is loaded should
Priviledged test suite When HotSausage.Privacy is loaded: 20 exp
```

FIGURE 17
OUR CUSTOM JASMINE TEST OUTPUT

HOTSAUSAGE CASE STUDY

To demonstrate our framework in action, we applied it to an already existing JavaScript application. This case study of the privacy module allowed us to verify the practicality of our work and see its overhead in a real world scenario.

As part of our research, we also work on the Medical Device Plug and Play (MDPnP) project [16]. The Medical

Device Plug and Play project aims to improve patient safety by developing standards for the safe operation and communication between medical devices. We gathered all of the requirements of the project and developed a mock system that demonstrated the possibilities of having medical devices function in coordination with each other. From initial design to final testing, we utilized a variety of technologies, including JavaScript, to implement a successful mock environment that integrated actual hardware in several scenarios to demonstrate the potential of the system [17].

The system is composed using mostly JavaScript based software models, but also contains hardware interfaces written in Java and a user interface written with JavaScript, HTML, and Flash [18, 10]. Other tools, such as XML and JSON are also used as communications carriers for data between components of the system.

I. Framework Application

Specifically, we incorporated the HotSausage framework in the modules of our system that handled the creation and storage of patient vital sign data. In our system, sensor devices send signals containing data payloads through the network to be stored in a repository of data. By modifying the output of each sensor device and the accessor methods of the storage component, we were able to easily integrate the privacy features of HotSausage into the system. We chose the vital sign creation and storage code to test our framework since its data should by nature be secure, immutable, and private.

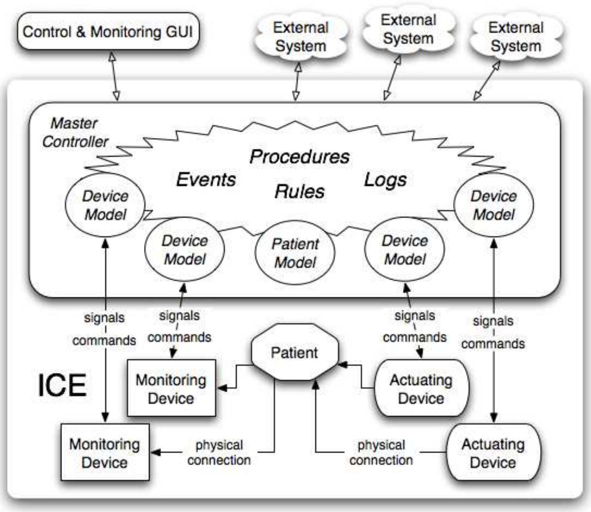


FIGURE 18
AN OVERVIEW OF THE ARCHITECTURE OF OUR MEDICAL DEVICE SIMULATION

Applying the framework to our system was a straightforward process requiring minimal rewriting of existing code. The only major changes that were required were in the declaration of functions that had access to private data as well as enabling privacy on vital sign objects as they were created by sensor devices.

II. Results and Analysis

After HotSausage was applied to our MDPnP system, we were able to analyze the effectiveness, performance, and usability of the modified system.

The framework behaved exactly as expected when applied to an actual application. Since the application was previously using closure-based privacy features, the conversion to HotSausage-based privacy had no change on the interface of the objects or on their functionality. We noted that the privacy of the data stored in each object was still subject to the same effective privacy that it had earlier, but without the overhead that is attached to closure-based privacy.

Although we did not instrument the MDPnP system before and after HotSausage application to get firm data, we did not notice any major performance impacts on the operation of the system. We decided not to instrument the system because the number of vital sign objects that are created and accessed per second in the system is relatively low (<100), and based on previous benchmarks of the framework, this low number of operations is not measureable using the finite clocks that JavaScript provides. The bottlenecks in the application remained the same as before the application of the framework, mostly in the interactive components of the user interface.

III. Conclusions

The framework behaved exactly as expected when applied to a real-world JavaScript application. All functionality that we had guaranteed through unit testing worked as expected. At first, we were wary that the performance penalties that we noted during our benchmarking would be a hindrance in real-world use, but that proved not to be the case. As mentioned earlier, the number of instantiations of privacy-enabled objects and accesses to private data were too small to be measureable by the techniques that we had at our disposal and no human-noticeable difference in responsiveness was detected. This small case study provided evidence that our framework is practical, usable, and provides the security features that it intends.

CONCLUSIONS AND FURTHER WORK

Although HotSausage is not yet production ready, its advantages are already apparent. While the collections and templates modules are not quite mature, the privacy module is almost ready for deployment into actual applications.

Our analysis of the privacy module has led us to identify several weaknesses and vulnerabilities, but we feel that we have addressed them to the best of our abilities. We have checks in place already to prevent the majority of vulnerabilities found, and have an implementation in progress for the attack that was discovered during our analysis. Although we have also discovered several vulnerabilities that cannot be addressed with our current solution, such as programmer negligence, we hope that by providing adequate documentation we will reduce or eliminate any attacks that take advantage of this weakness in our framework.

The most significant obstacle that we encountered during our implementation and analysis was performance. The additional processing overhead introduced by HotSausage is quite significant, although we contend that this additional work required is worth the functionality that the framework provides. Its advantages over both traditional types of JavaScript privacy are considerable and should be taken into account when authors choose which type of privacy strategy to employ in their applications. Our case study on our medical device simulation proved that our framework not only functions as it was designed to, but also is practical in real world applications.

In the future, we plan to continue to develop HotSausage and release it as open source software for inclusion in real world applications. The majority of work in the privacy module will be directed towards addressing the vulnerability discussed earlier and reducing the processing footprint of enabling privacy on an object and accessing private properties of an object. Lastly, we are considering aligning our codebase with the standards of CommonJS to facilitate its use as a library in both client and server-side applications [19]. We hope that by completing and releasing this framework to the community, we will be able to allow developers to write safer, more tamper resistant code with minimal effort.

ACKNOWLEDGEMENT

We would like to thank Steven Moser, Yun Young Lee, Nicholas Chen, Ralph Johnson, and the MDPnP project group – with special thanks to Mu Sun, Cheolgi Kim, Lui Sha, and Mary Flesner.

REFERENCES

[1] JavaScript. http://en.wikipedia.org/wiki/JavaScript

[2] JavaScript: The World's Most Misunderstood Programming Language. D. Crockford. http://www.crockford.com/javascript/javascript.html

[3] C. Ye and H. Wang. "Characterizing Insecure JavaScript Practices on the Web", In *Proc. of the WWW 2009*, pages 961-970, 2009

[4] W. S. (Editor). "Web Engineering: Principles and Techniques" IGI Publishing, ISBN 1-591-40433-9, 2005

[5] Google AdSense. https://www.google.com/adsense/

[6] Google Analytics. http://www.google.com/analytics/

[7] S. Lebresne *et al.* "Understanding the Dynamics of JavaScript" In *Proc. for the 1st workshop on Script to Program Evolution*, pages 30-33, 2009

[8] m3rabb / HotSausage / Overview – bitbucket.org. http://bitbucket.org/m3rabb/hotsausage/

[9] D. Flanagan. "JavaScript: The Definitive Guide" O'Reilly Media, ISBN 0-596-10199-6, 2006

[10] YUI Library. http://developer.yahoo.com/yui/

[11] eval – MDC. https://developer.mozilla.org/En/Core_JavaScript_1.5_Reference/Global_Functions/Eval

[12] Rhino – JavaScript for Java. http://www.mozilla.org/rhino/

[13] Rhino optimization – MDC. https://developer.mozilla.org/en/Rhino_Optimization

[14] Behavior Driven Development – Wikipedia. http://en.wikipedia.org/wiki/Behavior_Driven_Development

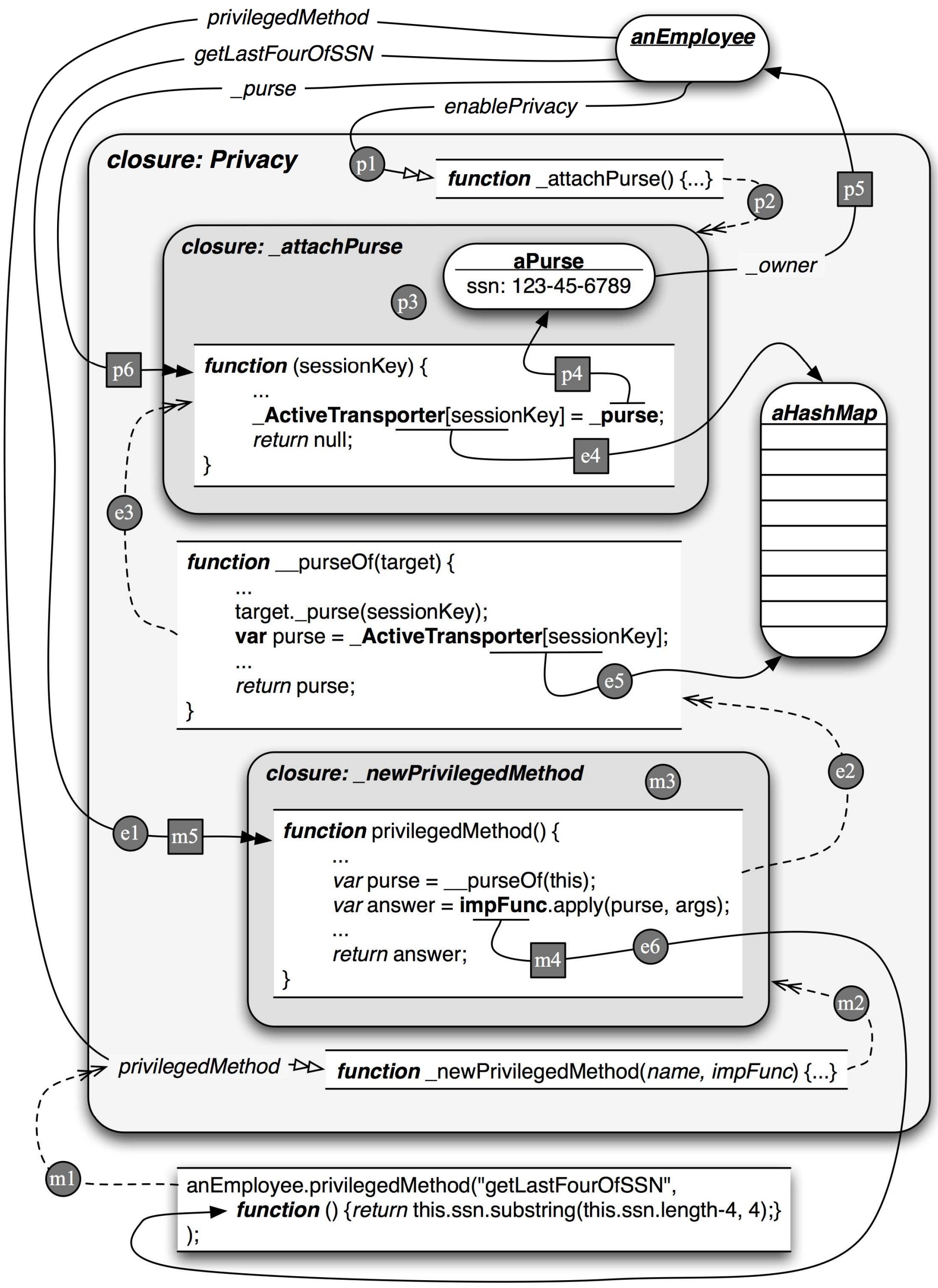
[15] pivotal's jasmine at master – GitHub. http://github.com/pivotal/jasmine

[16] MDPnP. https://agora.cs.illinois.edu/display/mdpnp/Home

[17] C. Meyer. "Mocking an Integrated Clinical Environment with JavaScript", 2009

[18] Adobe – Flash Player. http://www.adobe.com/software/flash/about/

[19] CommonJS: JavaScript Standard Library. http://commonjs.org/



```
function _attachPurse() {...}
```

```
function (sessionKey) {
    ...
    _ActiveTransporter[sessionKey] = _purse;
    return null;
}
```

```
function __purseOf(target) {
    ...
    target._purse(sessionKey);
    var purse = _ActiveTransporter[sessionKey];
    ...
    return purse;
}
```

```
function privilegedMethod() {
    ...
    var purse = __purseOf(this);
    var answer = impFunc.apply(purse, args);
    ...
    return answer;
}
```

```
function _newPrivilegedMethod(name, impFunc) {...}
```

```
anEmployee.privilegedMethod("getLastFourOfSSN",
    function () {return this.ssn.substring(this.ssn.length-4, 4);}
);
```

FIGURE 19
THE LIFECYCLE OF THE PURSE AND PRIVILEGED METHODS

SINGLE BLACK ARROWHEADS REPRESENT DIRECT RELATIONSHIPS
DOUBLE BLACK ARROWHEADS REPRESENT DIRECT REFERENCES TO FUNCTIONS TO EXECUTE
DOUBLE WHITE ARROWHEADS REPRESENT REFERENCES TO FUNCTIONS VIA DELEGATE OBJECTS
DOUBLE OPEN ARROWHEADS WITH DASHED LINES REPRESENT A FLOW OF EXECUTION

ROUND LABELS REPRESENT A STEP IN A SEQUENCE OF STEPS
SQUARE LABELS REPRESENT ASSIGNMENTS AS PART OF A SEQUENCE OF STEPS

STEPS P1 THROUGH P6 REFER TO THE PROCESS OF ENABLING PRIVACY AND ATTACHING A PURSE TO AN OBJECT
STEPS M1 THROUGH M5 REFER TO PROCESS OF ADDING A PRIVILEGED METHOD TO AN OBJECT
STEPS E1 THROUGH E5 REFER TO THE PROCESS OF EXECUTING A PRIVILEGED METHOD