



# DOMinator Control Flow

Project: DOMinator Control Flow  
Date, version: 03-05-2011, 1  
Classification: Public

# Summary

<b>INTRODUCTION.....</b>	<b>3</b>
<b>Internals .....</b>	<b>3</b>
1.1 SpiderMonkey and Firefox Core Modification.....	3
1.2 Extension Components.....	4
1.3 Internals Workflow .....	6
1.4 DOMinator GUI .....	7
<b>Known Issues.....</b>	<b>10</b>

## Introduction

DOMinator is a Firefox based software for analysis and identification of DOM Cross Site Scripting issues using dynamic runtime tainting model on strings.

Its workflow can be divided into two main modules:

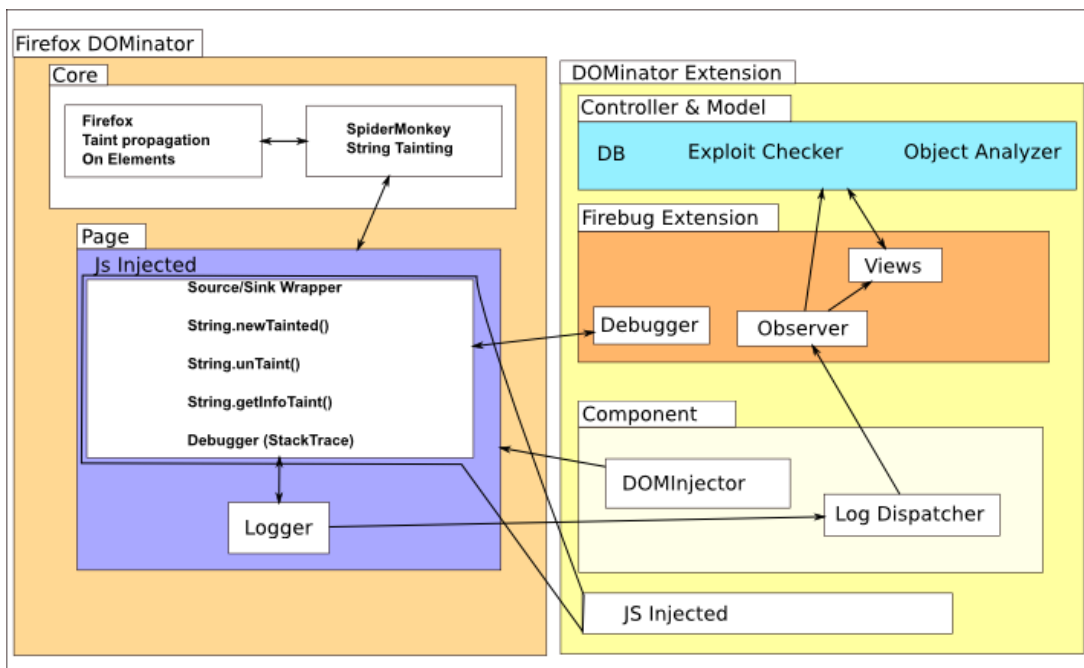
- the first one (Firefox DOMinator) concerning the implementation of tainting propagation made by modifying JavaScript interpreter SpiderMonkey and some parts regarding DOM objects implemented in Firefox kernel.
- the second one (DOMinator Extension), which is a Firefox and Firebug extension that performs log analysis, identifies criticalities and visualizes them by exploitability categories.

In the next chapters it'll be described the workflow and some of the technical details regarding the functioning of DOMinator.

## Internals

As shown in Figure 1, DOMinator is composed by two macro blocks.

1. Firefox DOMinator: implemented in C/C++ modifying strategical spots in Spidermonkey and Firefox in order to create a String Taint Propagation model to be exposed in HTML pages via JavaScript DOM.
2. DOMinator Extension: written in JavaScript, and deploying several modules concerning DOM Javascript injection of custom code, logging, analysis and exploitability level identification.



### 1.1 SpiderMonkey and Firefox Core Modification

Spidermonkey modifications regard:

- implementation of a tainting flag for JavaScript native Strings
- exposing of static String JavaScript methods for the flag (dis)activation on specific strings and a new object attribute
  - `String.newTainted(aString, "SourceName");`
  - `String.unTaint(aString);`
  - `aString.tainted;`
- tainting propagation on native JavaScript operations (replace, Regexp, concatenation etc.. )
- storing of specific operations in a data structure used for taint propagation back trace (from a tainted modified string to the tainted original string).
- exposing of a method returning the taint operations back trace in form of a JavaScript object.
  - `String.getInfoTaint(aString);`

Firefox code modifications regard:

- Location object overwriting by default
- implementation of taint propagation on DOM objects not implemented by SpiderMonkey (Eg. `HTMLAnchorElement.setAttribute("name", TaintedString)` afterward used for `document.write(HTMLAnchorElement.name)` )
- modification of String C++ `NSAString` class and its subclasses or similar for taint flag management flag and JS native string storage (for taint propagation).

## 1.2 Extension Components

Refer to previous figure,

- **./chrome/content:**
  - **domidebugger.js:** for stack trace generation.
  - **DOMIntruderFB.js:** Firebug extension (Views and Model). Implements the log observer, control and communication management with exploitation analyzers and the DB. Finally it performs dispatching on views.
  - **DOMIntruderFB.xul:** Firebug overlay.
  - **exploitChecker.js:** Flow analyzer using back trace on taint flow to identify critical patterns (Exploitable vs !Exploitable via euristic algorithms).
  - **objectAnalyzer.js:** Tainting back trace object analyzer used for information visualization in the views.

- **db.js**: SQLite DB management module.
- **DOMLog.db.sqlite**: SQLite Schema .
- **./components**:
  - **DOMIntruder.idl**: DOM exposed DOMinator object IDL.
  - **DOMIntruder.js**: Implementation of DOM exposed DOMinator object of log and other methods. Exposes an object named `___DI___` with the following methods:
    - *string go(in nsIDOMWindow win)*: Reads `./js/js_obj_debug.js` and injects in a head script element (to be improved).
    - *string logMe(in DOMString str)*: wrapper for btoa (to be deleted)
    - *string log( in DOMString str,in nsIDOMWindow win)*: (this method is called when source/sinks are tainted. It performs dispatching via ObserverService towards the Firebug DOMinator module).
    - *string jsonstr( in DOMString str)*: not used, to be removed.
    - *readonly attribute boolean wantCookies*: used to communicate with `js_obj_debug.js` wrappers if Cookies are to be considered tainted or not. To be improved. Probably is better to return a pref object. (Eg. {wantsCookies:true, wantsReferrer: true, useStackTraceDebugger:true etc}).
  - **DOMIObsDoc.js**: This xpcom component is an Observer waiting for `nsIWebProgress` events which are triggered when a DOM document is created.
- **./defaults/preferences**:
  - **defaults.js**: DOMinator default preferences.
- **./js**:
  - **jsInject\_obj\_debug.js**: This file contains a JavaScript which will be injected in the DOM on any HTML document. It contains the implementation of wrappers on Sink/Source and performs the extraction of the debugging information together with taint back trace of strings. Finally it calls the `___DI___`.log method defined in `./components/DOMIntruder.js` for the logging phase.
    - Example of a wrapper:

```
document.URL getter= function(){
    var s=(Components.lookupMethod(document, "URL")());
    s=String.newTainted((s),"URL");
    if(__domIntruderObj.__domIntruderDebugDom.logGetters &&
        __domIntruderObj.__domIntruderUtil.isToBeLogged(arguments))
        __domIntruderObj.__domIntruderUi.log("Getter","document.URL",s ,
            __domIntruderObj.__domIntruderUtil.getCallStack(arguments));
    return s;
};
document.URL setter= function(newv){
    if(newv.tainted)
        __domIntruderObj.__domIntruderUi.log("Setter","document.URL",newv,
            __domIntruderObj.__domIntruderUtil.getCallStack(arguments));
    Components.lookupMethod(document, "URL")(newv);
};
```

### 1.3 Internals Workflow

The workflow of the DOMinator extension follows (refer to previous figure):

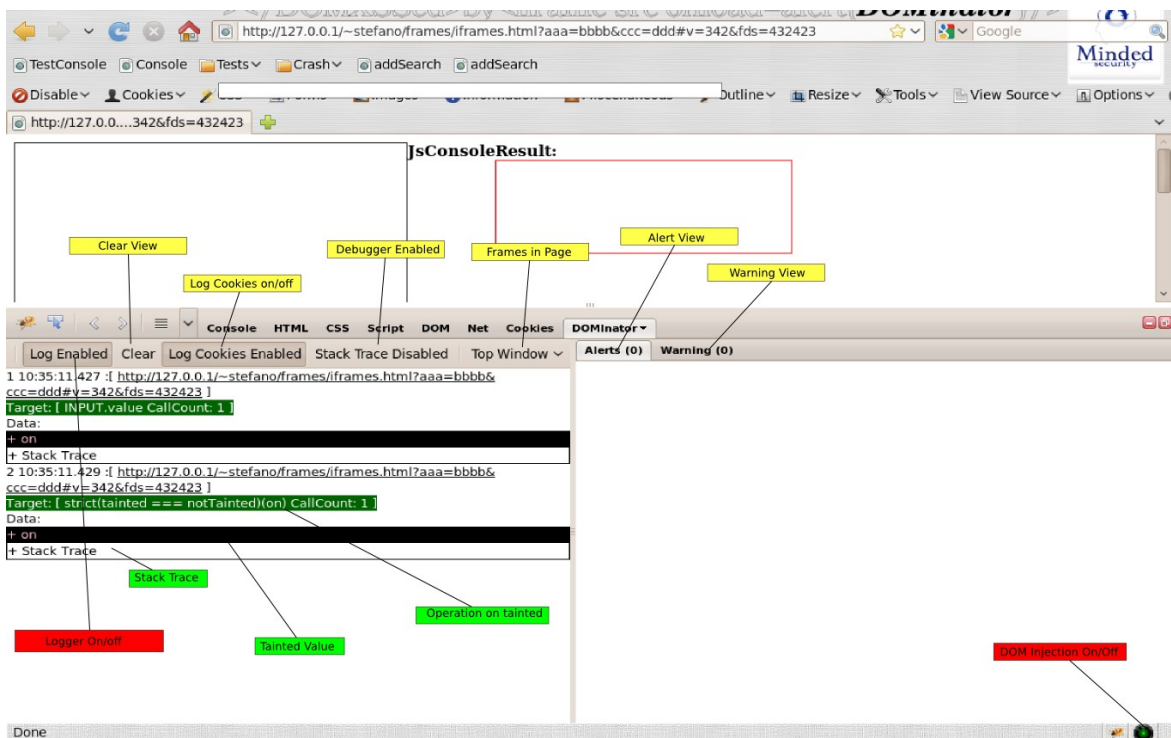
1. On first document DOM creation event DOMIObsDoc.js script inserts a SCRIPT node in the HEAD node which consists in a call to the object exposed by DOMIntruder.js component: (`__DI___.go(window)`)
2. once called at “user space”, method `__DI___.go(win)` reads `./js/js_Obj_debug.js` and insert it in HEAD node as a SCRIPT.
3. the injected script implements Source and Sinks wrappers, when any of them is triggered it will check if the arguments (getter/setter) is tainted and will perform the following steps:
  - 3.1. calls debugger to get the stack trace and saves it as an array.
  - 3.2. gets an object describing the tainting back trace about the tainted string by calling `String.getInfoTaint`.
  - 3.3. packs all the information in a JSON object
  - 3.4. calls log function `__DI___.log`.
4. `__DI___.log` method dispatch the event for any listening Observer.
5. At the moment the only observer for the log service is implemented in DOMIntruderFB and listens to “domintruder-logmessage” messages.
6. When observe method in DOMIntruderFB is called by the dispatcher:
  - 6.1. JSON data is parsed
  - 6.2. a DOMLogParser generates a Html node which will be used by the view

- 6.3. if the object data comes from a Sink exploitChecker is used (Eg if it's a sink location, *Sink\_location.isExploitable(ObjectToAnalyze)* is called) so that tainting backtrace information can be analyzed in order to define the exploitability level. This method returns a set of information used to identify if the sink is critical (exploitable) or not or even if it should not be visualized by the views.
- 6.4. otherwise (not a Sink) only tainting back trace information are extracted and the log is considered informative.
- 6.5. Saves everything in an in-memory DB.
- 6.6. Writes the node previously created in step 6.2 filled with all the extracted information in the correspondind view.
- 6.7. Views are updated with a new row containing all the needs in a way that can be readable by a human tester.

## 1.4 DOMinator GUI

DOMinator GUI is implemented as a Firebug extension in order the take advantage of Firebug's widgets and features.

At the moment there are a few buttons and menu as shown in the following figure:



When Firebug is activated the user will find a new Firebug tab named DOMinator, consisting in the following elements and widgets.

### **Buttons**

**Status Bar Button:** If DOMinator extension is installed there will be a small HAL9000 icon button in the status bar which enables or disables the script injection.

If it's disabled, the icon is red and any page will act as a normal HTML page, so normal navigation is possible. This feature acts globally.

**Log Disabled/Enabled Button:** the user can toggle log messaging on/off even if the script injection is still active. This feature can be used when there is high JavaScript activity or functions called in time intervals. When log is disabled nothing is written in the database. This feature acts globally.

**Clear Button:** This button clears the main view.

**Log Cookies Enabled Button:** This is a temporary functionality. The user can toggle this button to have less logging records or when cookies are considered a trustable source. This feature acts globally.

**Stack Trace Enabled:** This button will disable the stack trace creation for each log event. Stack trace is created using the debugger keyword and the JavaScript implementation is very slow. So it's suggested to keep it disabled until you need it for analysis purposes.

**Dropdown Frames Menu:** This menu is, at the time writing, just informative. It will list all the embedded frames in the page.

## *Views*

The panel is divided into two sub panels.

**Right Panel:** shows each record in chronological order.

**Left Panel:** contains two tabs:

- **Alert Tab:** This tab shows only operations on sinks that are supposed to be exploitable.
- **Warning Tab:** This tab shows operation that could be interesting for further analysis.

## *Rows description*

Each row on log panels represents a log record with the following format:

Log# TimeOfLog :[Frame location where it happened]

Operation

Tainted String

Stack Trace

**Log#:** represents a sequential number of the log event. This number is relative to the Frame the event happened.

**TimeOfLog:** represents the time the log event happened.

**Operation:** Describes the operation on tainted string which triggered the log event. Depending on the type of operation a colored background is displayed:

- **Green:** Operation on a source. This is informative.

- Example:  
`location.hash.split("|");`
- **Red**: The operation is a Sink and some of its argument are tainted.
  - Example:  
`document.write(location.search)`
- **Yellow**: an operation of assignment has been performed on a native Object or Array and only the Key is tainted.
  - Example:  
`o={}; o[location.hash]='test';`
- **Orange**: an operation of assignment has been performed on a Special Object (chrome Native) the Key is tainted.
  - Example:  
`window[location.hash]='test';`
- **LightCoral**: an operation of assignment has been performed on a Special Object (chrome Native) the Key AND the value are tainted .
  - Example:  
`window[location.hash]=location.search;`
- **Blue**: this color describes the following cases:
  1. an assigning operation to a sink object is performed.  
Example:  
`document.cookie="CookieName="+taintedValue+";..."`
  2. an XMLHttpRequest is performed and at least one of its arguments is tainted.  
Example:  
`(new XMLHttpRequest()).open("POST",location.hash.slice(1));`
  3. an operation of assignment has been performed on a Special Object (chrome Native) the Key AND the value are tainted.  
Example:  
`window[location.hash]=location.hash;`
- **Transparent**: In some cases the operation is displayed with no color. In particular:
  - if an object that contains tainted keys is requested with a non existent attribute:  
Example:  
`o={}; o=getQueryStringAsNameValuePairs();  
if(o.debug){  
  setDebugMode();  
}  
o.debug` it's not instantiated but it's requested and can be controlled.

The following screenshot shows the log contents after a log event triggered by the following code:

```
document.getElementById("result").innerHTML='sssss'+location.href+'/'+escape(location.hash);
```

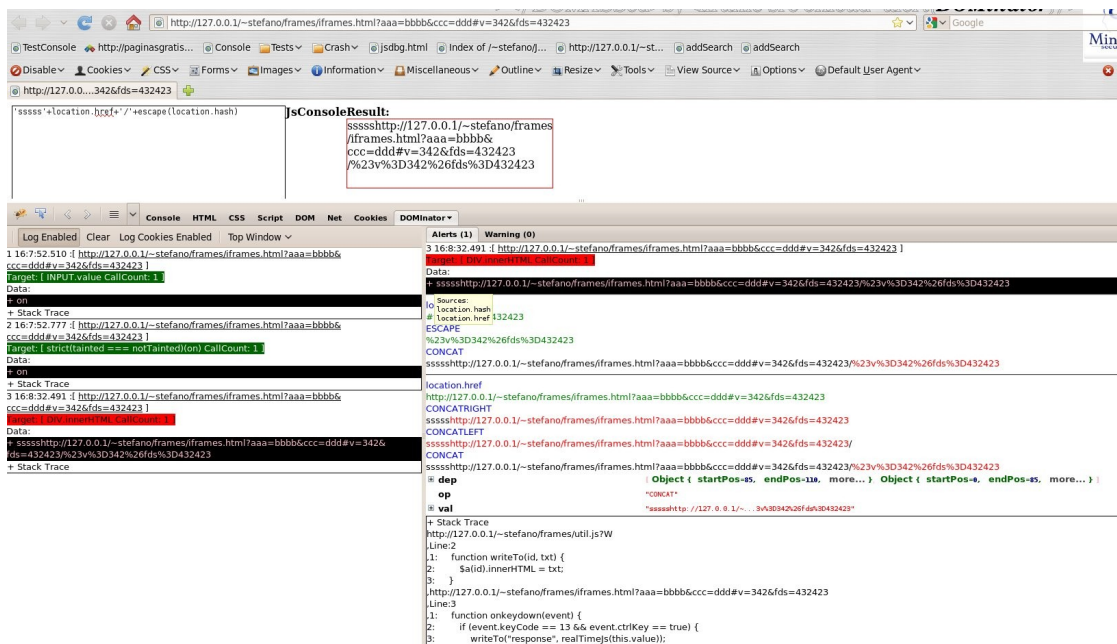
As can be noted, a red log is displayed in the Alert view because of location.href concatenation with no escaping. The user can see the operations performed on the tainted string starting from the used sources.

In this case used sources are location.href and location.hash. If the user moves the mouse on the final, tainted value (black background) a tooltip will show a list of all the original sources used to create the final value used by innerHTML.

By clicking on the box (in black background) under "Data:" string, a coloured list of the taint propagation trace that led to the final value is displayed. In this way the user can better understand where the exploitable injection is located.

Below the black div, there is the Stack Trace element which can be clicked in order to see the call stack which lead to the log alert.

Finally Alert and Warning tabs are updated with the total number of rows they contain each time a new row insertion happens.



## Known Issues

The following issues are known bug which will be fixed by me with the help of the community.

- Update management is missing.
- Regexp argument values (need modifications in Spidermonkey e Firefox) are missing in the tainting flow back trace.

- Source/Sink for Storage (needs Core Firefox modification)
- Missing About Dialog.
- Missing first DOMinator page on Firebug DOMinator view (branding).
- String characters extracted via s[i] are not tainted.
- Onunload Page does not clean up database (sometimes DOMinator Firefox must be closed to free some memory). (**Now Fixed**)
- Events like location=location.hash.slice(1) are not correctly visualized because of firebug model, persistent data model must be implemented according to Firebug guidelines.
- debugger stack trace is too slow, a C++ component should be implemented. (probably in the commercial version)
- Sometimes DOMinator crashes, it's probably due to the intensive use of the jsd debugger. (Firefox/spidermokey bug)
- <!-- [if ie] ... is obviously not parsed by firefox, so it completely misses them.
- It probably has some trusting issues between untrusted DOM and privileged code.