TransUnion

Integrating with Web Sites

How Web Integration Works

What is Multi-Domain Recognition?

TransUnion device recognition employs scripts from both your domain (the First-party domain) and TransUnion's domain (the Third-party domain). Including scripts from both of these domains allows TransUnion to:

- **Third-Party JavaScript:** Share fraud history for devices and accounts across TransUnion subscribers.
- **First-Party JavaScript:** Collect device information for users whose browsers are configured to disable third-party JavaScript, or that block the TransUnion domain.

TransUnion will provide a script for you to include on your site. This script will load the appropriate resources from both sets of domains. Components loaded by the script are dynamically generated and therefore not included with the script provided, nor should they be directly included on your page.

Retrieving the dynamic first party script will require some set up on your servers or network.

First-Party Dynamic JavaScript Options

Retrieving the dynamic first-party script can be done by:

- Setting up a reverse proxy to get the files from an TransUnion-hosted server
- Installing your own Java device print server.

Retrieving First-Party Dynamic JavaScript

Overview

You must serve up the TransUnion first-party dynamic JavaScript from within your own infraustructure. You can do this in one of the following ways:

- Use a reverse proxy to get the JavaScript components from TransUnion
- Host a WAR file, provided by TransUnion, that includes the JavaScript components

Retrieving First-Party Dynamic JavaScript Components via a Reverse Proxy

About Reverse Proxy Configuration

To deliver the first-party dynamic JavaScript components:

- You must tunnel requests through your site to TransUnion's servers.
- To do this, set up a reverse proxy that will forward all requests sent to a specific URI on your site to TransUnion's servers. You can define this URI in the configuration variables for the TransUnion-provided script.

Test and Production Server URLs

TransUnion provides the first-party dynamic JavaScript components from both test and production servers, at the following URLs:

Test	https://ci-first.iovation.com/*
Production	https://first.iovation.com/*

Do Not Cache Dynamic Components Using Content Distribution Networks (CDNs)

When using a Content Distribution Network (CDN) such as Akamai, ***make sure that the dynamic files are not cached by the CDN***. Caching these files results in sets of dynamic attributes becoming fixed for disparate sets of users, severely disrupting the recognition process. can severely disrupt the recognition process as the dynamic attributes will become fixed for a large disparate set of customers.

For Akamai, ensure that the caching option for the reverse proxy rule is set to no-store. This will prevent Akamai from caching the content and instead get new copies each time.

Configuring the Reverse Proxy

Important Configuration Notes

When setting up the reverse proxy:

- Because multiple resources may be requested, you must limit which specific files are forwarded.
- However, also provide a generic-enough mapping so that when TransUnion adds or updates resources in the future, you will get all of the updates.

IMPORTANT! When setting up a reverse proxy, ***do not:***

- Forward requests to the URI directly to TransUnion through a redirect; these requests can be blocked
- Create a special sub-domain; this can be just as easily blocked as TransUnion's domain
- Create a custom handler that intercepts the request and then requests the content on behalf of the customer; this prevents various HTTP headers from being analyzed from the user's machine. Standard reverse proxies will preserve those headers.

Configuration Steps

To configure the reverse proxy:

1. Set up a proxy configuration within your domain.
2. Specify a URI on your site to proxy the request. The default URI is `iojs`, however you can change this in the configuration section of your TransUnion JavaScript. Using the default URI, you would forward to TransUnion any request beginning with: http://my.domain.com/iojs
3. Direct the proxy to forward the requests to the following URL: https://first.iovation.com

Example: Configuring a Reverse Proxy in Nginx

Edit the Nginx configuration file (**default.conf**) and add the following lines to the server section. You must set the proxy path to iojs.

```
server {
    .. other configuration entries ...
    location /iojs/ {
    proxy_pass https://first.iovation.com/;
    }
    ... }
```

Save the configuration file and restart Nginx.

Example: Configuring a Reverse Proxy in Apache

Edit the Apache configuration file (**httpd.conf**) and enable the following modules:

- proxy_module

- proxy_http_module - to enable SSL forwarding

```
LoadModule proxy_module      modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

Add the following lines, in the main configuration or VirtualHost directive, assuming iojs is the URI.

```
SSLProxyEngine  On
ProxyPass /iojs/ https://first.iovation.com/
```

Save the configuration file and restart Apache.

Testing New Releases of the First-Party JavaScript

TransUnion periodically releases new versions of the first-party JavaScript. We strongly recommend testing these updates in our customer integration (CI) environment before we release production versions. We will notify you when we release updates, and provide up to two weeks to test. To test the updated JavaScript, complete the following steps:

1. Change the URLs in your proxy configurations to the following test URL in our CI environment:
   https://ci-first.iovation.com/
   When you have completed testing, restore the proxy configurations back to the production URL:
   https://first.iovation.com/
2. Test incoming transactions to verify the following:
   - There are no resource loading issues resulting in time-out responses back to the browser.
   - There are no changes in web page appearance, performance, or behavior.
   - There are no errors in the browser console.

Deploying First-Party JavaScript via Your Own Web Device Print Server

Installing a Web Device Print Server

IMPORTANT! If you will use TransUnion with more than one domain you must set up web device print servers for each domain.

1. Download the Multi-Domain Recognition deployment package from here:
   https://help.iovation.com/004_Download_SDKs/Download_Web_Device_Print_Components/Download_Web_Device_Print_Components
   This WAR file contains all of the files required to set up a web device print server in your domain.
2. If you don't already have a 64 bit Java web server **within the domain and port from which your pages are served**, establish one. You may also choose to set up a dedicated Java Server for the TransUnion JavaScript.
   a. If necessary, install a Java web server to a system that will be accessible to end users of

your service, such as a server in a DMZ. The server must reside under the same domain and port from which your pages are served.
For example, you can download Apache Tomcat for your OS: https://tomcat.apache.org/download-80.cgi

3. Deploy the web device print WAR file. For example, to deploy the WAR file to a Tomcat installation, place it under the /webapps folder under your Tomcat installation folder.
4. Create the file wdpoverride.properties and place in your CLASSPATH:
   a. Create a new file named wdpoverride.properties and place in your web server's CLASSPATH .
   b. In a text editor, add the text below to your **wdpoverride.properties** file. Replace `server_identifier` with an identifier for your server. This will help track where scripts were sent from. It is not recommended to use a value that will reveal information about your network infrastructure, such as hostnames . You will also need to change wdp-first-service to the URI/Context on your server where the WAR file is installed:

```
parameter.content.server.path=base64_encode(wdp-first-service/resources/
static)
parameter.realip.server.host=base64_encode(mpsnare.iesnare.com)
# Origin of the JavaScript
parameter.wdp.server.host=base64_encode(server_identifier)
# URL to a script that stores a token in your users' browser cache
parameter.ctoken.script.path=base64_encode(wdp-first-service/latest/logo.js)
parameter.wdp.js.versions=latest
parameter.wdp.js.version.latest=4.1.6
parameter.rtc.server.list=base64_encode(stun:stun.l.google.com:19302,
stun:stun3.l.google.com:19302,stun:stun2.l.google.com:19302,
stun:stun.stunprotocol.org:3478,stun:numb.viagenie.ca:3478,
stun:stun.vivox.com:3478,stun:stun.sip.us:3478,stun:stun.commpeak.com:3478,
stun:stun.barracuda.com:3478,stun:stun.epygi.com:3478)
```

5. Make the following changes to prevent memory leaks on your server from asynchronous logging appenders:
   a. In the WEB-INF/web.xml file for wdp-first-service, un-comment the `isLog4jAutoInitializationDisabled` parameter and set it to `true`:

```
<context-param>
    <param-name>isLog4jAutoInitializationDisabled</param-name>
    <param-value>true</param-value>
</context-param>
```

   b. Un-comment `Log4jServletContextListener` and `Log4jServletFilter`:

```
<listener>
    <listener-class>org.apache.logging.log4j.web.
Log4jServletContextListener</listener-class>
</listener>

<filter>
    <filter-name>log4jServletFilter</filter-name>
    <filter-class>org.apache.logging.log4j.web.Log4jServletFilter</filter-
class>
</filter>

<filter-mapping>
    <filter-name>log4jServletFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
    <dispatcher>ASYNC</dispatcher>
</filter-mapping>
```

6. Start the web device print server. Monitor the logs on the Java web server to make sure that there are no errors.
7. To verify that the web device print server is running correctly, open the following URL in a browser:
   http://*yourserver*:*port*/wdp-first-service/latest/dyn_wdp.js
   For example, if your server is called **deviceprint.com** and your port number is **80**, open:
   http://deviceprint.com:80/wdp-first-service/5.0.0/dyn_wdp.js
   If the server is running correctly the JavaScript will display in the browser.

Configuring Device Recognition JavaScript

Overview

These topics walk you through configuring TransUnion's JavaScript. TransUnion's JavaScript collects device information that you can then submit to TransUnion as part of your Device-Based Authentication or Fraud Prevention implementation.

Configuring TransUnion JavaScript on Web Pages

To configure TransUnion's JavaScript, you must define settings for a configuration object that is used by the JavaScript. If the object is not defined, default values will be used.

The configuration object has various sections and looks something like the following:

```
/* Copyright(c) 2016, iovation, inc. All rights reserved. */
window.io_global_object_name = "IGLOO";
window.IGLOO = window.IGLOO || {
  "bbout_element_id" : "iobb",
  "bb_callback": null,
  "loader" : {
    "uri_hook" : "iojs",
    "version" : "general5",
    "subkey" : "5FExse+oA1134BhiwCF2EeQ1TfisPJGha4CpVG2nd7E="
  }
};
```

TransUnion will provide you with a configuration script in addition to the collection script. You can include these separately, combine them or in-line the definition on your page.

IMPORTANT!

It is critical that the *configuration* comes **before** *TransUnion's loader script* otherwise configuration variables will not be used once the script starts running.

Configuration script sections

TransUnion Global Object Name

Before delving into the options available, a quick word on the io_global_object_name, IGLOO, by default. In earlier versions of the scripts, TransUnion configured the script through the use of global variables. While this approach is simple enough, it can add a lot of additional variables to the window object which has the potential to collide with other scripts.

To remedy this, TransUnion has created an object that encapsulates all of the settings and functionality and you can change the ID to prevent potential collisions by changing the name.

```
window.io_global_object_name = "<your custom name>"
```

If you do change the value, references functions and values as follows:

```
window.<your custom name>.xxxxx
```

Change the following as well:

```
window.IGLOO = window.IGLOO ||
```

to match your new custom name.

TransUnion configuration object has 2 main components:

- Generic settings that indicate how to retrieve a device print and restrictions on what information is collected, for instance Flash values
- Loader settings that indicate how to access first-party components and basic debug and versioning information

Each set of configuration options belongs in a specific section of the object:

```
/* Copyright(c) 2016, iovation, inc. All rights reserved. */
window.io_global_object_name = "IGLOO";
window.IGLOO = window.IGLOO || {
  // generic settings go here as
  // "option" : value
  ...
  "loader" : {
     // loader configuration options go here as
     // "option" : value
  }
};
```

Generic Configuration Options

Parameter	Type	Default Value	Description
`bbout_element_id`	string, optional		The ID of the HTML element to populate with the blackbox from the third-party JavaScript. If `bb_callback` is specified, this parameter has no effect.

`bb_callback`	function, optional		This JavaScript function is an event handler that is called when a collection method has finished updating a blackbox. This must be a function, not a string.
			
			Declare the function as follows:
			
			`"bb_callback" : function ( bb, complete) {`
			`  // code to process blackbox here`
			`}`
			
			The variables store the following:
			
			• `bb` – the updated value of the blackbox
			• `complete` – a boolean value indicating whether all the collection methods have completed.

`enable_rip`	boolean, optional	`true`	

Loader Configuration Options

Parameter	Type	Default Value	Description
`uri_hook`	string, optional	`iojs`	Location of dynamic first party components. This should be a reference to the web directory being proxied. You can use relative or absolute references, but should not use a complete URL.

Parameter	Type	Default Value	Description

			For example, if your reverse proxy is accessed from `http://mysite.com/iov/wdp/....`, and your page is loaded from: `http://mysite.com/app/mypage.html`, you would use:
			`"uri_hook" : "/iov/wdp"`
			
			If the reverse proxy is accessed at `http://mysite.com/app/iojs` and your page is at `http://mysite.com/app/page.html`, you might use:
			`"uri_hook" : "iojs"`
			
			NOTE
			This should not be a URL (i.e. include http(s)://host:port) as this will fail and the scripts must be loaded from the same domain as the page.

`subkey`	string, optional		This will be an TransUnion assigned value that tracks requests from your site. This is primarily used for debugging and troubleshooting purposes.
`version`	string, required	`general5`	This is the version of the script to load. The value should either correspond to a specific version you wish to use, or one of the following aliases to get the latest version of the code: • `general5` - the latest stable version of the JavaScript • `early5` - the latest available version of the JavaScript `general5` and `early5` may be the same code, however, any new changes will be released on `early5` prior to `general5`. Once the new release has been vetted in production and deemed satisfactory, `general5` will be updated to match `early5`.

`trace_handler`	function, optional		This JavaScript function can be used to provide tracing messages for the script. This will provide information on the progress of the script and is useful for debugging purposes.
			
			Declare the function as follows:
			`"trace_handler" : function (message) {`
			`// process/record trace message here`
			`}`

Parameter	Type	Default Value	Description
			where `message` is the trace message provided by the script.

Example Configuration File

Putting together basic configuration settings, results in a file similar to:

```
/* Copyright(c) 2016, iovation, inc. All rights reserved. */
window.io_global_object_name = "IGLOO";
window.IGLOO = window.IGLOO || {
  "bbout_element_id" : "iobb",
  "loader" : {
    "uri_hook" : "/iojs",
    "version" : "general5",
    "subkey" : "5FExse+oA1134BhiwCF2EeQ1TfisPJGha4CpVG2nd7E="
  }
};
```

Collecting a Blackbox

Methods for Collecting a Blackbox

TransUnion's script creates a blackbox that you must collect and send to your server. This blackbox will contain all the device information collected from the various sources.

There are three ways to collect a blackbox.

- Populate a hidden form field using the `bbout_element_id` parameter.
- Define the `bb_callback` function to collect the blackbox as it is generated.
- Call the JavaScript function `window.IGLOO.getBlackbox` to obtain the blackbox. You can combine `getBlackbox` and either of the other two methods.

Implementing the Hidden Form Field Collection Method

You can define a hidden form field for the script to populate. The blackbox will then be submitted along with other fields in the form. To use this method you must define `bbout_element_id` in the configuration script. This method is primarily useful when you have a single form. For other cases, consider using one (or both) of the other methods.

1. Add a hidden field to a form on your web page and set the `id` attribute for the field. This field will store the blackbox. Give the `id` attribute a descriptive name; you will need to reuse this later. Here is an example of a simple form with a hidden field, with the `id` attribute set to `ioBlackBox`:

```
<form action="/do_ctd" method="post" name="loginSubmitForm" id="loginSubmitForm">
            <fieldset>
                <ul>
                    <!-- Create hidden for blackboxes to go into -->
                    <input TYPE="hidden" NAME="ioBlackBox" id="ioBlackBox">
                    <li><input type="submit" value="Do CTD" id="submit1"
name="submit1">
                   </li>
                </ul>
             </fieldset>
          </form>
```

2. In the JavaScript configuration parameters, set the `bbout_element_id` parameter to the id of the hidden form field. For example, if the `id` attribute is set to `ioBlackBox`, set `bbout_element_id` to the following:

```
/* Copyright(c) 2016, iovation, inc. All rights reserved. */
window.io_global_object_name = "IGLOO";
window.IGLOO = window.IGLOO || {
  "bbout_element_id" : "ioBlackBox",
  "loader" : {
    "version" : "general5",
    "subkey" : "5FExse+oA1134BhiwCF2EeQ1TfisPJGha4CpVG2nd7E="
  }
};
```

Implementing the Callback Collection Method

The callback interface allows you to manage blackbox generation in a more event-driven manner. As blackbox collection progresses,the script fires update events as collection methods complete. These events trigger a user-defined callback function to update the page with the new blackbox value. When all of the collection methods are completed, a Boolean flag is set indicating no further updates are expected and the value is the final blackbox value.

In the JavaScript configuration parameters, set the `bb_callback` parameter to a function that processes the blackbox value, and that has the following signature:
`function bb_update_callback( bb, complete)`
Where:

- `bb` is the updated value of the blackbox
- `complete` is a boolean value that indicates whether all collection methods (Flash, etc.) are complete

```
/* Copyright(c) 2016, iovation, inc. All rights reserved. */
window.io_global_object_name = "IGLOO";
window.IGLOO = window.IGLOO || {
  "bb_callback": function ( bb, complete ) {
                    var bb_field = document.getElementById( "bb" );
                    bb_field.value = bb;
                },
  "loader" : {
    "version" : "general5",
    "subkey" : "5FExse+oA1134BhiwCF2EeQ1TfisPJGha4CpVG2nd7E="
  }
};
```

NOTE If `bb_callback` and `bbout_element_id` are both specified, the hidden field specified in `bbout_element_id` will not be populated, unless explicitly done so by the function specified in `bb_callback`.

Implementing the getBlackbox Collection Method

The last method for obtaining a blackbox is to use the `getBlackbox` function. This function returns an object that contains the current value of the blackbox along with a flag indicating whether the collection process has completed. This method is useful when the value is needed after the collection process has completed. It is also useful as a way to obtain the best value after some maximum amount of time to avoid any further delays in the user experience.

To implement the `getBlackBox` collection method:

1. Call the function. For example:

   ```
   var blackbox_info = window.IGLOO.getBlackbox();
   ```

   This returns an object with the following attributes:

   - `blackbox` – the updated value of the blackbox
   - `finished` – a Boolean indicating whether all the collection methods have completed.

Troubleshooting Errors

If you encounter errors such as failing to get a blackbox, you can use the loader configuration variable `trace_handler` parameter to troubleshoot. This will send output and status messages to your trace handler. A simple way to troubleshoot is to use something similar to:

```
/* Copyright(c) 2016, iovation, inc. All rights reserved. */
window.io_global_object_name = "IGLOO";
window.IGLOO = window.IGLOO || {
  bbout_element_id : "iobb",
  "loader" : {
    "version" : "general5",
    "subkey" : "5FExse+oA1134BhiwCF2EeQ1TfisPJGha4CpVG2nd7E=",
    "trace_handler": function ( msg ) {
                      console.log( msg );
                  }
  }
};
```

Blackbox Collection Examples

Overview

These examples will help you integrate TransUnion risk services into your web pages.

NOTE Besides the code samples below, it is important to set up the reverse proxy or web device print server as well. Make sure `uri_hook` in the `loader` configuration section is set appropriately. These examples assume the default configuration of `/iojs`.

Hidden Form Field Collection Example

Here is a sample web page that uses the hidden form field collection method.

```
<html>
<head>
  <title>Demo Integration Page</title>
  <meta charset="UTF-8">
</head>
<body>
  <form action="/do_ctd" method="post" name="loginForm" id="loginForm">
     <!-- Create a hidden field for the blackbox -->
     <input TYPE="hidden" NAME="ioBlackBox" id="ioBlackBox">
     <input type="submit" name="submit1">
  </form>
<!-- Include iovation JavaScript -->
<script language="javascript" src="config.js"></script>
<script language="javascript" src="loader.js"></script>
</body>
</html>
```

```
<html>
<head>
  <title>Demo Integration Page</title>
  <meta charset="UTF-8">
</head>
<body>
  <form action="/do_ctd" method="post" name="loginForm" id="loginForm">
     <!-- Create a hidden field for the blackbox -->
     <input TYPE="hidden" NAME="ioBlackBox" id="ioBlackBox">
     <input type="submit" name="submit1">
  </form>
<!-- Include iovation JavaScript -->
<script language="javascript" src="config.js"></script>
<script language="javascript" src="loader.js"></script>
</body>
</html>
```

Using the page layout from above, the hidden form field has an id of `ioBlackBox` that we set in `config.js` as `bbout_element_id:`

```
window.io_global_object_name = "IGLOO"
window.IGLOO = window.IGLOO || {
  "bbout_element_id" : "ioBlackBox",
  "loader" : {
    "version" : "general5",
    "subkey" : "5FExse+oA1134BhiwCF2EeQ1TfisPJGha4CpVG2nd7E="
  }
};
```

Callback Collection Example

The following example illustrates using the callbacks from each set of scripts to update multiple form fields. This simulates the case where an end user is reviewing an order that has a submission form at the top and bottom of the page. The example updates all fields when there is an update - not simply when collection is finished. This ensures that TransUnion will have something to work with, even when both collection methods do not fully complete.

```
<html>
<head>
  <title>Demo Integration Page</title>
  <meta charset="UTF-8">
</head>
<body>
<!-- Form 1 -->
  <form method=POST action="#">
    <input type=hidden name="io_bb"></input>
    <input type=submit name="Go first!"></input>
  </form>
<!-- Form 2 -->
  <form method=POST action="#">
    <input type=hidden name="io_bb"></input>
    <input type=submit name="Go second!"></input>
  </form>
<!-- Include iovation JavaScript -->
<script language="javascript" src="config.js"></script>
<script language="javascript" src="loader.js"></script>
</body>
</html>
```

```
<html>
<head>
  <title>Demo Integration Page</title>
  <meta charset="UTF-8">
</head>
<body>
<!-- Form 1 -->
  <form method=POST action="#">
    <input type=hidden name="io_bb"></input>
    <input type=submit name="Go first!"></input>
  </form>
<!-- Form 2 -->
  <form method=POST action="#">
    <input type=hidden name="io_bb"></input>
    <input type=submit name="Go second!"></input>
  </form>

<!-- Include iovation JavaScript -->
<script language="javascript" src="config.js"></script>
<script language="javascript" src="loader.js"></script>
</body>
</html>
```

Using the page layout from above, the callback function needs to populate both `io_bb` fields in the various forms. We do this using `bb_callback` in `config.js`:

```
window.io_global_object_name = "IGLOO"
window.IGLOO = window.IGLOO || {
  "bb_callback" : function (bb, complete) {
         var fields = document.getElementsByName( "io_bb" );
         var i = 0;
         for ( i = 0; i < fields.length; i++ )
            fields[i].value=bb;
     },
  "loader" : {
    "version" : "general5",
    "subkey" : "5FExse+oA1134BhiwCF2EeQ1TfisPJGha4CpVG2nd7E="
  }
};
```

Get Blackbox Function Collection Example

This example illustrates how to use the blackbox method. Submission of the form initates the capture of the blackbox. You can alter the send function to post the blackbox via AJAX or perform some other operation. In the example, an alert displays the collected value.

```
<html>
<head>
  <title>Demo Integration Page</title>
  <meta charset="UTF-8">
</head>
<body>
  <form method=POST action="#">
    <input type=submit name="Go first!" onclick="return send_bb();"></input>
  </form>
  <script type="text/javascript">
     function send_bb() {
         // make AJAX call here or do something else with blackbox
         // for illustration purposes, we are just going to do an alert here
         var bb = "";
         try {
            bb = window.IGLOO.getBlackbox();
            alert( "bb: " + bb.blackbox );
         } catch (e) { alert( "Unable to get blackbox. " + e );
     }
   </script>
<!-- Include iovation JavaScript -->
<script language="javascript" src="config.js"></script>
<script language="javascript" src="loader.js"></script>
</body>
</html>
```

As the functional interface is being used, there are no special configuration options required.

```
window.io_global_object_name = "IGLOO"
window.IGLOO = window.IGLOO || {
  "loader" : {
    "version" : "general5",
    "subkey" : "5FExse+oA1134BhiwCF2EeQ1TfisPJGha4CpVG2nd7E="
  }
};
```