Secure Browser API Specification

v.2.0.0 - Last modified 17-Mar-2017

The following Secure Browser Application Programming Interface (API) endpoints define interfaces between the secure browser and the test delivery system. The interfaces consist of required and optional methods, as shown below. All APIs depend on the first requirement, the global `browser` object.

Required Methods

1. R01. **Expose a window global object called `browser`**. All vendors are required to expose a window global object called `browser`. The APIs discussed below are exposed through this global object. Some APIs are supported as part of existing W3C specifications, and are identified as such.

2. R02. **Lock down environment to begin an assessment**. The testing web application will invoke this call prior to allowing students to start testing. The implementer is required to take any actions necessary to secure the testing environment. The steps taken to secure the environment are device specific and for example, include aspects such as disabling the ability to do screen captures, disabling the ability to voice chat when in secure mode, clearing the system clipboard, entering into a kiosk mode, disabling Spaces in OS X 10.7+ etc. The testing application will enable lockdown before an assessment commences and will disable the lockdown when the student has completed the assessment and is out of the secure test.

   ```
   void browser.security.lockDown (boolean enable, function onSuccess, function onError)
   ```

   `onSuccess` and `onError` are optional parameters. These functions are invoked after the lockdown has either been sucessfully enabled or disabled or if the operation failed. If you specify these callback parameters, it should be a function that looks like this:

   ```
   function(currentlockdownstate) {...}
   ```

   where `currentstate` indicates whether the lockdown is in effect or not. undefined or null indicates that we don't know what the current state is (likely only the case for onError callbacks)

3. R03. **Check if environment is secure**. Check if the environment is secure. The testing web application will invoke this prior to allowing students to start testing and periodically when inside the test.

   ```
   void browser.security.isEnvironmentSecure(function callback)
   ```

   `callback` is optional. If you specify this parameter, it should be a function that looks like this:

   ```
   function(state){...}
   ```

   The state is a JSON string containing two fields. The first is the `secure` field, which must return `true` only if all necessary locks have been enabled (or features disabled) to enable a secure testing environment, and none of these have been compromised since we entered the lockdown mode. The other field (`messageKey`) includes other details or information that is vendor specific. The intent here is to allow vendors to put additional information that augments the boolean `secure` flag:

   ```
   { 'secure' : "true/false", 'messageKey' : "some message" }
   ```

4. R41. **Retrieve the status of a particular browser capability**. `object browser.security.getCapability("feature")`

   returns either a Javascript object or literal with the following structure

`{<feature>:true|false}`

5. R42. **Set the status of a particular browser capability**. This allows us to explicitly enable or disable a specific feature on the browser. The feature must be one that is recognized by the vendor.

   ```
   void browser.security.setCapability(feature, value, function onSuccess, function onError)
   ```

   `onSuccess` and `onError` are optional parameters. These functions are invoked after the feature has either been sucessfully enabled or disabled or if the operation failed. If you specify these callback parameters, it should be a function that looks like this:

   ```
   function(jsonliteral) {...}
   ```

   where jsonliteral indicates the current value of the feature and it's state. `{<feature>:true|false|undefined}`. If the feature is unknown to the browser, the value for the key will be undefined

6. R05. **Retrieve information on the environment (device)**. The testing web application will invoke this to gather details about the platform on which it is running. This is used to augment any information that was discernible from the user agent.

   ```
   void browser.security.getDeviceInfo(function callback)
   ```

   `callback` should be a function that looks like this:

   ```
   function(infoObj){...}
   ```

   The infoObj is a JSON literal that includes a set of key value pairs. The complete list of data returned is vendor specific. At a minimum, we expect the following keys to be supported: `manufacturer`,`HWVer`, `SWVer`

7. R06. **Examine current list of running processes**. The testing application will invoke this to examine the list of all processes running on the client machine owned by the user, and compare it with a list of processes that we have deemed blacklisted during testing cycle. This call will be invoked both at the start of an assessment and periodically which the student is taking the assessment and at any point, if a blacklisted app is detected, the assessment will be stopped to preserve test integrity. It will return a list of running processes that match one or more of the blacklisted processes, if any.

   ```
   void browser.security.examineProcessList(string[] blacklistedProcessList, function callback)
   ```

   `callback` should be a function that looks like this:

   ```
   function(array of process names found){...}
   ```

   Example response: "
   `['taskmgr.exe','chrome.exe','ccSvcHst.exe','Dropbox.exe','EXCEL.EXE','svchost.exe','System']`"

   An empty array indicates no forbidden apps were found. Undefined or null return value to the callback indicates that some error occurred and we were unable to perform the match.

8. R07. **Close the browser**. The testing application will invoke this to shut down the browser when the user elects to exit the browser. The boolean parameter will determine if the browser should restart on exit or simply exit.

   ```
   void browser.security.close(boolean restart)
   ```

9. R15. **Is macOS Spaces Enabled**. Applicable to macOS only. This runtime browser property can be read by the testing application and returns true if Spaces is enabled, false otherwise.

   `browser.settings.isSpacesEnabled`: boolean property specifying if Spaces is enabled or not. It is undefined on all platforms other than those that implement 'Spaces'.

10. R16. **Get System Volume**. Get system volume: This runtime browser property can be queried by the testing application to get the System Volume. This is only available in desktop secure browsers.

    `browser.settings.systemVolume` : integer property that contains the scaled value of the current system volume (0-10).

11. R17. **Set System Volume**. Set system volume: This runtime browser property can be written to by the testing application to set the System Volume. This is only available in desktop secure browsers.

    `browser.settings.systemVolume` accepts a value for the system volume (0-10)

12. R18. **Mute System Volume**. Mute system volume: This runtime browser property can be written to by the testing application to mute the System Volume. This is only available in desktop secure browsers.

    `browser.settings.systemMute`: boolean property that reports whether speaker is muted or not.

13. R19. **Unmute System Volume**. Unmute system volume: This runtime browser property can be written to by the testing application to unmute the System Volume. This is only available in desktop secure browsers.

    `browser.settings.systemMute` set to true to mute, false to unmute.

14. R20. **Get System Mute Status**. Get the current status of the system volume. This is only available in desktop secure browsers.

    `browser.settings.systemMute` true if muted, false if unmuted.

15. R44. **Get permissive mode**. The testing web application will invoke this to determine if permissive mode is on or off. In permissive mode, a browser is expected to relax some of its stringent security hooks to allow assistive technology to work with the secure browser. For example, browsers that aggressively prevent other application UIs from presenting on top of them might want to relax this when in permissive mode.

    ```
    void browser.security.getPermissiveMode(function callback)
    ```

    `callback` is optional. If you specify this parameter, it should be a function that looks like this:

    ```
    function(permissivemode){...}
    ```

    The passed-in value is a boolean indicating if the browser is permissive or not. If undefined or null is passed in, that implies an error occurred with the get operation.

16. R21. **Set permissive mode**. The testing web application will invoke this to toggle permissive mode on or off. In permissive mode, a browser is expected to relax some of its stringent security hooks to allow assistive technology to work with the secure browser. For example, browsers that aggressively prevent other application UIs from presenting on top of them might want to relax this when in permissive mode.

    ```
    void browser.security.setPermissiveMode(enable, function callback)
    ```

    `callback` is optional. If you specify this parameter, it should be a function that looks like this:

    ```
    function(permissivemode){...}
    ```

    The passed-in value is a boolean indicating if the browser is permissive or not. If undefined or null is passed in, that implies an error occurred with the set operation.

17. R22. **Eliminate Security-Exposing APIs**. The following APIs are deprecated and shall NOT be exposed:

    - *Clear browser cache* (`void browser.security.clearCache()`)

- *Clear cookies* (`void browser.security.clearCookies()`)
- *Retrieve client IP address(es)* (`string[] browser.security.getIPAddressList()`)
- *Retrieve current list of running processes* (`string[] browser.security.getProcessList()`)

Text To Speech Synthesis (TTS)

NOTE: Browsers supporting W3C's Web Speech API do not need to implement these functions.

1. R08. **Speak Text (TTS)**. The testing application will invoke this to perform client side text to speech synthesis. The API call will be passed in a string with embedded speech markup (the param is required, the markup is optional), an options object to control the speech (required param) and a callback for TTS events (optional). The vendor must support plaintext and optionally support one of the following markup standards; SSML, Microsoft speech markup (for Windows) or Apple speech markup (for macOS). The ability to set the pitch, rate, voice, and volume is provided by this API call through the options object, which includes:

   `voicename` (required) - The voice to use from the getVoices call.

   `rate` (optional) - Speech playback rate, ranging from 1 to 20, where 10 is the default, 20 is twice as fast as 10, and 5 is half as fast as 10. 1 is the slowest available playback rate.

   `pitch` (optional) - Speech pitch, ranging from 1 to 20, where 10 is the default, but the actual pitch is voicepack dependent.

   `volume` (optional) - Speech volume, ranging from 0 to 10: where 5 is the default and 10 is twice as loud as 5. 0 will mute TTS. The speech volume is dependent on the system volume.

   `language` (optional) - Speech language, following the xml:lang attribute specification. This optional attribute can be used to narrow down the available voice names if more than one voice pack matches the specified voice name.

   `gender` (optional) - Indicates the preferred gender of the voice to speak the contained text. Enumerated values are: "male", "female", "neutral". This optional attribute can be used to narrow down the available voice names if more than one voice pack matches the specified voice name.

   The callback, if provided, is invoked for TTS events which include `start`, `end`, `word boundary`, `sentence boundary`, `synchronization/marker encountered`, `paused`, `resumed`, and `error`.

   ```
   void browser.tts.speak(string text, object options, function callback)
   ```

   The callback function's parameters are as follows:

   ```
   callback(reason, parms)
   ```

   where `reason` is a string which includes one of the following values: 'start', 'end', 'pause', 'resume', 'word', 'sentence', 'mark', and 'error', indicating a word, sentence, or mark boundary, or an error. For each of these `reason` strings, the `parms` would be an object

   ```
   parms = { start, end, length, type };
   ```

2. R09. **Stop speech (TTS)**. This is called by the testing application to stop any speech that may be in progress.

   ```
   void browser.tts.stop(function callback)
   ```

   'callback' is an optional function that if present, will be invoked with a string status when TTS has stopped speaking, or an error has occurred while trying to make speaking stop (e.g., it was not speaking at the time). The state string will contain a code sych as 'stop' (if speech was stopped) or 'error' (if an error occurred).

```
callback(state)
```

3. R10. **Get speech status (TTS)**. This is called by the testing application to inspect the current status of speech. The valid values are listed below.

   ```
   void browser.tts.getStatus(function callback)
   ```

   `callback` is optional. If you specify this parameter, it should be a function that looks like this:

   ```
   function(status){...}
   ```

   Where `status` is one of:

   `NotSupported` : TTS initialization failed.

   `Uninitialized` : TTS is not initialized

   `Initializing` : TTS initialization in progress

   `Stopped` : TTS is initialized and there is nothing playing

   `Playing` : playing is in progress

   `Paused` : playing was paused

   `Unknown` : unknown status

4. R11. **Get available voices (TTS)**. This is called by the testing application to get a listing of the available voice packs in the current system.

   ```
   void browser.tts.getVoices(function callback)
   ```

   `callback` should be a function that looks like this:

   ```
   function(array of voice objects found){...}
   ```

   Each voice object is a JSON literal with the following properties:

   - name (required)
   - language (optional)
   - gender (optional).

   Example value passed to the callback function:

   ```
   [{name:'US English Female TTS', gender:'female', language:'en-US'},{name:'US Spanish Male
   TTS', gender':male', language:'es-es'}]
   ```

   Empty array indicates no voice packs are available. Undefined or null indicates that an error occurred attempting to get the voices and no information is available.

5. R13. **Pause speech (TTS)**. This is called by the web application to temporarily pause speech. Corresponding events are fired to notify the callback provided in the `speak` function of this.

   ```
   void browser.tts.pause(function callback)
   ```

`callback` is optional. If you specify this parameter, it should be a function that looks like this:

```
function(string state){...}
```

Callback is invoked when pause has occurred. If state is null or undefined, then an error occurred during pause. Otherwise 'pause' or 'error' are the expected states in the callback with error meaning something like there was nothing being spoken to pause.

6. R14. **Resume speech (TTS)**. This is called by the web application to resume speech if it was previously paused.

   ```
   void browser.tts.resume(function callback)
   ```

   `callback` is optional. If you specify this parameter, it should be a function that looks like this:

   ```
   function(string state){...}
   ```

   Callback is invoked when resume has occurred. If state is null or undefined, then an error occurred during resume. Otherwise 'resume' or 'error' are the expected states in the callback, with error meaning something like the speech engine was not in a paused state.

Optional APIs

1. R23. **Empty system clipboard**. The testing application will invoke this to force clear any data that may be in the system clipboard. This is a optional method. The implementer can choose to use the `browser.security.lockDown` to perform the same operation.

   ```
   void browser.security.emptyClipBoard()
   ```

2. R24. **Get Application Start Time**. The testing application will invoke this to determine the time that the application was launched in UTC. This is mainly used to track application uptime. If this is not provided, the web application can track it using local/session storage but it is desirable to have this information natively supported:

   ```
   DateTime browser.settings.appStartTime
   ```

   Example response:

   ```
   "2017-01-15T18:15:30Z "
   ```

3. R40. **Retrieve system MAC address(es)**. The testing application will invoke this to assist in diagnostics. It is difficult to rely on source IP addresses to distinguish between end user machines within our testing servers as firewalls/NATs/Proxies are commonly in use at the schools. The MAC addresses allow us to distinguish end client machines behind a common firewall for diagnostics purposes.

   ```
   void browser.security.getMACAddress(function callback)
   ```

   `callback` should be a function that looks like this:

   ```
   function(array of mac addresses){...}
   ```

   If the array passed into the callback is undefined or null, we were unable to retrieve the MAC addresses.

   Example response:

   ```
   "['00:55:65:C0:00:EA']"
   ```

Audio Recorder (W3C)

NOTE: Browsers supporting W3C's Web Audio API for playback, MediaStream Recording for recording, and Mozilla's MediaDevices.getUserMedia() do not need to implement these functions.

Audio Recorder (Non-W3C)

1. R25. **Initialize audio recorder**. This method is called by the testing application once to initialize the audio API after a page loads. The event listener passed in as argument is used to notify events to caller about progress. Any attempts to call this method when it has already been called should be treated as a reset and reinit.

   `void browser.recorder.initialize (function eventListener)`

   Events expected:

   `INITIALIZING` – indicates that initialization is in progress

   `READY` – Initialization is done and internal data structures are loaded

   `ERROR` – Initialization failed with information on failure cause

2. R26. **Get audio recorder status**. This method is called to enquire about the status of the recorder. Return values are

   `void browser.recorder.getStatus(function callback)`

   `callback` is optional. If you specify this parameter, it should be a function that looks like this:

   `function(status){...}`

   values expected are

   `IDLE` – no recording in progress

   `ACTIVE`- recording in progress

   `INITIALIZING` – initialization in progress

   `ERROR` – terminal error state and reinit is required

   `STOPPING` – recording is done and final book keeping and generation of encoded audio is in progress

   `PLAYING` - recorder is playing back some audio

   `PAUSED` - recorder is paused playing back some audio

3. R27. **Get audio recorder capabilities**. This method is called to enquire about the capabilities of the platform. Throws error if called before initialize is completed successfully.

   `void browser.recorder.getCapabilities(function callback)`

   `callback` should be a function that looks like this:

   `function(capability object literal){...}`

   The object literal returned will have the following values:

`isAvailable` – recording is supported (Boolean)

`supportedInputDevices` – a list of audio input devices detected. Each of these device definitions includes device id, device description/label, supported sample size(s), supported sample rate(s), supported channel count(s), encoding format(s) supported, default input device.

`supportedOutputDevices` – a list of audio output devices detected. Each of these device definitions includes device id, device description/label, supported sample size(s), supported sample rate(s), supported channel count(s), encoding format(s) supported, default output device.

If the object literal returned is null or undefined, we encountered an error.

4. R28. **Initiate audio capture**. This method is called to initiate capture. Throws error if called prior to successful initialization. Throws errors if the options passed in are not supported on the device. Throws error if capture status is currently not IDLE.

   `void browser.recorder.startCapture(options, eventListener)`

   The `options` object includes:

   `captureDevice` – the device id to use for data capture (int)

   `sampleRate` – the line rate to capture the raw audio in (8 kHz, 11 kHz etc.) (specified as int in Hz)

   `channelCount` – 1 (mono), 2 (stereo) … (specified as int)

   `sampleSize` – 8-bit, 16-bit, etc. (specified as int)

   `encodingFormat` – SPX, HE-AAC, Opus, etc. (specified as string)

   `qualityIndicatorDesired` – whether to perform and report a recording quality check or not (Boolean)

   `progressEventFrequency` – how frequently the event listener should be called back to report progress events either based on time or on units of data collected. For example, we could ask for periodic progress events every 2 seconds to keep us notified as recording is happening, or every 30KB of new data collected.

   `captureLimit`– object literal that specifies time or size for the data capture after which the recorder should automatically stop capturing and fire an end event (specified as {duration: 40} or {size:250}, unit for duration is in seconds and for size, is in KB).

   The event listener is passed in to receive capture events. The events include:

   `START` – Capture started

   `INPROGRESS` – Progress event with progress data (34 seconds of audio captured, 36 seconds of audio captured etc or 10KB of audio captured, 30 KB of audio captured etc.)

   `END` – Capture complete. The `END` event is special. This event gives us the pointer to the data collection for the encoded audio. In addition, a quality check is performed on the captured audio stream to evaluate whether it is good or not.

5. R29. **Stop recording**. This method is called to stop audio capture. Throws error if status is currently not "RECORDING".

   `void browser.recorder.stopCapture()`

6. R30. **Retrieve recording**. This method is called to retrieve base64 encoded audio data that was previously captured (or played back by the recorder). If the `END` event for audio capture includes the base64 encoded audio,

then this call is optional. Note: If the event does not include the data, the testing application will be invoking this API directly in the callback for the `END` event.

```
void browser.recorder.retrieveAudio(function callback)
```

`callback` should be a function that looks like this:

```
function(recordedAudio){...}
```

null or undefined implies there was an error retrieving audio.

7. R31. **Playback a recording**. This method is called to play back a recording made through the recorder at some prior time (even in a previous session of the browser) in an asynchronous manner. This API is optional if the browser supports HTML5 webaudio to play back encoded audio (encoded using the format specified in the `startcapture` call) obtained by a call to `retrieveAudio()`. The playback function is passed in the base64 audio string and a callback function.

   ```
   void browser.recorder.play(b64audio, function callback)
   ```

   The callback function is expecting the following events:

   `PLAYBACK_START` - Playback has started. The event includes the id of the audio passed in

   `PLAYBACK_STOPPED` - Playback has stopped (either because the audio stream is done, `pausePlayback()` or `stopPlayback()` has been invoked). The event includes the id of the audio passed in.

8. R32. **Stop playback**. This method is invoked to stop an ongoing audio playback. Throws error if status is currently not "PLAYING".

   ```
   void browser.recorder.stopPlay()
   ```

9. R33. **Pause playback**. This method is invoked to pause an ongoing audio playback. Throws error if status is currently not "PLAYING".

   ```
   void browser.recorder.pausePlay()
   ```

10. R34. **Resume playback**. This method is invoked to resume an already paused audio playback. Throws error if status is currently not "PAUSED".

    ```
    void browser.recorder.resumePlay()
    ```

11. R43. **Retrieve list of audio recordings**. Retrieve a list of all audio recordings.

    ```
    void browser.recorder.retrieveAudioFileList(function callback)
    ```

    `callback` should be a function that looks like this:

    ```
    function(filenames){...}
    ```

    Where `filenames` is an object with the property `files`, containing an array of file names.

    ```
    {'files' : [file1, file2, ...]}
    ```

12. R45. **Retrieve audio file from filename**. Retrieve audio data based on filename obtained from `retrieveAudioFileList()`.

    ```
    void retrieveAudioFile(filename, function callback)
    ```

`callback` should be a function that looks like this:

```
function(b64audio){...}
```

Secure Browser Standards Compliance

Required

1. R35. **HTML5 compliant**. The secure browser must be HTML5 compliant: https://www.w3.org/TR/html5/ and http://html5test.com

2. R37. **CSS3 compliant**. The secure browser must be CSS3 compliant: https://www.w3.org/TR/2014/REC-css-namespaces-3-20140320 and http://css3test.com

Optional

1. R38. **W3C Web Audio compliant**. W3C Web Audio API: https://www.w3.org/TR/webaudio

2. R39. **W3C Web Speech compliant**. W3C Web Speech API: https://dvcs.w3.org/hg/speech-api/raw-file/9a0075d25326/speechapi.html

   Note: As of Firefox v.51, SSML support via Web Speech lacks the features necessary for optimal TDS TTS delivery. Thererfore, support for TTS APIs as described in this document is still necessary.