9. August 2019 Studie · Mobile Sicherheit · Web & Cloud · 11 Min. Lesezeit

Erkennung von Code Injection Schwachstellen in HTML5 Apps

Archiviert. Dieser Inhalt wird nicht mehr aktualisiert. Er bleibt zur Information verfügbar.

Stand 2019-08-09
Quelle https://technology2.a-sit.at/veroeffentlichungen/erkennung-von-code-injection-schwachstellen-in-html5-apps/

Um mobile Cross-Plattform Applikationen zu entwickeln, werden oft Frameworks eingesetzt, die auf JavaScript und HTML5 basieren. Dadurch bringen solche Applikationen jedoch nicht nur den Vorteil der Plattformunabhängigkeit mit sich, sondern auch die Schwachstellen von Browseranwendungen, allem voran die Möglichkeit, potentiell schadhaften Code einzuschleusen. In diesem Bericht wird ein Tool vorgestellt, das die Verwendung der vor Code Injection schützenden Browserfunktion Content Security Policy (CSP) überprüfen und beurteilen kann. Ebenso wird aufgezeigt, wie festgestellt werden kann, ob in eine Applikation, die CSP nicht oder falsch verwendet, potentiell Code über Apache Cordova-Plugins eingeschleust werden kann.

1. Einführung

Smartphones werden primär entweder mit Android oder iOS als Betriebssystem ausgeliefert. Um Applikationen für beide Betriebssysteme zu erstellen, müssen diese in der Regel sowohl für Android als auch für iOS individuell entwickelt werden. Abhilfe schaffen hier Cross-Plattform Frameworks, die es erlauben, die gleiche Anwendung auf beiden Betriebssystemen auszuführen. Viele der Frameworks basieren hierbei auf HTML5 und JavaScript, da diese Technologien auf allen gängigen Plattformen vorhanden sind. Auch das für mobile Cross-Plattform Anwendungen am meisten verwendete Framework [1] Apache Cordova [1] basiert auf Browsertechnologie. Durch die Verwendung dieser Frameworks erben die damit erstellten Applikationen jedoch nicht nur die Vorteile, sondern auch die Nachteile der darunterliegenden Technologien. So stellten Jin et al. [2] fest, dass es möglich ist, potentiell schädlichen Javascript-Code in solche Applikationen 1 https://cordova.apache.org/ © A-SIT, Partner für Sicherheitsfragen einzuschleusen. Im Vergleich zu Webapplikationen kann dieser Code aus einer Vielzahl an Quellen kommen, wie z.B. aus gescannten QR-Codes, auf dem Smartphone gespeicherten Kontakten, oder sogar aus der Auflistung benachbarter WiFi-Access Points.

Cross Plattform-Applikationen haben im Vergleich zu reinen Browseranwendungen zusätzlich oft auch Zugriff auf tieferliegende Funktionen des Systems. Dadurch kann der angerichtete Schaden potentiell höher sein. Deswegen ist ein Schutz vor solchen Angriffen wichtig. Um diese Attacken zu verhindern, wurde die Content Security Policy (CSP) eingeführt. Durch CSP werden bestimmte Funktionen deaktiviert, die die Einschleusung bzw. Ausführung von Code möglich machen. Die CSP ist jedoch konfigurierbar und erlaubt die (teilweise) Aktivierung potentiell gefährlicher Funktionen. Deswegen überprüfen wir in einem ersten Schritt, ob Content Security Policy überhaupt aktiviert ist und welche Einstellungen getroffen wurden. In einem weiteren Schritt wird dann überprüft, ob potentiell Code-ausführende Methoden in der Applikation verwendet werden und ob die Eingabe für diese Funktionen aus Übergabewerten von Cordova-Plugins wie dem QR-Scanner oder dem Kontakte-Plugin stammen.

2. Hintergrund

Die folgenden Abschnitte beleuchten Hintergrundwissen über das Apache Cordova-Framework und die Content Security Policy.

2.1. Apache Cordova

Apache Cordova [3] ist ein Framework, das zur Erstellung von Cross-Plattform Anwendungen verwendet werden kann. Laut AppBrain verwenden beinahe 8% der Android Apps auf Google Play Apache Cordova [1]. Damit ist es das am meisten verwendete Cross-Plattform Framework. Cordova-Applikationen werden in JavaScript, HTML5 und CSS3 geschrieben. Die Benutzeroberfläche basiert dadurch nicht auf nativen Interface-Komponenten der jeweiligen Betriebssysteme, sondern werden innerhalb eines Browsers angezeigt, ähnlich einer mobilen Webseite. Im Vergleich zu Webseiten haben Cordova-Anwendungen jedoch Zugriff auf eine Vielzahl an nativen Funktionen. So gibt es eine Reihe an offiziellen Plugins, die beispielsweise Zugriff auf das Dateisystem, Geräteinformationen, oder Kamera ermöglichen. Zusätzlich zu diesen Erweiterungen können von der Entwicklerin bzw. vom Entwickler auch beliebige weitere Erweiterungen erstellt werden.

2.2. Content Security Policy

Der Inhalt der Content Security Policy kann entweder innerhalb eines HTTP-Header Feldes übertragen oder direkt in der HTML-Datei mithilfe eines Meta-Tags innerhalb des Head-Elements. Da die HTML-Dateien bei Cross-Plattform Apps meist direkt mit der Applikation ausgeliefert werden (um auch offline zu funktionieren), kommt hier in der Regel der Meta-Tag zur Verwendung. Der Meta-Tag sollte hierbei möglichst früh im Dokument vorkommen, da dieser erst nach dem Parsen zur Verwendung kommt und nicht retrospektiv auf bereits geladene Inhalte angewandt wird [4]. Der Meta-Tag mit einer beispielhaften Content Security Policy sieht wie folgt aus:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'
data: gap: https://ssl.gstatic.com 'unsafe-eval'; style-src 'self'
'unsafe-inline'; media-src *; img-src 'self' data: content:;">
```

Um Code Injection Angriffe zu verhindern, erzwingt die Content Security Policy eine striktere Trennung zwischen Code und Layout. Dazu werden einige problematische Funktionen deaktiviert. So ist es mit aktiviert Content Security Policy nicht mehr möglich, JavaScript Code und HTML-Layout in einer Datei zu

mischen. Das heißt es können weder `<script>` -Blöcke mit JavaScript-Code, noch die Event-Handler von HTML-Elementen wie z.B. `onclick` verwendet werden. Ebenso wird die dynamische Evaluierung von JavaScript-Code deaktiviert. Dadurch können Funktionen wie `eval()`, die einen String mit JavaScript-Code als Parameter nehmen, nicht mehr ausgeführt werden. Die Content Security Policy erlaubt zwar, die problematischen Funktionen durch Direktiven wie `unsafe- eval` oder `unsafe-inline` wieder zu aktivieren, dadurch sinkt jedoch die Schutzwirkung der Content Security Policy, vor allem wenn die betreffenden Funktionen global aktiviert werden.

3. Evaluierung der Content Security Policy

In diesem Schritt wird überprüft, ob eine App Content Security Policy einsetzt, um Code Injection Angriffe zu verhindern. Dazu wird als Grundlage das Android Package (die .apk-Datei) der Applikation verwendet. Im darin befindlichen Ordner `assets/www` werden alle HTML-Dateien untersucht. Dazu werden diese mit Hilfe eines HTML-Parsers geparst und auf das Vorhandensein des Meta-Tags geprüft. Ist dieser nicht vorhanden, so ist die App potentiell anfällig für Code Injection-Attacken. Ist der Meta-Tag vorhanden, so wird die darin enthaltene Content Security Policy in einem weiteren Schritt evaluiert. Sollte hierbei eine unsichere Konfiguration festgestellt werden, so ist eine Applikation trotz Vorhandensein einer Content Security Policy potentiell anfällig für Code Injection-Attacken. In diesen Fällen kann die App im nächsten Schritt auf potentielle Quellen für eingeschleusten Code untersucht werden.

4. Evaluierung von Code Injection Schwachstellen

Um potentielle Code Injection Schwachstellen in Programmen zu erkennen, wird eine statische Code-Analyse durchgeführt. Dazu wird nur das Android Package mit dem darin enthaltenen Quellcode benötigt. In einem ersten Schritt wird mithilfe des WALA-Frameworks [5] und der DASCA-Erweiterung [6] ein Call Graph erstellt. Dieser Call Graph stellt die Aufrufbeziehungen zwischen den verschiedenen Methoden des untersuchten Programms dar. Bei jedem Node handelt es sich um eine Funktion der untersuchten Applikation. Diese Nodes bzw. Funktionen lassen sich weiter in sogenannte Basic Blocks unterteilen, welche eine Anzahl an nicht zu unterteilenden Instruktionen enthält.

Auf Basis des erstellen Call Graphs wird nach potentiellen Code Injection Schwachstellen gesucht. Dazu werden die zu den einzelnen Nodes des Call Graphs gehörenden Basic Blocks auf den Aufruf sogenannter Sinks untersucht. Bei diesen Sinks handelt es sich in unserer Analyse um Funktionen, die Code dynamisch auswerten können. Sollte eingeschleuster Code diese Funktionen erreichen, so würde dieser durch die Sinks interpretiert und ausgeführt werden. Die vollständige Liste der derzeit untersuchten Sinks befindet sich in Tabelle 1. Ausgehend von den gefundenen Sinks wird Backtracking verwendet, um mögliche Quellen für die Einschleusung von Quellcode zu finden. Hierbei wird nach der für Cordova-Apps spezifischen Quelle der Plugins gesucht. Dazu wird von den Sink-Aufrufen aus rückwärtsgehend nach der Quelle der Parameter der Sinks gesucht. Hierbei werden die einzelnen Basic Blocks des untersuchten Nodes rückwärts abgewandert. Für jeden Parameter der Sink-Funktion werden während dem Rückwärtsgehen Zuweisungen und Veränderungen der untersuchten Variablen überwacht. Die in den Befehlen verwendeten Variablen werden dabei als potentielle Quellen markiert.

Tabelle 1 Untersuchte Sinks

innerHTML	outerHTML
writeln	Write
html	Append
prepand	Before
after	replaceAll
replaceWith	eval

Ist der Algorithmus am Beginn einer Funktion angelangt, so wird überprüft, ob Parameter der untersuchten Funktion als potentielle Quelle markiert sind. Ist dies der Fall, so werden auch alle aufrufenden Funktionen untersucht. Diese werden mit Hilfe des Call Graphs identifiziert. Ausgehend von der Instruktion des Aufrufs werden wieder die einzelnen Basic Blocks nach dem oben beschriebenen Schema untersucht. Dies ist solange der Fall, bis die Quelle(n) der verwendeten Parameter ausfindig gemacht wurden. Sollte es sich hierbei um das Ergebnis eines Cordova-Plugin Aufrufes handeln, so wird dieser als potentielle Quelle für eingeschleusten Code für eine weitere manuelle Inspektion markiert.

Zur Überprüfung der Funktionalität wurde eine Demo-App erstellt, welche anfällig für Code Injection ist. Anhand dieser Applikation werden im folgenden einige Beispiele präsentiert, welche die Funktionsweise des Tools veranschaulichen sollen. Dazu betrachten wir im Folgenden die Funktion namens `vulnerableMethod` :

```
var methods = {
vulnerableMethod: function(parameter1, parameter2) {
var string = "Got values: ";
var htmlContent = string + parameter1 + parameter2;
document.getElementById("element-id").innerHTML = htmlContent;
},
};
```

Die Methode erhält zwei Parameter, welche in weiterer Folge zusammen mit einer lokalen Variable verbunden werden. Danach wird die daraus sich ergebene Variable einem Element mit Hilfe von `innerHTML` hinzugefügt. Durch die Verwendung von `innerHTML` wird potentiell enthaltener Code nach dem Hinzufügen sofort ausgeführt. In der von WALA verwendeten Repräsentation sieht diese Funktion wie folgt aus:

```
   Instruktionen:

   BB0 BB1 0  v5 = new <JavaScriptLoader,LArray>@0   [5=[arguments]] 1  v8 =
global:global $$undefined      [8=[string]] 3  v11 = global:global $$undefined
[11=[htmlContent]] 6  v15 = binaryop(add) v13:#Got values:, v3index.js [13=
[string]3=[parameter1]] 7  v14 = binaryop(add) v15, v4    [14=[htmlContent]4=
[parameter2] 11  v21 = global:global document     [21=[$$destructure$rcvr12]] 12
check v21               [21=[$$destructure$rcvr12]] BB2 15  v23 = dispatch
v22:#getElementById@15 v21,v24:#element-id exception:v25index.js [22=
[$$destructure$elt12]21=[$$destructure$rcvr12]] BB3 16  fieldref v23.v26:#innerHTML
= v14 = v14 index.js [14=[htmlContent]] BB4
```

Die Methode wurde in fünf Basic Blocks (BB) aufgeteilt, wobei der erste und letzte per Konvention jeweils leer ist. Ausgehend von Basic Block 0 untersucht das Tool jede Instruktion auf Verwendung eines der definierten Sinks. Dazu wird über jede Instruktion iteriert. Im Beispiel wird ein solcher Sink in Basic Block 3 bei Instruktion 16 gefunden. Ebenso kann hier festgestellt werden, dass es sich bei der verwendeten Variable um Variable 14 (`htmlContent`) handelt. Von dieser Variable wollen wir nun wissen, ob diese (zum Teil) durch Cordova-Plugins beeinflusst sind.

Ausgehend von Instruktion 16 wird Backtracking durchgeführt. Da es sich bei Instruktion 16 um die erste Instruktion des Basic Blocks handelt, wird die Untersuchung bei den Vorgängerblöcken ausgeführt. Da die untersuchte Funktion keine Verzweigungen aufweist, gibt es im Beispiel nur einen Vorgängerblock, den Basic Block 2. Dieser beinhaltet eine Instruktion, welche jedoch die untersuchte Variable weder verwendet noch modifiziert. Deswegen wird wieder mit dem Vorgängerblock fortgefahren, in diesem Fall Basic Block 1. Ausgehend von der letzten Instruktion (Instruktion 12) des Basic Blocks wird nach Modifikationen an der untersuchten Variable gesucht. Eine solche Modifikation findet in Instruktion 7 statt. Hier wird der untersuchten Variable 14 der Wert von Variable 4 (`parameter2`) und der nicht benannten Variable 15 zugewiesen. Diese beiden Variablen 4 und 15 ersetzen nun Variable 14, da es sich um eine Zuweisung handelt. In der nächsten Instruktion (Instruktion 6) wird der untersuchten Variable 15 der Wert von Variable 13 (`string`) und Variable 3 (`parameter1`) zugewiesen. Dadurch kann Variable 15 durch Variable 13 und 3 ersetzt werden. Bei den weiteren Instruktionen wird keine der untersuchten Variablen modifiziert. Nach Erreichen des Basic Blocks 0 kann festgestellt werden, dass es sich bei zwei der drei untersuchten Variablen um Parameter der Funktion handelt. Deswegen müssen auch alle aufrufenden Funktionen nach dem dargelegten Prinzip untersucht werden um die Quelle für die Parameter zu finden.

Um die Quelle der Parameter zu finden, muss vorher festgestellt werden, ob eine Funktion als Callback eines Cordova Plugin-Aufrufes verwendet wird. Dazu werden alle in der Applikation vorkommenden Aufrufe von Cordova Plugins untersucht. Dabei werden von allen Aufrufen von Plugin-Methoden aus in den aufgerufenen Methoden nach dem Aufruf der Cordova-Exec Funktion gesucht. Diese Funktion ruft in weiterer Folge den nativen Teil des Plugins auf, der die erweiterten Funktionen bereitstellt. Ausgehend von der Exec-Funktion wird untersucht, welche Funktion als Callback angegeben wurde. Dazu wird ähnlich zum vorhin beschriebenen Algorithmus Backtracking verwendet, um die Zuweisung der Callback Funktion zu erkennen. Sollte es sich hierbei um eine Funktion handeln, deren Parameter von einem Sink verwendet werden, so wurde eine potentielle Code Injection-Schwachstelle gefunden.

5. Einschränkungen

Die erstellten Tools unterstützen nur die Inspizierung von lokalem Code, der sich bei Auslieferung im Android Package befindet. Sollte weiterer Code während der Ausführung nachgeladen werden, so kann dieser nicht untersucht werden.

Das Tool zur Erkennung von Code Injection-Schwachstellen kann nicht erkennen, ob ein Wert der von einem Cordova-Plugin stammt, wirklich durch den Nutzer beeinflussbar ist. Dadurch werden potentiell Aufrufe markiert, die jedoch in Wirklichkeit keine Quelle für Schadcode sind. Weiters wird angenommen, dass die Plugins die Werte per Callback an die Applikation liefern. Sollte sich ein Plugin nicht an diese übliche Vorgehensweise halten, so werden potentielle Schwachstellen nicht erkannt. Ebenso kann es dazu kommen, dass bestimmte Quellcode-Konstrukte bzw. Bibliotheken beim Backtracking nicht unterstützt werden und es dadurch zu Ungenauigkeiten bei der Evaluierung kommt. Etwaige Einschränkungen der verwendeten Bibliotheken [6] treffen ebenso zu.

6. Fazit

Cross-Plattform Frameworks sind nützlich, um Applikationen für verschieden Betriebssysteme zu entwickeln. Da viele dieser Frameworks auf Browsertechnologien aufbauen, sind die erstellten Applikationen jedoch auch für Code Injection anfällig. Um Applikationen davor zu schützen, sind bei der Entwicklung von Cross-Plattform Applikationen verschiedene Vorkehrungen von Seiten der EntwicklerInnen nötig. So sollte Content Security Policy korrekt eingesetzt werden beziehungsweise verhindert werden, dass NutzerInnen-Daten potentiell gefährliche Funktionen erreichen. Um zu überprüfen, ob diese Maßnahmen korrekt umgesetzt wurden, wurden zwei Tools erstellt, die potentielle Schwachstellen bei der Umsetzung der Vorkehrungen aufzeigen können.

Referenzen

[1] AppTornado GmbH, „PhoneGap / Apache Cordova - Android SDK statistics | AppBrain," 05 08 2019. [Online]. Available: https://www.appbrain.com/stats/libraries/details/phonegap/phonegap-apache-cordova. [Zugriff am 06 08 2019].

[2] X. Jin, X. Hu, K. Ying, W. Du, H. Yin und G. N. Peri, „Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation," ACM Conference on Computer and Communications Security, 2014.

[3] The Apache Software Foundation, „Apache Cordova," [Online]. Available:

https://cordova.apache.org/. [Zugriff am 06 08 2019].

[4] „Content Security Policy Level 2," W3C, 15 12 2016. [Online]. Available:

https://www.w3.org/TR/CSP2/#delivery-html-meta-element. [Zugriff am 04 07 2019].

[5] IBM, „Wala - T.J. Watson libraries for analysis," [Online]. Available:

http://wala.sourceforge.net/wiki/index.php/Main_Page. [Zugriff am 08 08 2019].

[6] A. D. Brucker und M. Herzberg, „On the Static Analysis of Hybrid Mobile Apps: A Report on the State of Apache Cordova Nation," in *International Symposium on Engineering Secure Software and Systems*, 2016.