No Escape from Reality: Security and Privacy of Augmented Reality Browsers

Richard McPherson
University of Texas at Austin
richard@cs.utexas.edu

Suman Jana
University of Texas at Austin
suman@cs.utexas.edu

Vitaly Shmatikov
University of Texas at Austin
shmat@cs.utexas.edu

ABSTRACT

Augmented reality (AR) browsers are an emerging category of mobile applications that add interactive virtual objects to the user's view of the physical world. This paper gives the first system-level evaluation of their security and privacy properties.

We start by analyzing the functional requirements that AR browsers must support in order to present AR content. We then investigate the security architecture of Junaio, Layar, and Wikitude browsers, which are running today on over 30 million mobile devices, and identify new categories of security and privacy vulnerabilities unique to AR browsers. Finally, we provide the first engineering guidelines for securely implementing AR functionality.

1 Introduction

Augmented reality (AR) technologies enhance users' perception of the world by blending interactive virtual objects with the visual representation of actual objects in real time [2, 3]. Traditional AR applications range from medical visualization to aircraft navigation, but only recently have consumer mobile devices become sufficiently powerful to run AR software.

AR applications have three stages: sensing input, transforming sensed objects (e.g., adding virtual objects), and rendering the transformed objects to the user. Modern AR platforms ease the burden of implementing these tasks. By far the most popular platforms are *AR browsers* like Junaio, Layar, and Wikitude, available as SDKs or standalone mobile apps. Junaio has more than 20 million users and over 20,000 content developers who have created more than 210,000 AR "channels" [14]. Layar has 1.5 million users and 9,000 content developers [18]. Wikitude has 13 million users [29] and over 30,000 content developers.

All existing AR browsers are based on Web browsers and are similar to them in the sense that they, too, fetch and display interactive content from websites ("channels," in AR parlance). In addition to rendering HTML and executing JavaScript, AR browsers provide support for the three key tasks necessary for AR functionality: sensing, transforming, and displaying transformed objects. They enable AR channels to (1) access sensors on the mobile device, including the onboard camera and GPS location, (2) create and manipulate a variety of 2D and 3D interactive virtual objects, and (3) display virtual objects on top of the camera feed, realistically blending them with real objects. The resulting AR content combines image recognition, geolocation, interactive virtual objects, conventional Web content, and control code written in JavaScript (see an example in Fig. 1).

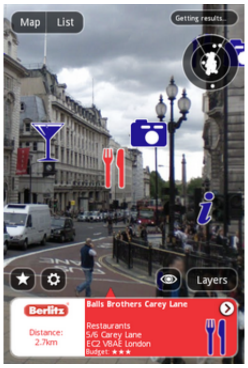


Figure 1: A Layar-based mobile app [7].

The basic architecture of AR services is shown in Fig. 2. From the security and privacy perspective, its key aspect is that the AR browsers provide augmentation mechanisms, but the actual AR content comes from channels created by independent developers. Just like a conventional Web browser is an interface between the user and Web content from independent websites, an AR browser is an interface between the user and independent AR content. An AR browser is thus responsible for ensuring that malicious AR content cannot access content from other sources, nor damage or abuse the user's system outside the browser.

A major difference between Web browsers and AR browsers is the business model. Web browsers are typically part of the standard software distribution, and their developers are paid by the licensing fees from OEMs and OS owners and by the search engines. This model works because there is already a wealth of Web content. AR browsers, however, need a different model because there is not much AR content available today. Their sources of revenue include advertising injected into AR content, registration fees from content developers, and revenue sharing for paid content. This business model has an impact on the architecture of AR services: unlike Web content, which is accessed directly from the Web browser, requests to load third-party AR content must go through the AR service provider, as shown in Fig. 2.

Our contributions. We perform the first systematic analysis of the security and privacy properties of AR browsers and how they differ from Web browsers. Untrusted AR content presents new, unique types of threats, yet—in contrast to Web-browser specifications—the latest Augmented Reality Markup Language (ARML) specification [19] barely mentions security or privacy, and they are often overlooked in the design of the existing AR browsers.

WWW 2015, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3469-3/15/05.
http://dx.doi.org/10.1145/2736277.2741657.

We start by analyzing the **functional requirements** needed to support the sensing, transforming, and rendering of AR content. These include new ways of combining AR objects and conventional HTML content from multiple origins, new APIs for accessing objects outside the browser, new mechanisms for controlling the display of AR and HTML objects, and new ways of launching content.

Then, for each functional requirement, we investigate how it is implemented by the existing AR browsers. All AR browsers are based on Web browsers, which do not support AR functionality, forcing AR browsers to resort to ad-hoc cross-origin mechanisms, APIs that open holes in the browser sandbox, custom techniques for composing visual content from different origins, and non-standard delegation schemes for authentication credentials.

Architectural flaws in these mechanisms result in security and privacy vulnerabilities. We explore the threat model of AR browsers and demonstrate several **new categories of threats** caused by the AR browsers' unique combination of high-volume visual data gathering, image-triggered code execution, outsourced image processing, and merging images from the onboard camera with third-party content. For example, individual-specific items such as license plates can automatically launch malicious AR content, enabling fully automated stalking and tracking; malicious AR channels can abuse image-triggered code execution; and a conventional webpage can hijack the AR browser installed on the user's mobile device and use it to gain unauthorized access to the device's camera and GPS without the user's permission. We also show AR browsers amplify existing threats such as cross-site scripting, clickjacking, cookie stealing, and leakage of private information.

For each design flaw, we present our **recommendations**. Some are easy to fix, others require a substantial re-design, but none are mere "bugs." They all stem from the fact that standard system components used in today's mobile and Web applications are insufficient to securely support AR functionality. For each functional requirement of the AR browsers, we explain which features and system abstractions are needed to implement it properly

2 AR Services

AR services are deployed by AR *service providers* such as Junaio and Layar. These companies supply AR client software (we use the term *AR browser*) to users and maintain dedicated AR servers through which users access third-party AR content (see Fig. 2). AR *content providers* are independent developers who create AR content, host it on their own servers, and register this content with AR service providers. We use the term *channel* generically for any AR content, but the actual terminology differs from service to service (e.g., channels are called *layers* in Layar).

By analogy with conventional Web, AR service providers are similar to Web-browser developers, while AR channels are similar to Web applications. There are important differences, however. AR providers make money by charging for SDK licenses, features such as cloud storage for AR channels, and per-user fees from third-party apps that connect to their services. All providers analyzed in this paper allow a limited use of free channels, but some charge for commercial channels and/or may insert banner ads into free channels. Therefore, they typically require that browsers initiate access to third-party channels via providers' own servers.

2.1 Functional requirements

Access to native resources on the user's device. The AR browser must have access to the onboard camera and GPS location to recognize images and locations that launch AR content, and to correctly add AR objects to the camera feed.

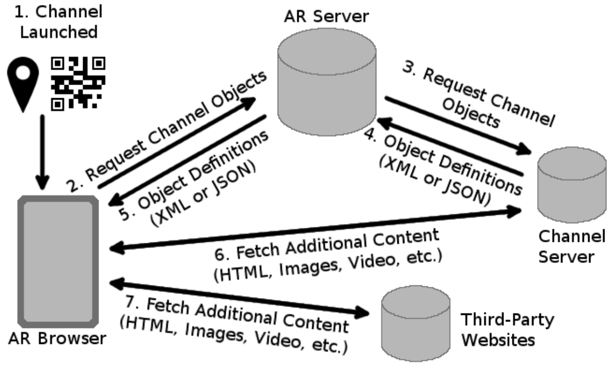

1. Channel Launched
AR Server
2. Request Channel Objects
5. Object Definitions (XML or JSON)
3. Request Channel Objects
4. Object Definitions (XML or JSON)
6. Fetch Additional Content (HTML, Images, Video, etc.)
Channel Server
7. Fetch Additional Content (HTML, Images, Video, etc.)
AR Browser
Third-Party Websites

Figure 2: Architecture of a typical AR service.

Support for interactive, non-HTML AR content. In addition to HTML content such as images and text, AR content may include 2D and 3D models and animations that cannot be described in HTML. AR channels thus include service-specific XML or JSON defining how to place and render these objects.

Image-triggered code execution. AR browsers access content in non-standard ways: they send images from the device's camera to their servers, which attempt to recognize certain pictures and QR codes and automatically launch the associated AR channels.

Outsourced image processing. Image recognition is a computationally heavy task that may not be feasible on low-powered mobile devices and often involves proprietary algorithms. Furthermore, image-based code execution requires the server to extract the trigger image from the camera feed and match it against a proprietary database of registered images. Therefore, AR browsers send images from the phone's camera to the AR provider for processing.

Visual composition of AR content. The AR browser is responsible for constructing a visual stack that combines non-HTML AR content, such as interactive 3D models, with HTML content from multiple origins (e.g., online ads) on top of the camera feed.

Indirect retrieval of AR content. Instead of directly fetching AR content from its developers, AR browsers typically submit requests via the AR provider's server. This enables providers to charge fees for registration and usage, inject advertising, etc.

2.2 Components of AR services

AR browsers. Fig. 3 shows the generic architecture of an AR browser, including (1) one or more instances of an embedded Web browser such as WebView, (2) a "native" component with direct access to OS-managed resources such as the camera and GPS location, and (3) ad-hoc mechanisms for gluing these pieces together.

AR channels. An AR channel is roughly similar to a website. It defines an augmented reality experience by specifying AR content to display and how to display it. This content may include AR objects linked to a geolocation ("points of interest" or POI), HTML pages, audio, video, etc., as well as JavaScript to control these objects. The channel may also specify the actions to take when a certain object comes into view or is clicked by the user.

For example, an AR channel may overlay historical pictures when viewing landmarks,[1] show reviews for nearby restaurants,[2] or control an avatar running around the scene.[3] A channel may directly incorporate third-party content—for instance, include online ads in

[1] `http://layar.it/YuDzik`

[2] Wikitude Restaurants

[3] `junaio://channels/?id=127275`

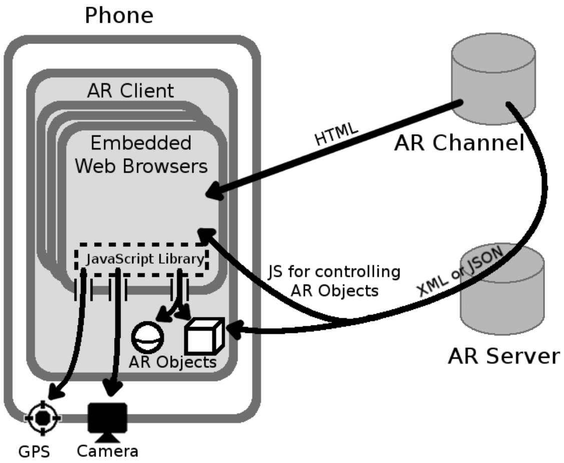


Figure 3: Architecture of an AR browser.

its HTML—or instruct the AR browser to load a third-party webpage when the user performs a certain action.

A user launches a channel by selecting it from a list provided by the AR browser (based on the geolocation or most popular channels) or by scanning an image.

AR servers. As explained in Section 2.1, requests to load a channel are sent by the AR browser to the AR provider's server, not directly to the channel server (see Fig. 2). Each request includes some combination of the channel's id, the location of the device, and other data. The AR server forwards the request to a server that the channel owner registered with the AR provider. The AR server may also handle the authentication of users to channels (Section 9). The response from the channel with the XML or JSON definitions of AR objects is forwarded via the AR server, too. Subsequent requests may be sent by the browser directly to the channel server.

2.3 Specific AR browsers

We focus on the most popular AR browsers. **Junaio** is an AR browser developed by Metaio to augment both print media and geolocation-based environments (Fig. 4). **Layar** focuses primarily on adding AR features to print media such as magazines and newspapers (Fig. 5), but also supports geolocation-based AR. AR content for Layar is served by *layers*, but we will refer to them as *channels* for terminological consistency. **Wikitude** is another AR browser, but some of its features did not execute correctly in our testing, thus we discuss only the features we were able to evaluate.

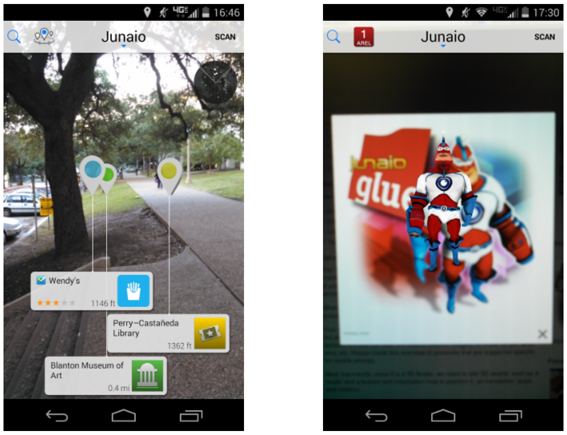
Unlike HTML, AR content is browser-specific, i.e., a Junaio browser can only display Junaio channels. Augmented Reality Markup Language (ARML) is a proposed standard that unifies the XML format of AR objects [19].

3 Threat Model

We are concerned with five classes of attackers.

AR attackers. Just like a standard Web attacker, an AR attacker controls the malicious content of his AR channel and may trick or entice users into visiting it. He cannot observe users' network communications with other destinations, nor execute any code on their machines other than JavaScript served by his own channel.

Unlike conventional Web browsers, AR browsers automatically launch a channel whenever they scan a picture or QR code associated with it. This introduces a new attack vector: since the attacker can choose any image for his channel, he can trick users' browsers into automatically launching malicious content by placing this image in a public place (e.g., as a sticker on a wall).



(a) *(b)*

Figure 4: 4a is a Junaio channel showing nearby places of interest. 4b is a channel showing a 3D model placed over the Junaio logo.

Ad attackers. AR channels can include third-party content such as syndicated ads. An "ad attacker" tricks a trusted website or AR channel into incorporating his malicious content, e.g., via ad brokers. We assume that ads can run arbitrary JavaScript, but are confined into iframes when rendered by the AR browser.

Web attackers. The focus of this paper is on malicious AR content, but we also investigate how the mere presence of AR browsers on the device can be exploited by conventional Web attackers (Section 4.2). A Web attacker controls his own website (but not the network) and may lure users to it via enticing content, ads, etc.

Curious AR services. We assume that AR browsers are benign (the issues raised by malicious mobile apps are well beyond our scope), but we do investigate privacy risks caused by user-specific visual data collected by AR services.

Network attackers. We briefly analyze privacy threats posed by network attackers. Either through man-in-the-middle attacks or by being on the same network as the victim, a network attacker can listen in on the communications between the AR browser and the AR provider, AR channel owners, and third-party servers.

4 Out-of-sandbox Native Access

AR browsers cannot function without access to the camera and GPS location. Both are required to launch AR channels and to correctly add AR objects to the camera feed. Consequently, all AR browsers equip JavaScript with some form of access to native device resources outside the browser. Script access to AR objects is also required by the ARML 2.0 specification [19, Section 9.1].

These custom APIs effectively open holes in the Web-browser sandbox, intended to support native access by the channel's own JavaScript. Unfortunately, the WebView embedded browser where this JavaScript is executed does not provide any way to restrict access to these APIs. Consequently, they can be accessed by any Web content regardless of its origin.

Another common functionality is launching AR browsers via custom URLs. This is needed for interoperability [21, Section 5]: for example, one AR browser may launch another AR browser to render proprietary content that is not supported by the first browser.

4.1 Doing it wrong

The control code of Junaio channels is written in JavaScript and executed in an embedded WebView. This WebView is extended with custom APIs for accessing the camera, reading and changing the Junaio-reported geolocation, controlling the device's light, making

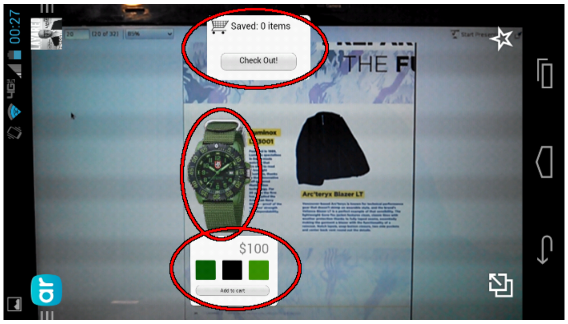


Figure 5: A Layar channel running on a scanned magazine page. The AR objects are circled. Clicking any color below the 3D watch model changes its color. The user can also add the watch to his or her shopping cart.

requests to the channel server, opening conventional Web-browser windows, or loading a different channel.

These APIs are accessed via AREL, a JavaScript library that encodes commands in pseudo-URIs. For example, *arel://media/website/?action=open&external=true&url=http%3a%2f%2f www.google.com* instructs the Junaio app to launch Google.com in a conventional browser. To pass this pseudo-URI from WebView to the Junaio app, the channel's Web code pushes it to the global "commandQueue" and sets *window.location* to *"arel://requests Pending"*. The Junaio app intercepts the URL load event, reads the command off the queue, and performs the requested action. In Junaio on Android, however, any content—regardless of its origin and even if running inside an iframe—can bypass AREL and execute native commands directly, without user permission, by setting *window.location* to the corresponding pseudo-URI.

4.2 Risks

In this section, we are concerned with (1) conventional Web attackers, whose malicious pages are viewed by mobile users in conventional browsers, (2) "ad attackers," whose untrusted HTML is incorporated into trusted AR channels but confined into iframes, and (3) AR attackers who directly control malicious AR channels.

Conventional Web content breaking out of the sandbox. Conventional webpages cannot access the camera or other native resources outside the browser unless explicitly authorized by the user. Unfortunately, the AR browser's access rights can be hijacked by malicious webpages to gain this access without involving the user.

Suppose the user has the Junaio app installed on his Android phone. The user accidentally visits a malicious webpage in a *regular Web browser* (e.g., Android's default system browser) by clicking on an ad, a link in a spam message, etc. The malicious page contains a URL of the form *junaio://channel*=... and a script in the page forces the browser to open this URL. This generates an Android intent, which automatically starts the Junaio app and launches any channel chosen by the attacker, e.g., the attacker's own channel. Like all Junaio channels, the attacker's channel automatically has access to the device's camera, can take pictures of the user and its surroundings, etc. Layar, too, can be automatically launched from a conventional webpage via a pseudo-URI.

This attack completely bypasses OS access control. Even though the user granted camera access only to Junaio or Layar, this access has now been hijacked by a conventional webpage. The attack can even be stealthy. Having read images from the camera, the attacker's channel can relaunch the regular Web browser and immediately redirect the user to the page he was initially browsing.

This vulnerability is *generic* because the ability to automatically launch an AR browser is required for interoperability [21]. The presence of a single AR browser on the user's device can thus be exploited by any conventional webpage to bypass user permissions.

Malicious ads breaking out of the sandbox. Because native-access rights are not restricted to the channel's own origin, any iframe can hijack them. In Section 5.2, we describe how native-access capabilities can be used by a malicious ad to perform a cross-site scripting attack against any origin of its choosing.

Furthermore, malicious third-party iframes included into a trusted channel can redirect the entire AR browser to a malicious channel. For example, in Layar, a script in an iframe can use a *layar://* command to switch the browser to a different channel. In Junaio, the switchChannel command in AREL (also accessible from an iframe) has the same effect. This can be exploited for undetectable phishing: a malicious iframe can automatically switch the browser to a visually indistinguishable malicious channel.

Malicious AR content abusing native access. The ability of AR channels to access resources outside the browser sandbox presents privacy risks to their users. In Junaio, as long as the channel's transparent overlay (Section 5.1) continues to run in the background, it can surreptitiously grab images from the camera and send them to the channel server even after the user moved away from the place where he launched the channel. The user's location can be tracked in a similar fashion in Junaio, Layar, and Wikitude.

4.3 How to do it right

Interfaces to native resources must be protected by origin-based access control, lest they are hijacked by untrusted iframes. Recent solutions to the problem of unauthorized native access by third-party origins in mobile apps, e.g., NoFrak by Georgiev et al. [9], may be applicable to AR browsers. Furthermore, AR browsers should be re-designed to support fine-grained native-access permissions. For example, instead of unfettered access to a camera, the channel would be restricted to accessing it only via pre-defined system abstractions such as "recognizers" [12] for specific objects.

To prevent conventional webpages from gaining unauthorized access to the camera and other resources by launching the AR browser and directing it to the attacker's channel, the user should be asked for confirmation whenever the AR browser is invoked automatically (this presents interface-design and usability challenges).

5 Support for Non-HTML AR Content

In addition to HTML content such as images and text, interactive AR content includes videos, animations, and 2D and 3D models with unique visual presentation requirements. These AR objects cannot be described in HTML alone, thus AR services rely on XML or JSON definitions to specify how to place and render these objects, and on JavaScript to control these objects at runtime.

Just like conventional websites, AR channels may combine content from different origins. AR browsers must therefore confine untrusted content. In conventional Web browsers, the same-origin policy (SOP) ensures that content from a given origin—defined by the protocol (HTTP or HTTPS), domain name, and port number—cannot access the non-trivial attributes of any content from a different origin [27]. Web browsers also provide origin-based isolation mechanisms such as iframes and structured cross-origin communication mechanisms such as postMessage.

In AR browsers, interactive, non-HTML AR objects make the confinement problem much harder because these objects must be described in XML or JSON, which are not governed by the SOP. Therefore, the AR browser cannot rely on the underlying Web browser to provide isolation between origins.

5.1 Doing it wrong

Junaio. In Junaio, AR objects are defined in an XML page returned by the channel server. Junaio supports floating clickable

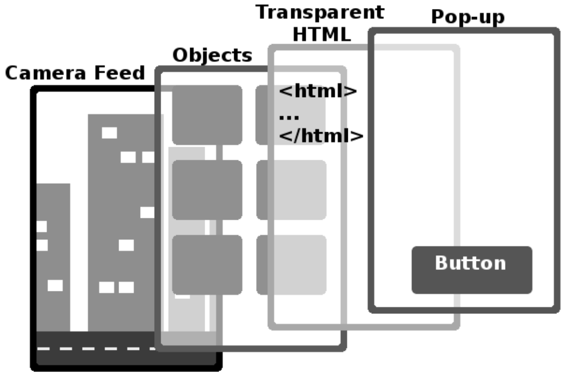


Figure 6: Junaio's visual stack. AR objects are on top of the camera feed, the transparent overlay on top of the objects. If an object is clicked, a popup appears at the very top.

objects ("points of interest"), 3D models, floating pictures, movies, and 360-degree panoramas (Figs. 4a and 4b). The Junaio browser renders these objects in the visual stack shown in Fig. 6.

On top of the AR objects, Junaio places a transparent window implemented using WebView (Android) or UIWebView (iOS). We call it the *transparent HTML overlay*. This overlay provides GUI functionality to channels and enables them to control AR objects outside WebView via special browser interfaces and a custom JavaScript library called AREL (Section 4.1). These interfaces can be used to create, destroy, animate, move, or resize AR objects, to read and modify their parameters such as id, name, geolocation, and the associated popup, and to handle events based on channel state, object state, or user's interaction with the object (e.g., channel ready, object loaded, sound finished playing, object rotated).

The URL of the transparent overlay is specified in the XML page and may belong to a different origin than the AR channel itself. This URL cannot be viewed by the user. The channel—and any third-party content included in the channel—can also supply JavaScript that will be executed inside the transparent page.

Clicking a link in the transparent overlay loads the destination in the same window, replacing the old page. JavaScript in the transparent overlay can also open an opaque window with a conventional embedded Web browser. Another way to open an opaque window is via a popup (Section 5.2). JavaScript continues to run in the background after opening the window.

Layar. The Layar browser displays AR objects on top of the visual feed from the device's camera (Fig. 7). The objects can be HTML webpages, 2D images, 3D models, or videos, and can have actions associated with them, such as placing a phone call, sending an SMS or email, launching a website, loading or refreshing channels, sharing the channel on Facebook or Twitter, and loading movies and music. Actions are specified in the object definition via pseudo-URIs such as 'tel:', 'sms:', 'mailto:', 'layar://' , 'layarshare://'.

Wikitude. Wikitude is architecturally similar to Junaio. AR content includes a transparent webpage that shows a GUI and controls AR objects via a custom JavaScript library called ARchitect. Object types include *HtmlDrawable*, intended to display HTML content. HtmlDrawables have an evalJavaScript function that can be used to execute JavaScript inside a drawable (it worked only sporadically in our testing on Android 4.4.2).

5.2 Risks

In this section, we are concerned with AR attackers, who may incorporate trusted content into their malicious AR channels, and "ad attackers," whose malicious content (e.g., online ads) is incorporated into trusted AR channels but confined into iframes.

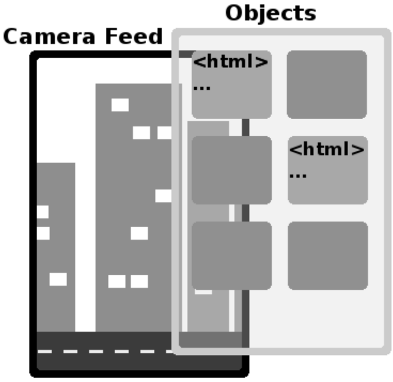


Figure 7: Layar's visual stack. AR objects, which can include HTML pages, are overlaid on the camera feed.

Cross-site scripting. The XML definition of an AR object in Junaio can have a *popup* field with a textual description and an array of buttons. When such an object is clicked, a partially transparent window with the popup's description and buttons is opened on top of the transparent overlay (Fig. 6). Each button contains either a URL, or JavaScript code. When a button is clicked, the associated URL is loaded in an opaque window. If the button contains JavaScript, it is executed in the transparent overlay—even if the origin of the content in the overlay is different from the origin of the channel that provided the script.

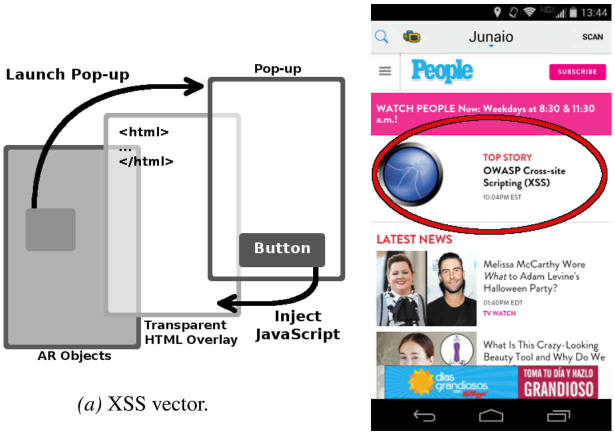
This setup opens a hole in the same-origin policy. A malicious channel can specify any origin for the transparent page and associate an arbitrary script with a button. When the button is clicked, this script is injected into the transparent page and gains unrestricted access to all content from this page's origin—see Fig. 8a. This cross-site scripting (XSS) vulnerability can be exploited, for example, to modify the victim's DOM (see Fig. 8b) or steal cookies.

HtmlDrawable objects in Wikitude contain an even simpler XSS vulnerability. A malicious channel can specify any URL for an HtmlDrawable object and use evalJavaScript to inject an arbitrary script into this object.

Universal cross-site scripting. The above XSS attacks assume that the channel is malicious. Unfortunately, even if (1) the channel itself is benign, (2) all untrusted, third-party content, such as online ads, is correctly confined to iframes, and (3) the embedded Web browser running the channel's HTML correctly enforces the SOP, confined third-party content in Junaio can perform XSS attacks against *any* origin of its choosing.

Consider a benign Junaio channel that includes an AR object with a popup button and suppose that the channel's transparent HTML page contains an ad in an iframe (Fig. 9a). Malicious JavaScript hidden in such an ad can (1) use AREL commands (Section 4.1) to change the script associated with the popup, and (2) change the URL of the transparent overlay to the victim page (Fig. 9b). When the button is clicked, the attack script is executed in the victim page (Fig. 9c). This is a *universal XSS vulnerability*: a malicious ad can inject an arbitrary script into any origin whatsoever.

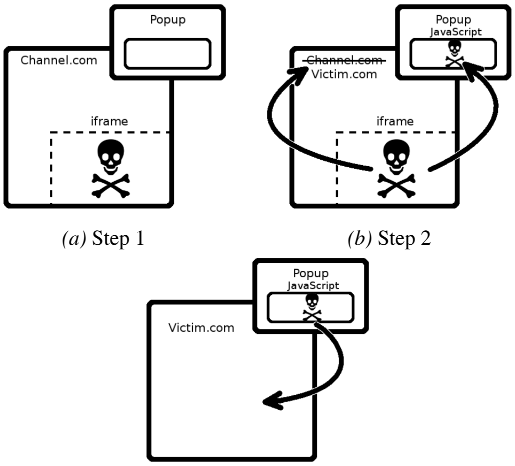
As a proof of concept, we have implemented this attack against Twitter. Our channel includes an HTML page and one geolocation object associated with a popup. At first, this popup simply launches `google.com`. The channel's HTML page contains an iframe with a button. When clicked, this button executes JavaScript which issues an AREL command to Junaio to associate the popup with an attack script, then changes the URL of the transparent page to Twitter with an attacker-chosen tweet text. When the user opens the



(a) XSS vector.

(b) Exploiting XSS.

Figure 8: Cross-site scripting (XSS) in Junaio



(a) Step 1

(b) Step 2

(c) Step 3

Figure 9: Universal XSS vulnerability in Junaio.

popup and clicks the button, Junaio unwittingly injects the script into the Twitter page, where it submits the attacker's tweet.

Other capabilities available to malicious code in an iframe include launching an opaque browser window, turning on and off the camera and the light, removing all AR objects, switching channels, and launching audio and video files.

5.3 How to do it right

Quick patches. The cross-site scripting vulnerabilities described above are caused in part by the fact that the origin of HTML incorporated into an AR channel may be different from the channel's own origin. One plausible defense is for the AR browser to ensure that the two origins are the same; another is to sanitize XML so that it does not contain scripts, which is a notoriously difficult problem [4]. Both defenses require the AR browser to carefully reason about the origins of content specified in custom XML definitions, thus replicating a complex piece of Web-browser functionality. Furthermore, both defenses disable important functional features of AR browsers (such as controlling the appearance of AR objects from another origin) and may break existing applications.

In Wikitude, where evalJavaScript allows channels to inject scripts into an HtmlDrawable regardless of its origin, restricting the origin is not feasible because HtmlDrawable is intended to display content from origins other than the channel itself.

Principled solutions. The root cause of many security holes described in this section is that AR objects cannot be described in HTML, thus AR browsers must use custom mechanisms to enable HTML content to control these objects. Standardizing AR object description languages, including them in HTML5 via either tags, or a special document type, (e.g., *channel*), and adding support for these new HTML5 features into browsers would allow AR content to execute entirely within WebView, eliminating the need for XML and some of the ad-hoc browser interfaces.

Unfortunately, assigning origins to these tags is not trivial. In the existing AR browsers, all AR objects are treated as if their origin is the domain where the main AR channel is hosted. Since these objects may contain JavaScript, this is extremely dangerous.

The alternative is to extend the same-origin policy to AR tags. These tags are intended to support 3D models, animations, UI elements, etc. which may come from different domains but are intended to work smoothly together to produce a unified AR experience. A naive extension of the SOP would isolate the AR HTML tags based on their domains, but this would prevent them from communicating. The developers would then have to implement cross-origin communication mechanisms, which is fraught with peril [26]. Enforcement of the SOP is complicated further by the fact that several of these new tags may need plugins to be rendered (similar to Flash).

Lacking HTML5 support for AR, WebView can at least provide origin-restricted APIs that render arbitrary objects on top of camera images and let JavaScript inside WebView control these objects.

6 Image-Triggered Code Execution

The ability to scan their surroundings and to recognize and track images is fundamental in AR browsers [19, Section 7.5.1.2]. This enables new methods for invoking AR content: for example, a Junaio or Layar channel can be launched simply by scanning a picture or QR code associated with the channel.

6.1 Doing it wrong

When the user is viewing his surroundings through the Junaio or Layar browser, the AR service is continuously analyzing the camera feed. As soon as it recognizes an image associated with some channel, it automatically launches and executes the channel's content, without any confirmation prompts. The user cannot preview the URL or any other information about the content, with one exception: for QR codes (but not pictures), Layar previews the URL by showing it as a button before launching the channel. Unfortunately, its URL parser is broken (Fig. 10). For example, if the URL in the QR code is *http:////attacker.com*, it will not be displayed in the preview, but the browser will launch AR content hosted at *http://attacker.com*. In Junaio, *after* a channel is fully loaded, the user can see its description and the developer's name.

6.2 Risks

In this section, we are primarily concerned with AR attackers who can choose any picture or QR code as the automatic trigger for their malicious channels. For some (but not all) attacks, the attacker needs to physically place these images in public places.

Fully automated, stealthy, large-scale tracking. Because not all AR services vet pictures associated with AR channels, they can be used for automated stalking and tracking. For example, Layar's image recognition algorithm is sufficiently precise to distinguish between license plate numbers. An AR attacker can register a Layar channel associated with the photo of a specific license plate. Whenever any of the millions of Layar users scan their surroundings and the license plate is prominent in the camera's view, the channel is launched automatically and the plate's location, along with its entire visual environment, is sent to the channel's owner,

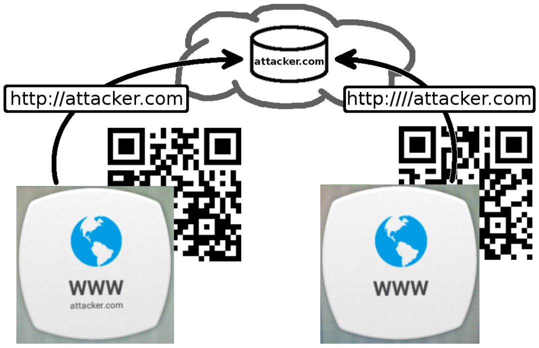


Figure 10: Both codes launch the same channel, but Layar fails to parse the code on the right and does not show the URL.

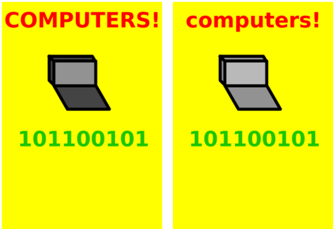


Figure 11: Depending on the angle, each poster nondeterministically launches its own channel or the channel associated with the other poster.

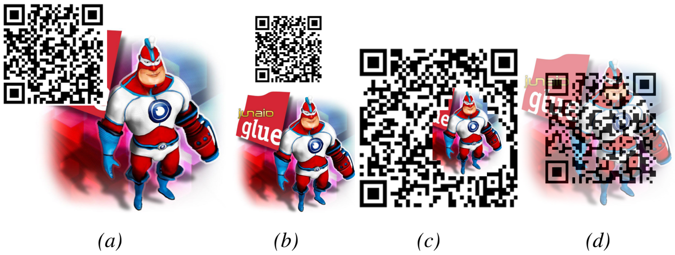
enabling him to track the plate's movements. Other sensitive items can be tracked in a similar fashion.

Automatically launching malicious content. As mentioned above, when an image is recognized by the AR service's (black-box) recognition algorithm, the code of the associated channel executes without user confirmation or channel identification. If a scanned image contains sub-images associated with different channels or a familiar image in an unusual environment or an image that is similar yet subtly different from a familiar image, the user cannot know ahead of time what channel will be launched.

Image recognition algorithms suffer from false positives and are inevitably nondeterministic from the user's point of view [30]. Unfortunately, user interfaces of the AR browsers are derived from the underlying Web browsers and do not inform the user about spurious matches and other potential problems with visual identification.

This can be exploited by an AR attacker in two ways: (1) register an image trigger that is very similar to an image already associated with a trusted channel, or (2) combine a malicious channel's trigger with a trusted channel's trigger into a single composite image. In either scenario, the AR browser may be tricked into automatically launching the malicious channel when scanning the attacker's image on a building wall, bus shelter, etc.

In Layar, the same picture may be associated with multiple channels. For example, we have been able to register our channel with the same movie-poster image as one of Layar's demo channels. If there are multiple channels associated with a picture, the user can open a menu in the corner to see channel names and switch between them. It is possible, however, to create visually similar images that automatically and nondeterministically launch different channels without the browser presenting the channel selection menu to the user. Each poster in Fig. 11 nondeterministically launches the channel associated with the poster or the (completely different) channel associated with the other poster. At many viewing angles and lighting, the channel selection menu is not offered.



(a) *(b)* *(c)* *(d)*

Figure 12: Different combinations of the Junaio mascot and QR code launch different channels.

Furthermore, a malicious channel can suppress the channel selection menu using the native-access capabilities described in Section 4.2. A *layar://[channelname]* pseudo-URI instructs the browser to launch a channel. In this case, the browser does not show other channels associated with the image. Consider a malicious channel that associates itself with the same image as a benign channel. If the user previews the malicious channel before flipping to the benign channel, the first object loaded from the malicious channel can reload the entire channel using *layar://[channelname]* and the other, benign channel will no longer be visible to the user.

The other risk is composite images that include a trusted image in an unexpected visual environment. When faced with a composite image, Junaio's choice of the channel to launch depends on the camera angle and distance. For example, the images in Figs. 12a and 12b launch different channels depending on whether the mascot or the QR code is more prominent. Sometimes, changing the angle of the camera by a few degrees changes which channel is launched. The image in Fig. 12c automatically launches the channel associated with the mascot when scanned from a close distance, and the one associated with the QR code when scanned from further away. Fig. 12d launches the channel associated with the QR code, even though the mascot is visible. This means that even after scanning a familiar image, a user cannot be sure that the automatically launched channel is the one he expects.

6.3 How to do it right

The risk of an AR attacker registering an image trigger that is specific to an individual (e.g., a license plate) is inherent in AR services. A service may attempt to filter out such images during channel registration, but this requires deep semantic analysis of the submitted images and will be inevitably bypassable. This inherent risk is exacerbated by the fact that AR content is executed immediately after the image is scanned.

First, AR browsers should inform the user about the origin of AR content before launching it (at the very least, display the developer's name and basic information about the channel). Second, automatic, image-triggered code execution is fraught with danger and should be used sparingly—for example, only with trusted channels—and not with every image that happens to fall into the camera's field of vision. Third, AR browsers should develop better user interfaces that inform users about the possibility of spurious image matches and nondeterministic launches of unexpected content.

7 Outsourced Image Processing

AR browsers must continuously analyze the device's camera feed in order to recognize automatic content triggers and to anchor or position AR objects on the screen.

7.1 Doing it wrong

AR browsers such as Junaio and Layar do not process the captured images on the device; instead, they send them to central AR servers. There are several reasons to outsource image processing. First, for

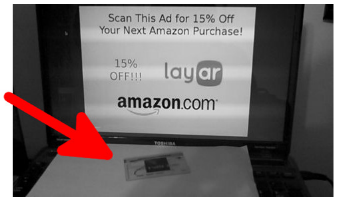


Figure 13: An image sent by the Layar browser over HTTP so that the Layar server can recognize content triggers. Note the accidentally captured credit card.

business reasons—injecting ads, charging content providers, keeping usage statistics, etc.—all AR content retrieval is mediated by the AR service. Second, to facilitate image-based channel launching, recognition of trigger images is done at the server. Because this involves matching against proprietary databases using proprietary algorithms, centralized image processing helps protect intellectual property and removes the need to replicate and update the service's image database on millions of devices. Third, many image recognition algorithms are computationally intensive and would heavily task low-powered mobile devices.

7.2 Risks

In this section, we are concerned with (1) network attackers who observe network traffic between the device and the provider's AR server, and (2) the AR service itself.

Accidental overcollection of sensitive data is a big risk in this setting. For example, the Layar browser sends raw camera images to the server over unencrypted HTTP and includes the phone's location into the GET request for the channel's JSON. Combining images with location data is a serious privacy concern for many users [10]. All sensitive items in the image (see Fig. 13) and request are leaked to any Wi-Fi eavesdropper.

Even if network communications are secure, the AR service inevitably collects a tremendous amount of raw visual data about its users' physical environment. This is an inherent design flaw of all existing AR services. The users must trust them to safeguard captured images, which contain a lot of sensitive information that is completely irrelevant to the AR functionality: screens, credit cards, license plates, etc. Furthermore, a user has no way to learn which data is sent to AR servers. For example, in addition to the unencrypted camera images sent at the start of each scan and the geolocation, the Layar browser occasionally sends a log message to the server with the phone's make, model, and OS version number.

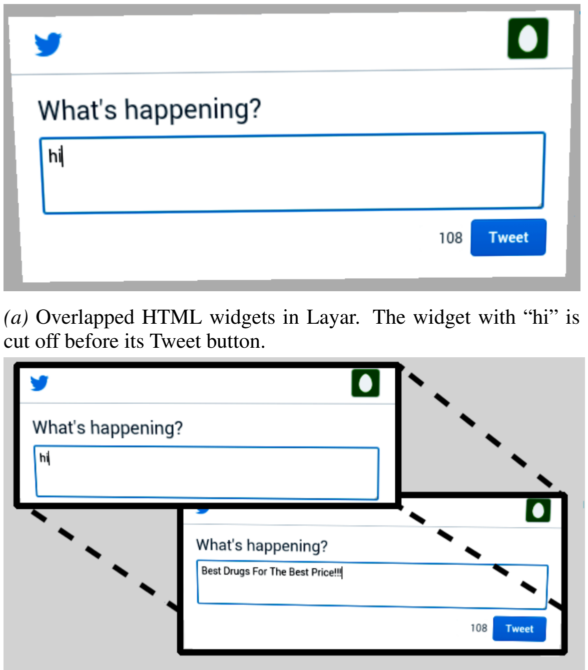
7.3 How to do it right

When image recognition is outsourced to the server, a secure protocol should be used to prevent accidental leakage of irrelevant information in the images. If the server is attempting to recognize a channel trigger on a magazine page, there is no need for it to "see" the physical objects surrounding the page.

This is a difficult problem, but there has been some recent progress. Osadchy et al. described a prototype system for secure outsourced face matching [20]. This system cannot be directly applied in AR browsers, however, because images matched by AR browsers may appear in different lighting, under different angles, etc. Another approach is taken by Darkly [13], which can perform simple computer vision tasks without access to raw image details.

8 Visual Composition

To render images, text, 2D and 3D models, and HTML content from multiple origins on top of the camera feed, AR browsers maintain complex visual stacks. Junaio and Layar's visual stacks are described in Section 5.1.



(a) Overlapped HTML widgets in Layar. The widget with "hi" is cut off before its Tweet button.

(b) The two HTML widgets expanded. In the attack, the bottom widget is not fully shown (its tweet text is covered and not visible to the user).

Figure 14: Clickjacking in Layar.

In Layar, a channel can use a webpage as an AR object, called an *HTML widget*. Each widget opens in its own WebView and does not display URLs or Web-browser buttons. HTML widgets may not be covered by other types of AR objects, but can be overlaid on each other to create a visual AR experience.

8.1 Doing it wrong

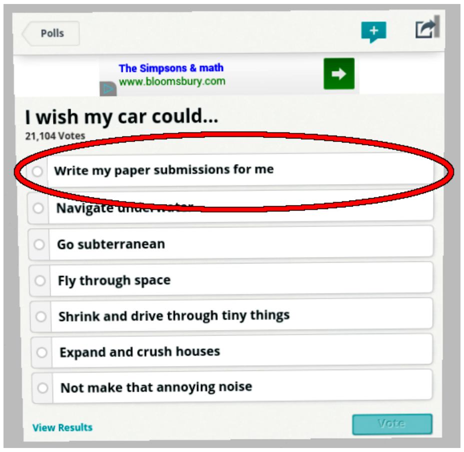
Conventional Web browsers provide the *iframe* abstraction that allows composition of HTML content from different origins. To defend against clickjacking, a webpage can ensure that it is not framed by a page from a different origin, via either framebusting [25] (moving itself to the top frame), or X-Frame-Option [31].

AR browsers must deal with both HTML and non-HTML content, and thus resort to custom mechanisms to implement the functional equivalent of *iframe*. Consequently, standard defenses based on framebusting or X-Frame-Option no longer work. For example, as described above, Layar puts each instance of HTML content into its own WebView instance. Each instance acts like an *iframe* and can be overlaid on other instances. Therefore, a malicious AR channel can overlay content from another origin (B) on top of its own content (A) without B being technically "framed" by A.

8.2 Risks

In this section, we are concerned with an AR attacker whose channel combines his own malicious HTML content with trusted HTML content from other origins.

By cleverly overlaying HTML widgets from different origins, a malicious channel can "hijack" the user's clicks. The user sees a button that appears to belong to some window, but the click is actually captured by a different window. For example, Fig. 14a shows a malicious Layar channel that overlays two Twitter windows. The user may think that the visible "Tweet" button submits the "hi" tweet, but it actually belongs to the bottom window and thus submits the invisible, malicious tweet. Because the victim page is in



Polls

The Simpsons & math
www.bloomsbury.com

I wish my car could...
21,104 Votes

○ Write my paper submissions for me

○ Navigate underwater

○ Go subterranean

○ Fly through space

○ Shrink and drive through tiny things

○ Expand and crush houses

○ Not make that annoying noise

Figure 15: Overlapped HTML widgets in Layar. The first option is in a different widget and, surprisingly, not part of the actual Slashdot poll.

the top/main frame of its own WebView instance, it cannot prevent this attack or even detect when it is being framed in this way.

8.3 How to do it right

Defenses against clickjacking in AR browsers would benefit from a whole-browser equivalent of X-Frame-Option. Layar already prevents non-widget objects from covering widgets, but there is no way for HTML content to specify that its widget—or any WebView in which it is displayed—should not overlap with other widgets.

In general, using conventional browsers such as WebView to render AR content is dangerous because it forces AR browsers to use ad-hoc mechanisms for visually combining content from different origins. A principled solution to clickjacking in AR browsers should involve a clean-slate redesign of their user interfaces.

9 Indirect Retrieval of AR Content

AR content comes from independent third-party developers. While in theory the AR browser could fetch this content directly from the developers' servers, in practice the business models of AR services require them to track usage, charge fees for channel registration, inject advertising, and, in general, tightly monitor the interaction between their browsers and third-party content. Consequently, content requests must pass through the AR provider's own servers.

9.1 Doing it wrong

Some AR browsers enable third-party channels to authenticate users or keep track of users' preferences between their visits. For example, Layar supports a cookie-based user authentication scheme that can be deployed by geolocation channels (Fig. 16). When requesting a channel, the Layar browser sends a POST request to the Layar server and attaches the cookies associated with the channel's origin. The Layar server then attaches these cookies to the GET request it forwards to the channel server.

Cookie security depends on the binding between the cookie and its origin. A conventional Web browser keeps this binding and automatically attaches the cookie to every request sent to its origin. In Layar, channel launches are mediated by the Layar server, which must maintain the same cookie-origin binding as the Layar browser.

The Layar browser learns the origin of the channel from the Layar server. When the browser first loads the channel, the cookies are set by the channel's authentication page and thus correctly bound to the channel's origin at that time. If this origin changes later (e.g., the channel moves to a different domain), the Layar server notes the change and forwards all browser requests to the new location. Critically, the Layar server does *not* notify the browsers connected to the channel that the channel's origin has changed. The browsers continue to attach the cookies from the old origin to their requests, and the Layar server obliviously forwards them to the new origin.

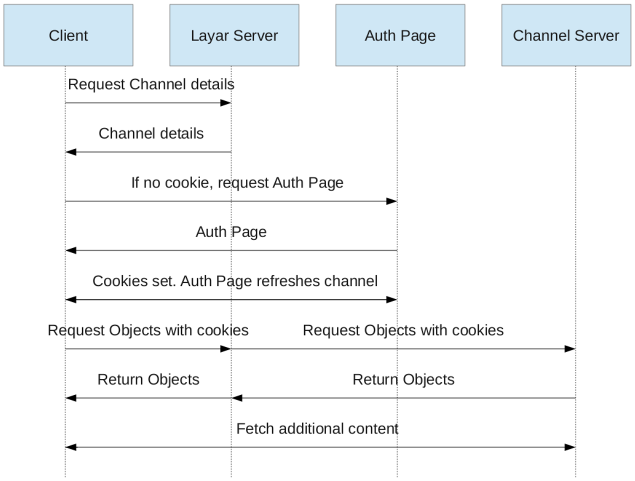


Figure 16: User authentication in Layar.

9.2 Risks

In this section, we are concerned with AR attackers who lie about their channels' URLs. By "desynchronizing" the Layar browser's and the Layar server's understanding of the channel's origin, a malicious channel can steal cookies from *any* origin (Fig. 17).

For example, the attacker initially tells Layar that the URL of his channel is *https://www.twitter.com*. When a user launches the channel, the Layar browser attaches Twitter cookies to every channel update request. Next, the attacker changes his channel's URL to *https://attacker.com*. The Layar server registers the change, but the browsers connected to the channel continue to attach Twitter cookies to every channel update request. The Layar server forwards these requests, cookies attached, to *https://attacker.com*, and the attacker steals all of its users' Twitter cookies.

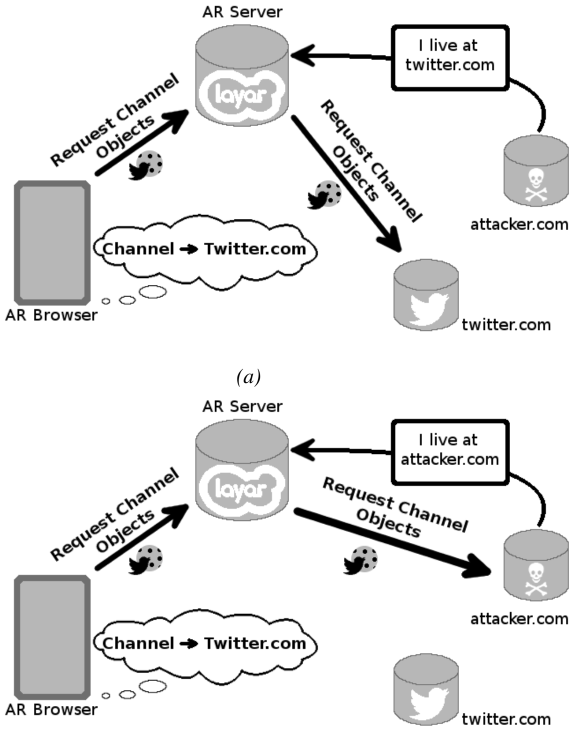
This attack works for any domain of the attacker's choosing (we tested it for Twitter and Facebook). Note that many AR channels are integrated with online social networks, thus the user is likely to be logged into Facebook and Twitter through his AR browser.

9.3 How to do it right

The first defense is to avoid replicating the state of the browser on the Layar server. The browser may request the URL of the channel server from the Layar server, but subsequent communication should be conducted directly between the browser and the channel server. The same-origin policy within the browser will then ensure that cookies are disclosed only to their origins. This defense, however, may break Layar's business model.

The second defense is for the Layar server to ensure that it agrees with the browser about the channel server's URL. This defense requires re-engineering the protocol between the browser, Layar server, and channel server.

The final defense is to use an authentication protocol that supports delegation, e.g., OAuth. In current Layar, channels may use OAuth 1.0 to authenticate the Layar server. This protects benign AR developers from spoofed Layar servers, but not legitimate Layar servers from malicious developers, and thus does not help against the cookie-stealing attack.



(a)

(b)

Figure 17: Layar cookie stealing attack.

10 Related Work

Azuma et al. [2, 3] identified three major properties of AR systems, exhibited by all AR browsers in our study: combining real and virtual objects, real-time interactivity, and support for 3D blending of virtual and real objects. Several papers suggested adding augmented reality to mobile applications such as tour guides [1, 8]. Spohrer et al. [28] explored the idea of associating information with real-world objects using "WorldBoard channels." Just like AR channels analyzed in this paper, "WorldBoard channels" can display HTML-encoded information overlaid on real-world objects. Kooper et al. [16] used the term "real world-wide web" for the combined information space created by merging real-world objects with the HTML content of the WWW. ARML [19] is a proposed standard for defining geospatial AR objects through XML.

Roesner et al. [22, 23] have surveyed various security, privacy, and legal concerns arising from the widespread use of AR technologies. By contrast, we analyze the technical architecture of popular, deployed AR platforms. With the exception of clickjacking and general privacy concerns, none of the issues we discovered are mentioned in these papers.

Several recent papers focused on privacy concerns arising from the unrestricted access to sensor data by untrusted third-party applications. Darkly [13] prevents certain privacy violations by applications based on the OpenCV computer vision library; D'Antoni et al. [6] and Jana et al. [12] show how to confine AR applications by adding fine-grained permissions to the OS. These systems are concerned with protecting users from untrusted applications, whereas we investigate whether and how trusted AR applications protect users from untrusted content (i.e., our threat model is similar to the standard threat model of Web browsers).

Some of our attacks involve pictures or QR codes placed in a public area to trick AR browsers into launching a malicious AR channel. Lookout Mobile Security used a QR code to force Google Glass to connect to an attacker-controlled Wi-Fi access point.[4] Both attacks employ malicious QR codes, but the similarities end there. The attacks described in Section 6.2 exploit the deficiencies of user interfaces in AR browsers, not software vulnerabilities. Other related work includes security threats involving QR codes [15] and the use of QR codes for malware distribution and phishing [17]. Dabrowski et al. [5] recently demonstrated numerous attacks involving hiding a QR code inside of another QR code, similar to our our attacks in Section 6.2.

Clickjacking attacks against conventional Web content were analyzed in [11, 24, 25]. In Section 8.2, we explained that our clickjacking attacks and defenses are somewhat different because of the architectural differences between Web browsers and AR browsers.

11 Conclusions

Augmented reality (AR) browsers are a new technology with exciting potential. We presented the first in-depth analysis of their security and privacy properties, identified multiple architectural flaws, and proposed short-term fixes for specific vulnerabilities as well as directions for future research on building secure AR browsers.

We have reported our findings to Junaio, Layar, and Wikitude. Junaio informed us that they will incorporate our results into their latest internal build. Wikitude was aware of the security flaw in HtmlDrawable (Section 5.2) and is looking into adding security mechanisms. Layar never responded to us.

Acknowledgments. This work was partially supported by the NSF grants CNS-0746888 and CNS-1223396, a Google research award, NIH grant R01 LM011028-01 from the National Library of Medicine, and Google PhD Fellowship to Suman Jana.

12 References

[1] G. Abowd, C. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: A mobile context-aware tour guide. *Wireless Networks*, 3(5), 1997.

[2] R. T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, 1997.

[3] R. T. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent advances in augmented reality. *Computer Graphics and Applications*, 21(6):34–47, 2001.

[4] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *WWW*, 2010.

[5] A. Dabrowski, K. Krombholz, J. Ullrich, and E. Weippl. QR inception: Barcode-in-barcode attacks. In *SPSM*, 2014.

[6] L. D'Antoni, A. Dunn, S. Jana, T. Kohno, B. Livshits, D. Molnar, A. Moshchuk, E. Ofek, F. Roesner, S. Saponas, M. Veanes, and H. J. Wang. Operating system support for augmented reality applications. In *HotOS*, 2013.

[7] Layar launches "world's first augmented reality store". `http://eurodroid.com/2010/04/28/layar-launches-worlds-first-augmented-reality-store`, 2010.

[8] S. Feiner, B. MacIntyre, T. Höllerer, and A. Webster. A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. *Personal Technologies*, 1(4), 1997.

[9] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid Web/mobile application frameworks. In *NDSS*, 2014.

[10] B. Henne, M. Harbach, and M. Smith. Location privacy revisited: Factors of privacy decisions. In *CHI*, 2013.

[11] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and defenses. In *USENIX Security*, 2012.

[4] `http://www.techweekeurope.co.uk/news/google-glass-security-vulnerability-internet-of-things-122073`

[12] S. Jana, D. Molnar, A. Moshchuk, A. Dunn, B. Livshits, H. J. Wang, and E. Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *USENIX Security*, 2013.

[13] S. Jana, A. Narayanan, and V. Shmatikov. A scanner Darkly: Protecting user privacy from perceptual applications. In *S&P*, 2013.

[14] Become a Junaio developer. `http://www.slideshare.net/metaio_AR/why-to-become-a-junaio-developer`, 2013.

[15] A. Kharraz, E. Kirda, W. Robertson, D. Balzarotti, and A. Francillon. Optical delusions: A study of malicious QR codes in the wild. In *DSN*, 2014.

[16] R. Kooper and B. B. MacIntyre. Browsing the real-world wide web: Maintaining awareness of virtual information in an AR information space. *International Journal of Human-Computer Interaction*, 16(3), 2003.

[17] K. Krombholz, P. Frühwirt, P. Kieseberg, I. Kapsalis, M. Huber, and E. Weippl. QR code security: A survey of attacks and challenges for usable security. In *HCI*, 2014.

[18] Layar introduction for developers. `http://www.slideshare.net/layarmobile/layar-introduction-for-developers`, 2011.

[19] Open Geospatial Consortium. OGC augmented reality markup language 2.0 (ARML 2.0) [candidate standard]. `http://www.opengeospatial.org/projects/groups/arml2.0swg`, 2013.

[20] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. SCiFI - A system for secure face identification. In *S&P*, 2010.

[21] C. Perey. A proposal for AR browser interoperability. `http://www.perey.com/ARStandards/AR_Browser_Interoperability_Architecture_Jan_21_2014_v1_2.pdf`, 2014.

[22] F. Roesner, T. Kohno, T. Denning, R. Calo, and B. C. Newell. Augmented reality: Hard problems of law and policy. In *UPSIDE*, 2014.

[23] F. Roesner, T. Kohno, and D. Molnar. Security and privacy for augmented reality systems. In *Communications of the ACM*, volume 57, pages 88–96, 2014.

[24] G. Rydstedt, E. Bursztein, and D. Boneh. Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization. In *WOOT*, 2010.

[25] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: A study of clickjacking vulnerabilities at popular sites. In *W2SP*, 2010.

[26] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postMessage in HTML5 websites. In *NDSS*, 2013.

[27] Same origin policy. `http://www.w3.org/Security/wiki/Same_Origin_Policy`.

[28] J. Spohrer. Information in places. *IBM Systems Journal*, 38(4):602–628, 1999.

[29] Wikitude for agencies. `http://www.slideshare.net/wikitude/wikitude-media-portfolio-presentation`, 2012.

[30] Z. Wu, Q. Ke, M. Isard, and J. Sun. Bundling features for large scale partial-duplicate web image search. In *CVPR*, 2009.

[31] The X-Frame-Options response header. `https://developer.mozilla.org/en-US/docs/HTTP/X-Frame-Options`.