The Bridge between Web Applications and Mobile Platforms is Still Broken

Philipp Beer, Lorenzo Veronese, Marco Squarcina, Martina Lindorfer
TU Wien

Contributions

Malicious application

Android Custom Tab

Attack similar to **XS state inference** and **CSRF**

UI flaw

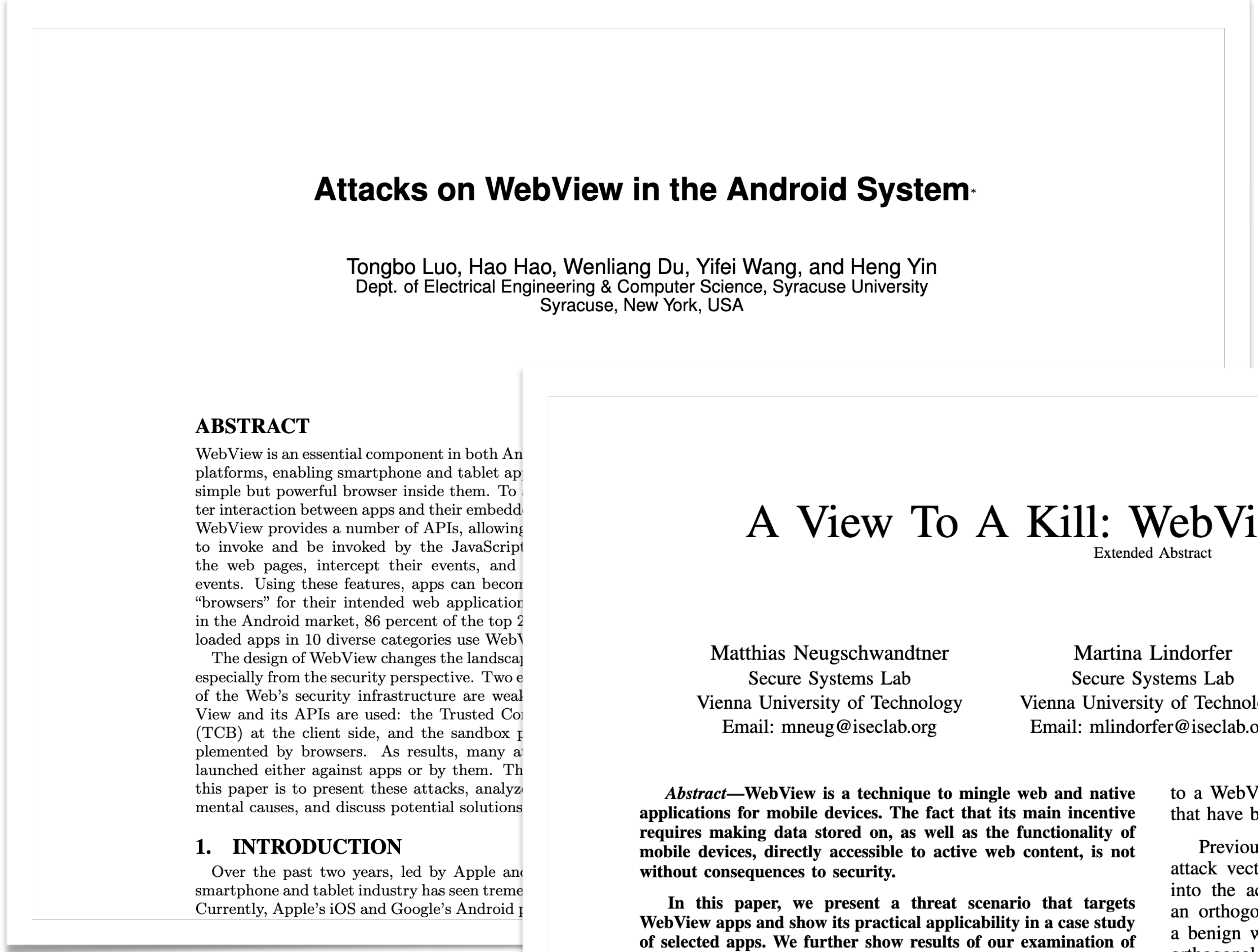
Achieves Stealthiness

Malicious website

Android WebView Attack

Accesses to user's **microphone/camera**

Related Work



Attacks on WebView in the Android System

Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin
Dept. of Electrical Engineering & Computer Science, Syracuse University
Syracuse, New York, USA

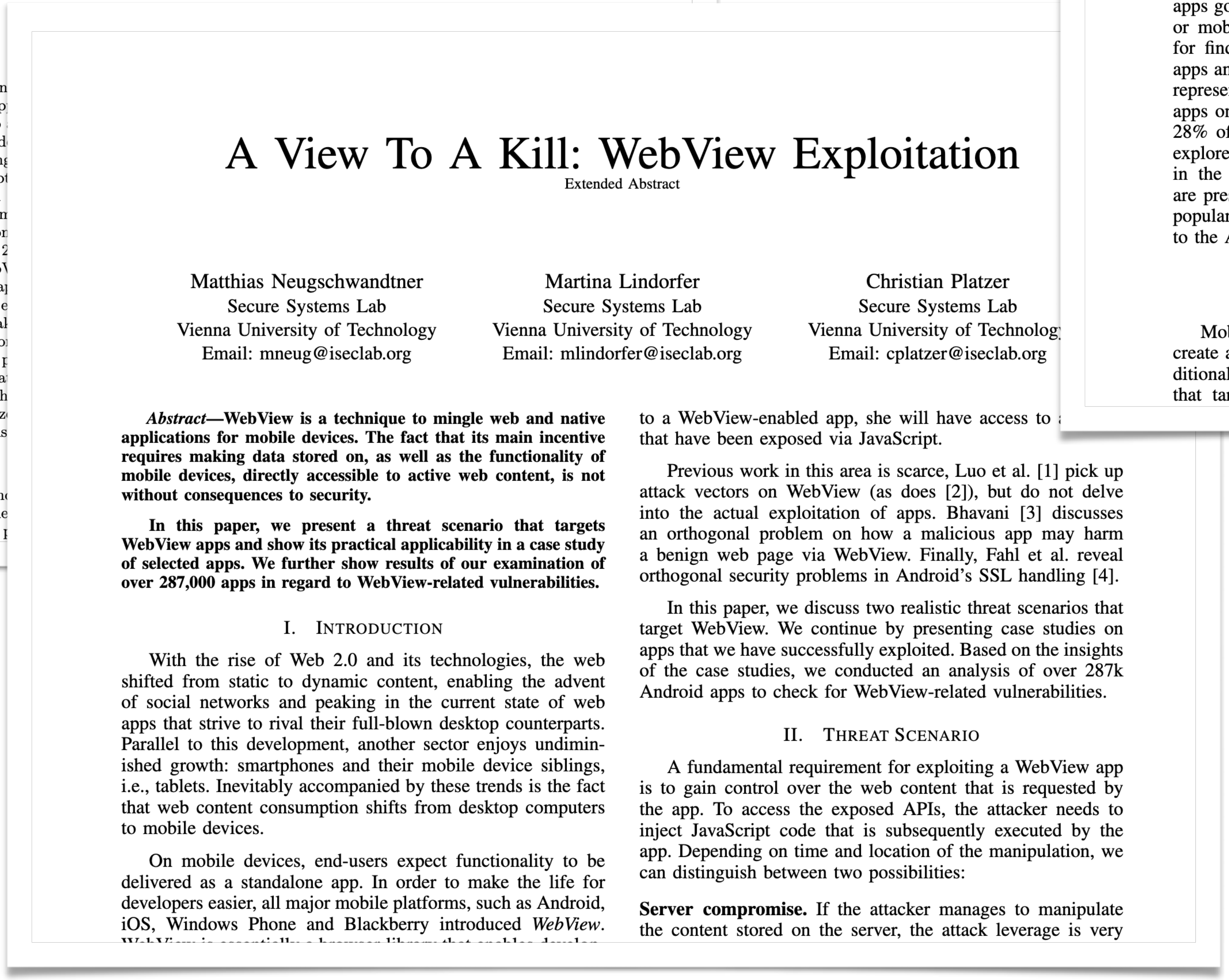
ABSTRACT

WebView is an essential component in both An… platforms, enabling smartphone and tablet ap… simple but powerful browser inside them. To … ter interaction between apps and their embedd… WebView provides a number of APIs, allowing … to invoke and be invoked by the JavaScript … the web pages, intercept their events, and … events. Using these features, apps can becom… "browsers" for their intended web application… in the Android market, 86 percent of the top 2… loaded apps in 10 diverse categories use WebV…

The design of WebView changes the landscap… especially from the security perspective. Two e… of the Web's security infrastructure are weal… View and its APIs are used: the Trusted Co… (TCB) at the client side, and the sandbox p… plemented by browsers. As results, many a… launched either against apps or by them. Th… this paper is to present these attacks, analyze… mental causes, and discuss potential solutions…

1. INTRODUCTION

Over the past two years, led by Apple an… smartphone and tablet industry has seen treme… Currently, Apple's iOS and Google's Android …



A View To A Kill: WebView Exploitation

Extended Abstract

Matthias Neugschwandtner, Secure Systems Lab, Vienna University of Technology, Email: mneug@iseclab.org
Martina Lindorfer, Secure Systems Lab, Vienna University of Technology, Email: mlindorfer@iseclab.org
Christian Platzer, Secure Systems Lab, Vienna University of Technology, Email: cplatzer@iseclab.org

***Abstract*—WebView is a technique to mingle web and native applications for mobile devices. The fact that its main incentive requires making data stored on, as well as the functionality of mobile devices, directly accessible to active web content, is not without consequences to security.**

In this paper, we present a threat scenario that targets WebView apps and show its practical applicability in a case study of selected apps. We further show results of our examination of over 287,000 apps in regard to WebView-related vulnerabilities.

I. Introduction

With the rise of Web 2.0 and its technologies, the web shifted from static to dynamic content, enabling the advent of social networks and peaking in the current state of web apps that strive to rival their full-blown desktop counterparts. Parallel to this development, another sector enjoys undiminished growth: smartphones and their mobile device siblings, i.e., tablets. Inevitably accompanied by these trends is the fact that web content consumption shifts from desktop computers to mobile devices.

On mobile devices, end-users expect functionality to be delivered as a standalone app. In order to make the life for developers easier, all major mobile platforms, such as Android, iOS, Windows Phone and Blackberry introduced *WebView*.

to a WebView-enabled app, she will have access to … that have been exposed via JavaScript.

Previous work in this area is scarce, Luo et al. [1] pick up attack vectors on WebView (as does [2]), but do not delve into the actual exploitation of apps. Bhavani [3] discusses an orthogonal problem on how a malicious app may harm a benign web page via WebView. Finally, Fahl et al. reveal orthogonal security problems in Android's SSL handling [4].

In this paper, we discuss two realistic threat scenarios that target WebView. We continue by presenting case studies on apps that we have successfully exploited. Based on the insights of the case studies, we conducted an analysis of over 287k Android apps to check for WebView-related vulnerabilities.

II. Threat Scenario

A fundamental requirement for exploiting a WebView app is to gain control over the web content that is requested by the app. To access the exposed APIs, the attacker needs to inject JavaScript code that is subsequently executed by the app. Depending on time and location of the manipulation, we can distinguish between two possibilities:

Server compromise. If the attacker manages to manipulate the content stored on the server, the attack leverage is very

A Large-Scale Study of Mobile Web App Security

Patrick Mutchler*, Adam Doupé†, John Mitchell*, Chris Kruegel‡ and Giovanni Vigna‡
*Stanford University
{pcm2d, mitchell}@stanford.edu
†Arizona State University
doupe@asu.edu
‡University of California, Santa Barbara
{chris, vigna}@cs.ucsb.edu

Abstract

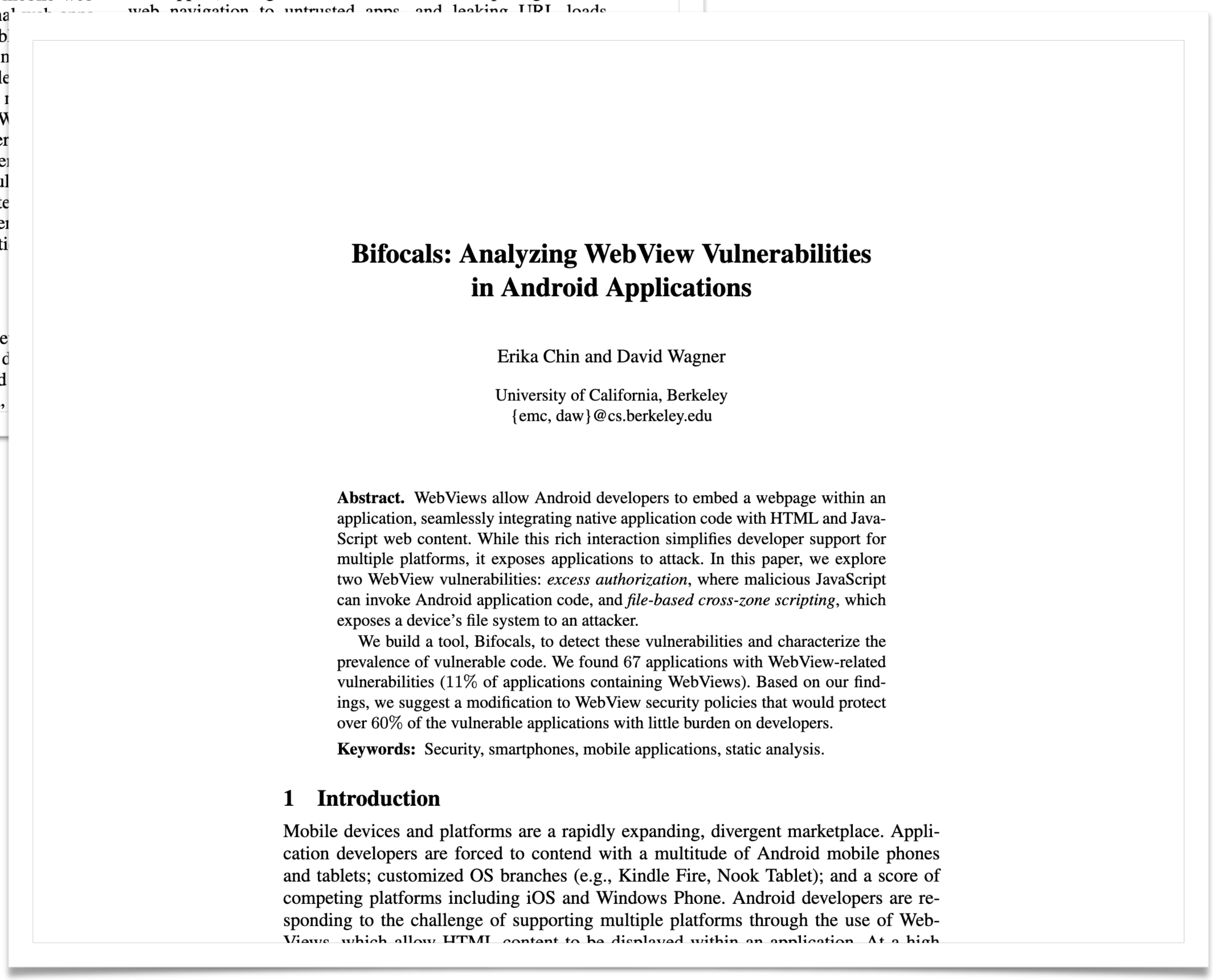
Mobile apps that use an embedded web browser, or *mobile web apps*, make up 85% of the free apps on the Google Play store. The security concerns for developing mobile web apps go beyond just those for developing traditional … or mobile apps. In this paper we develop scalab… for finding several classes of vulnerabilities in … apps and analyze a large dataset of 998,286 mobile … representing a complete snapshot of all of the free … apps on the Google Play store as of June 2014. W… 28% of the studied apps have at least one vulner… explore the severity of these vulnerabilities and ide… in the vulnerable apps. We find that severe vul… are present across the entire Android app ecosyste… popular apps and libraries. Finally, we offer sever… to the Android APIs to mitigate these vulnerabiliti…

I. Introduction

Mobile operating systems allow third-party de… create applications ("apps") that run on a mobile … ditionally, apps are developed using a language and … that targets a specific mobile operating system, …

of the causes of vulnerabilities in mobile web apps but an inadequate understanding of their true prevalence in the wild.

In this work we study three vulnerabilities in mobile web apps (loading untrusted web content, exposing stateful …



Bifocals: Analyzing WebView Vulnerabilities in Android Applications

Erika Chin and David Wagner
University of California, Berkeley
{emc, daw}@cs.berkeley.edu

Abstract. WebViews allow Android developers to embed a webpage within an application, seamlessly integrating native application code with HTML and JavaScript web content. While this rich interaction simplifies developer support for multiple platforms, it exposes applications to attack. In this paper, we explore two WebView vulnerabilities: *excess authorization*, where malicious JavaScript can invoke Android application code, and *file-based cross-zone scripting*, which exposes a device's file system to an attacker.

We build a tool, Bifocals, to detect these vulnerabilities and characterize the prevalence of vulnerable code. We found 67 applications with WebView-related vulnerabilities (11% of applications containing WebViews). Based on our findings, we suggest a modification to WebView security policies that would protect over 60% of the vulnerable applications with little burden on developers.

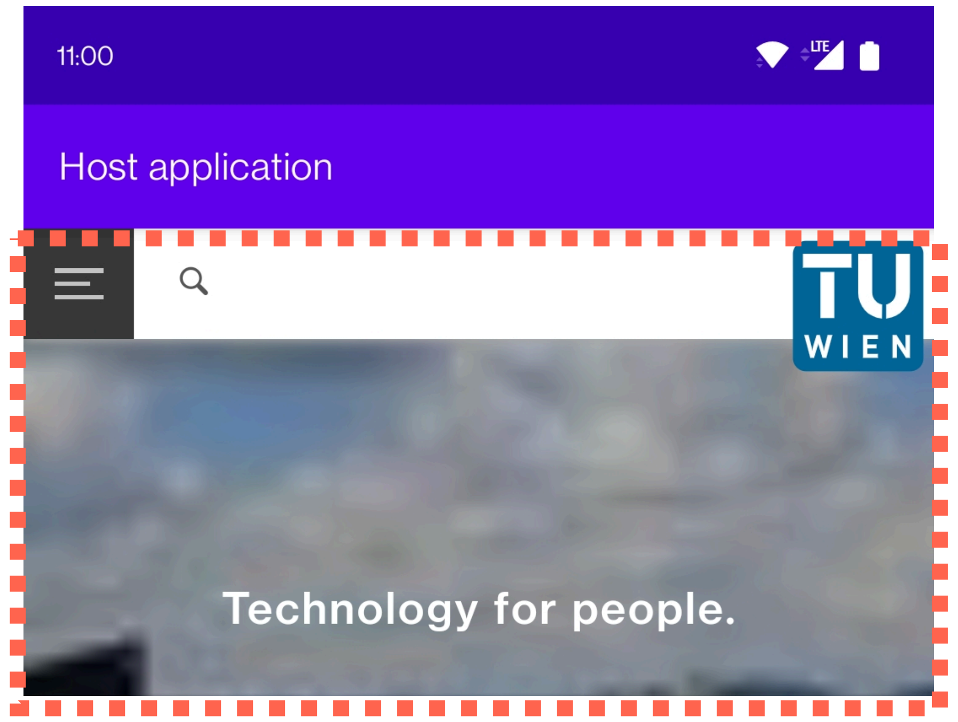
Keywords: Security, smartphones, mobile applications, static analysis.

1 Introduction

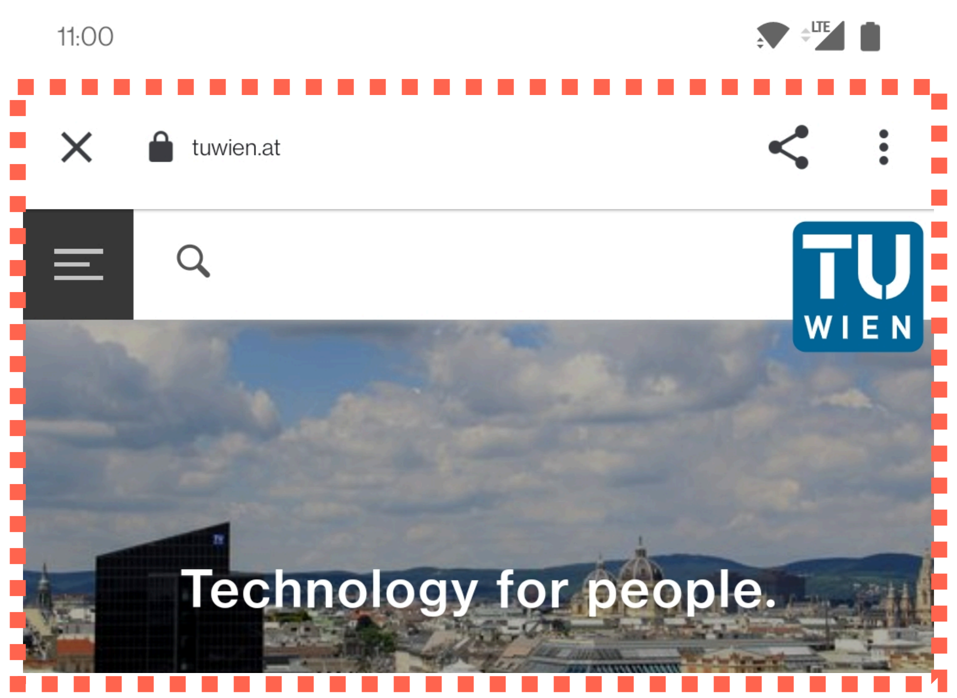
Mobile devices and platforms are a rapidly expanding, divergent marketplace. Application developers are forced to contend with a multitude of Android mobile phones and tablets; customized OS branches (e.g., Kindle Fire, Nook Tablet); and a score of competing platforms including iOS and Windows Phone. Android developers are responding to the challenge of supporting multiple platforms through the use of Web-

Integrating Web Content in Mobile Apps

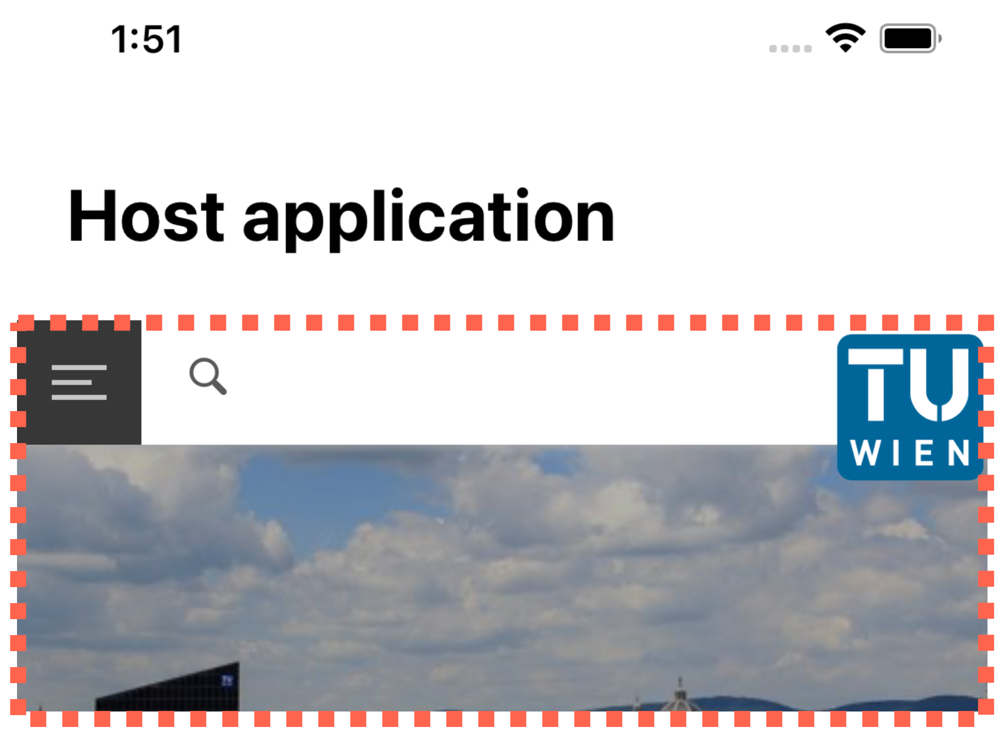
- Serve as in-app browsers
- Android
  - WebView
  - Custom Tab
  - Trusted Web Activities
- iOS
  - WKWebView
  - SFSafariViewController



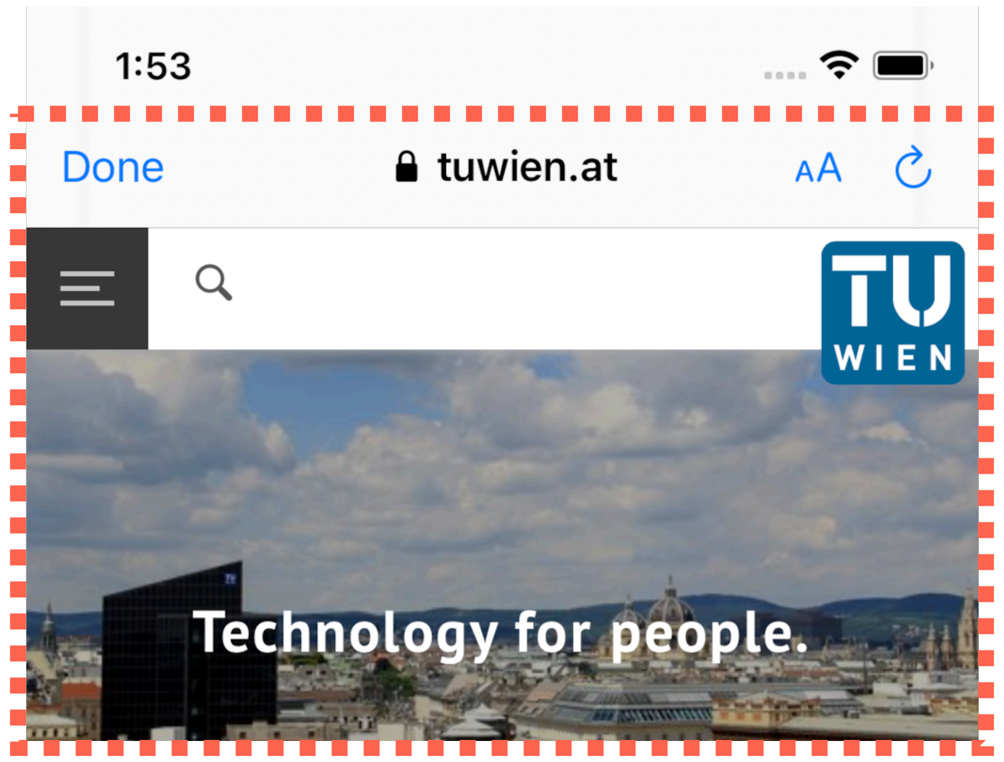
Android WebView



Android Custom Tab



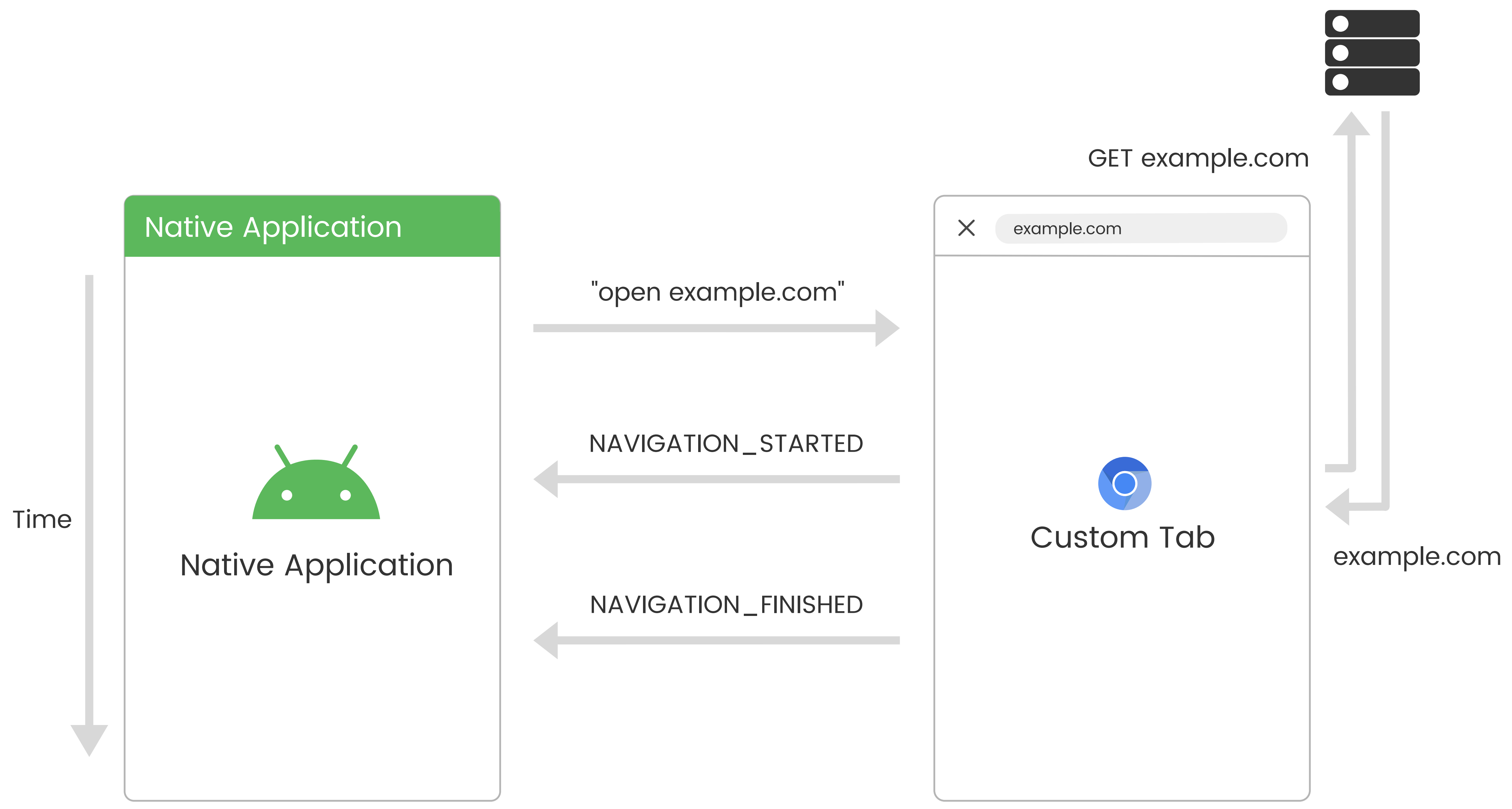
iOS WKWebView



iOS SFSafariViewController

Custom Tab

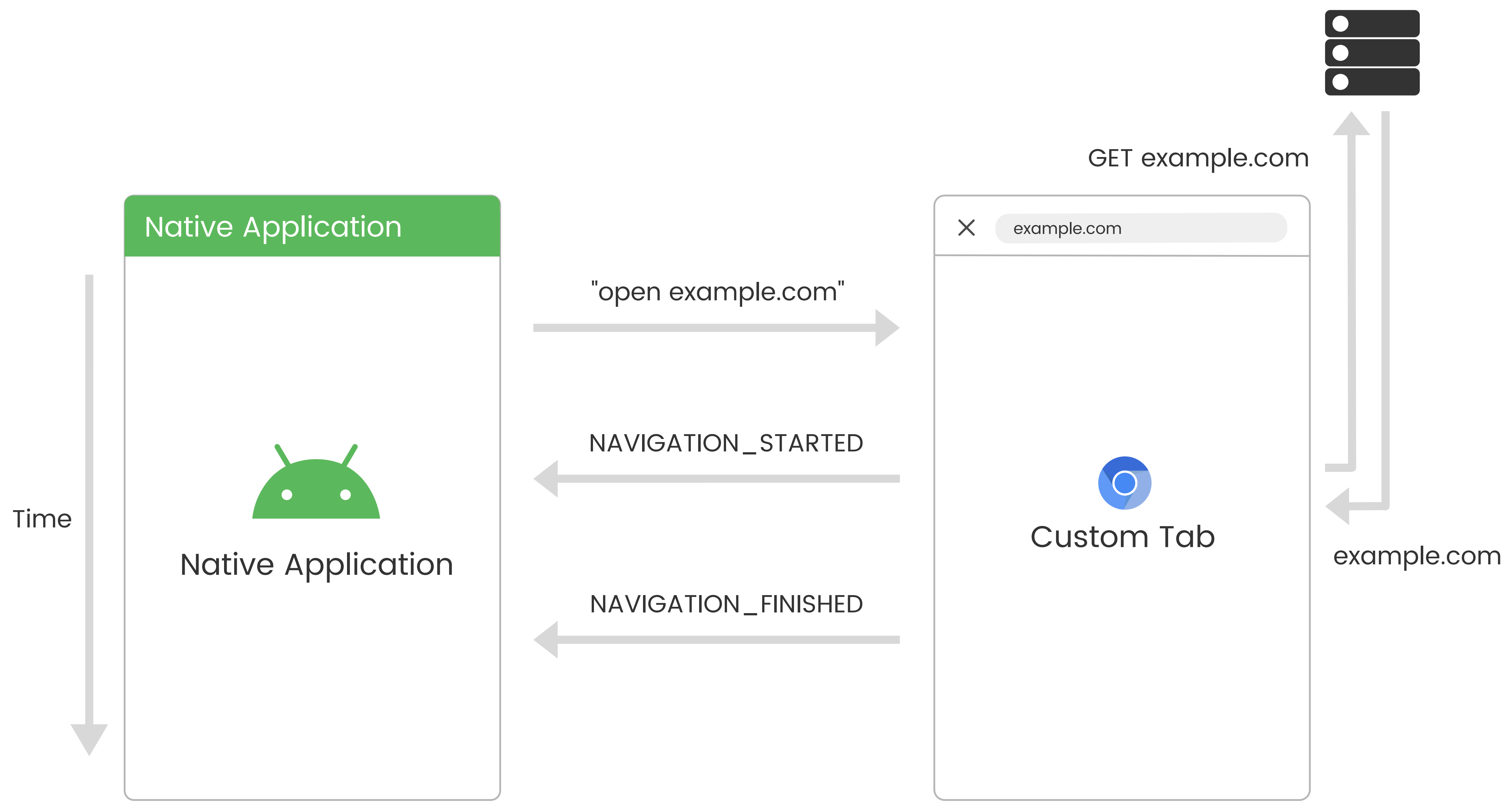
- Report navigation callbacks to host application
- Custom Tabs share state with browser
- Useful for e.g. SSO



Custom Tab Callback Principle

Custom Tab Attack

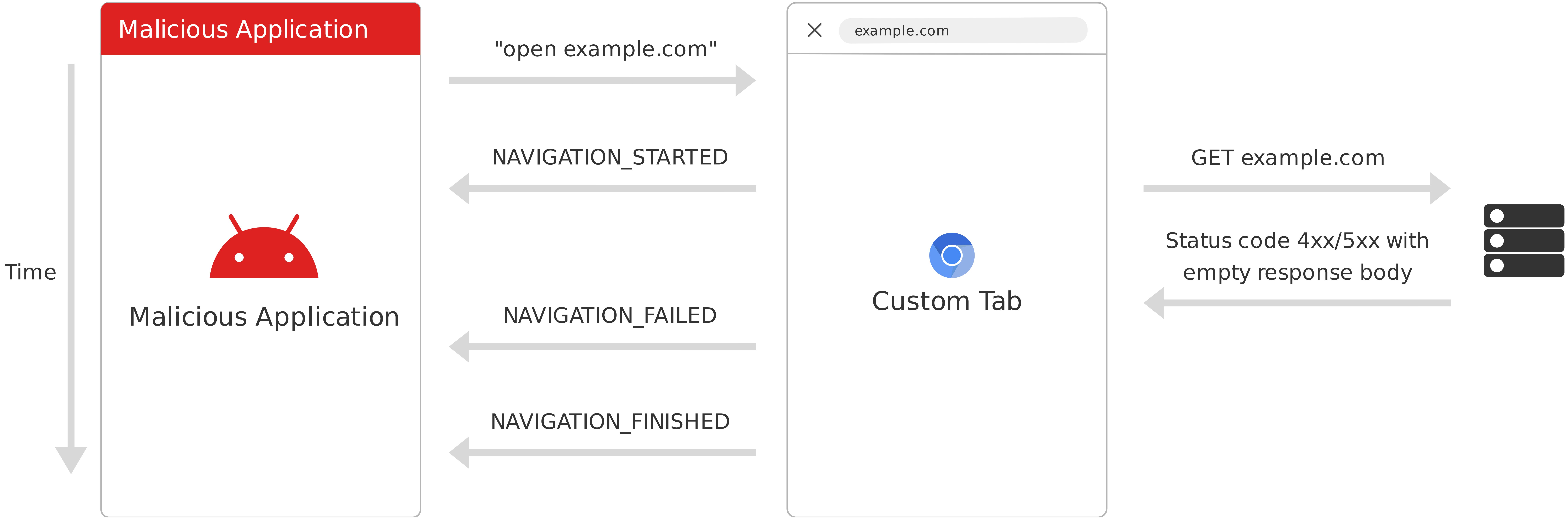
- Features enable attack similar to XS-leak to infer user information
- Malicious app uses event sequence to infer user data
- Three approaches
  - Status code-based approach
  - Redirection-based approach
  - Timing-based approach



Custom Tab Callback Principle

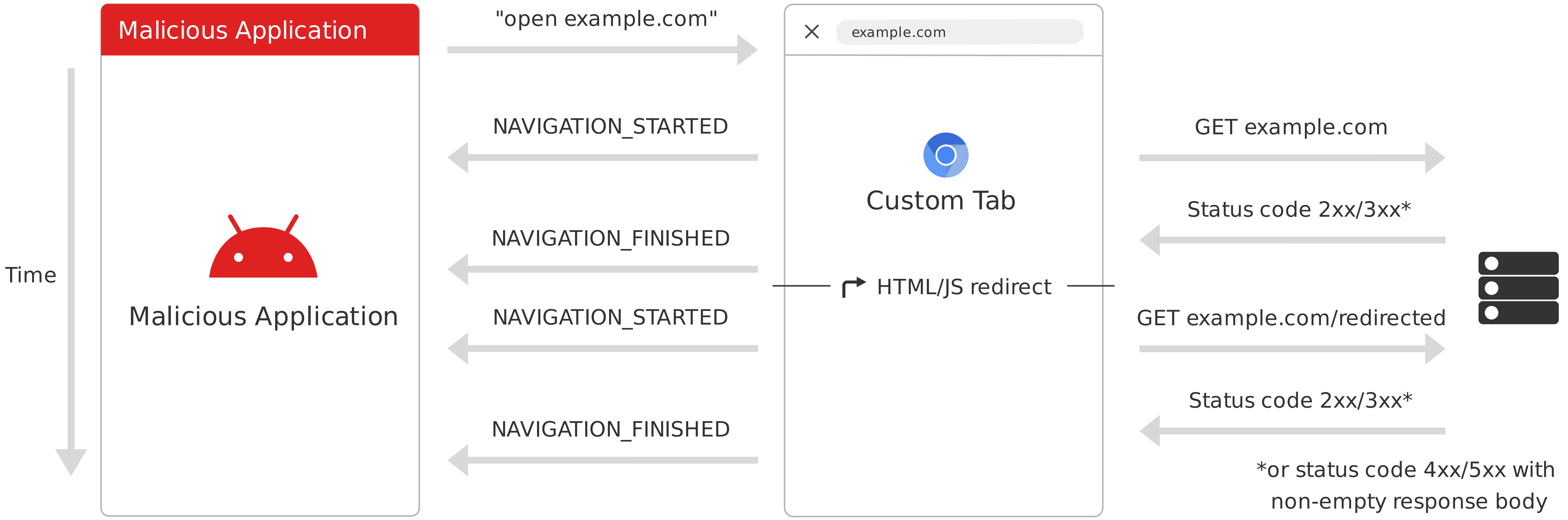
Status code-based approach

- Additional failed event triggered on 4xx/5xx response code and empty response body



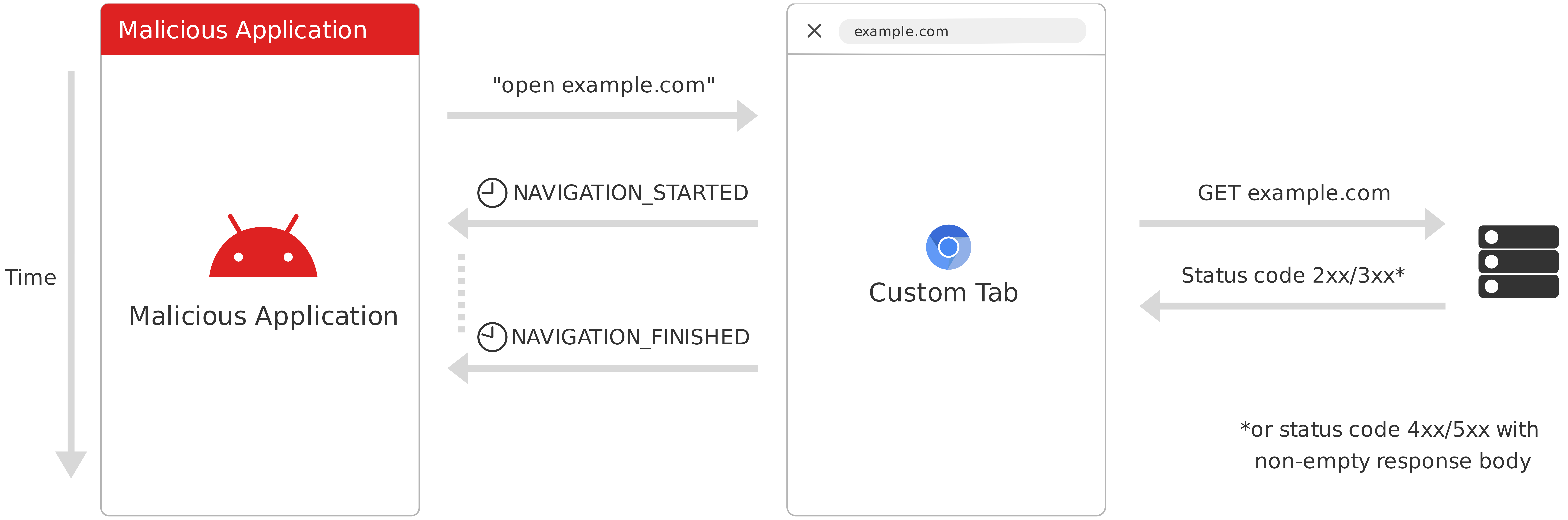
Redirection-based approach

- Finished/failed event triggered for every JS/meta redirection

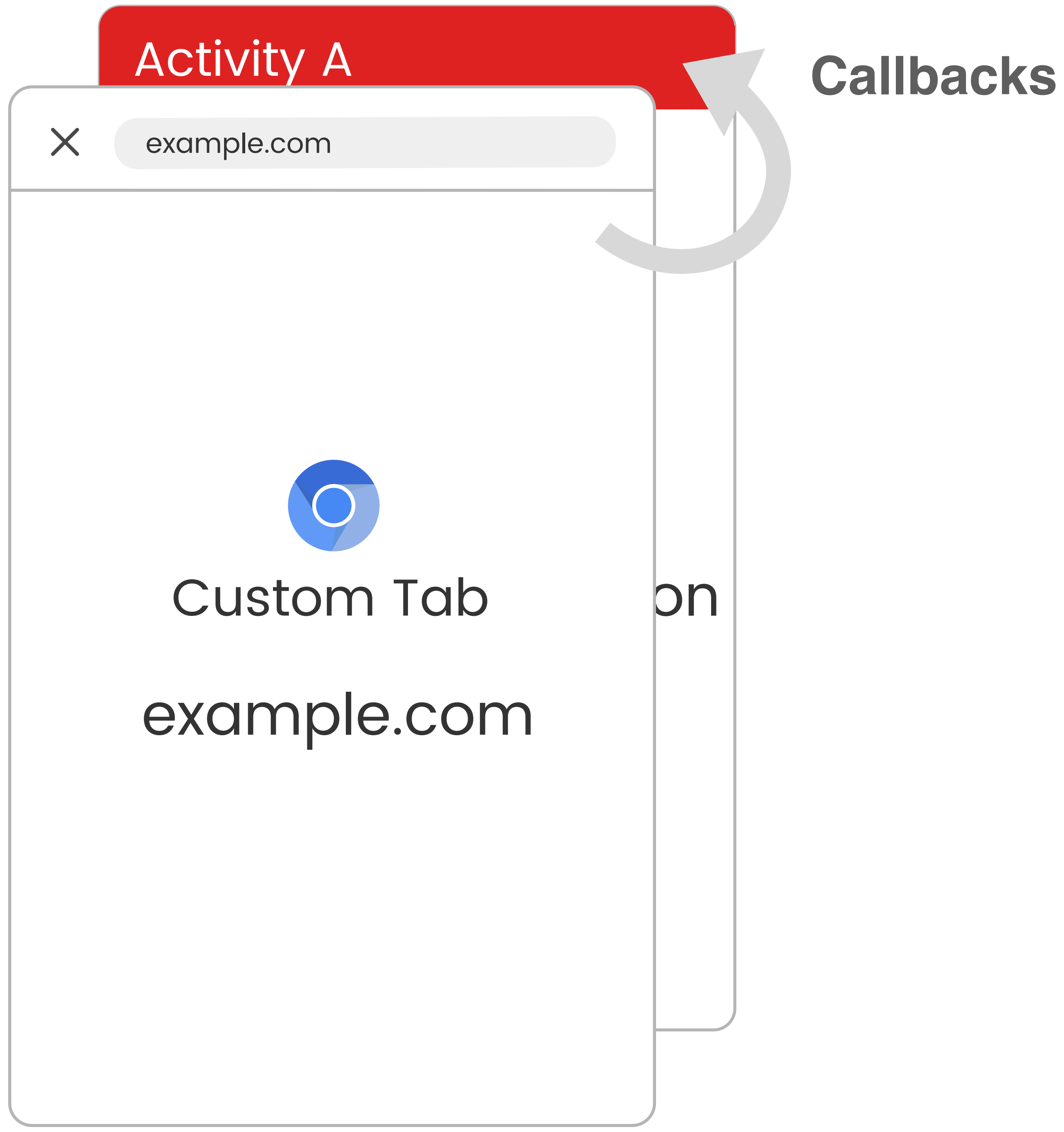


Timing-based approach

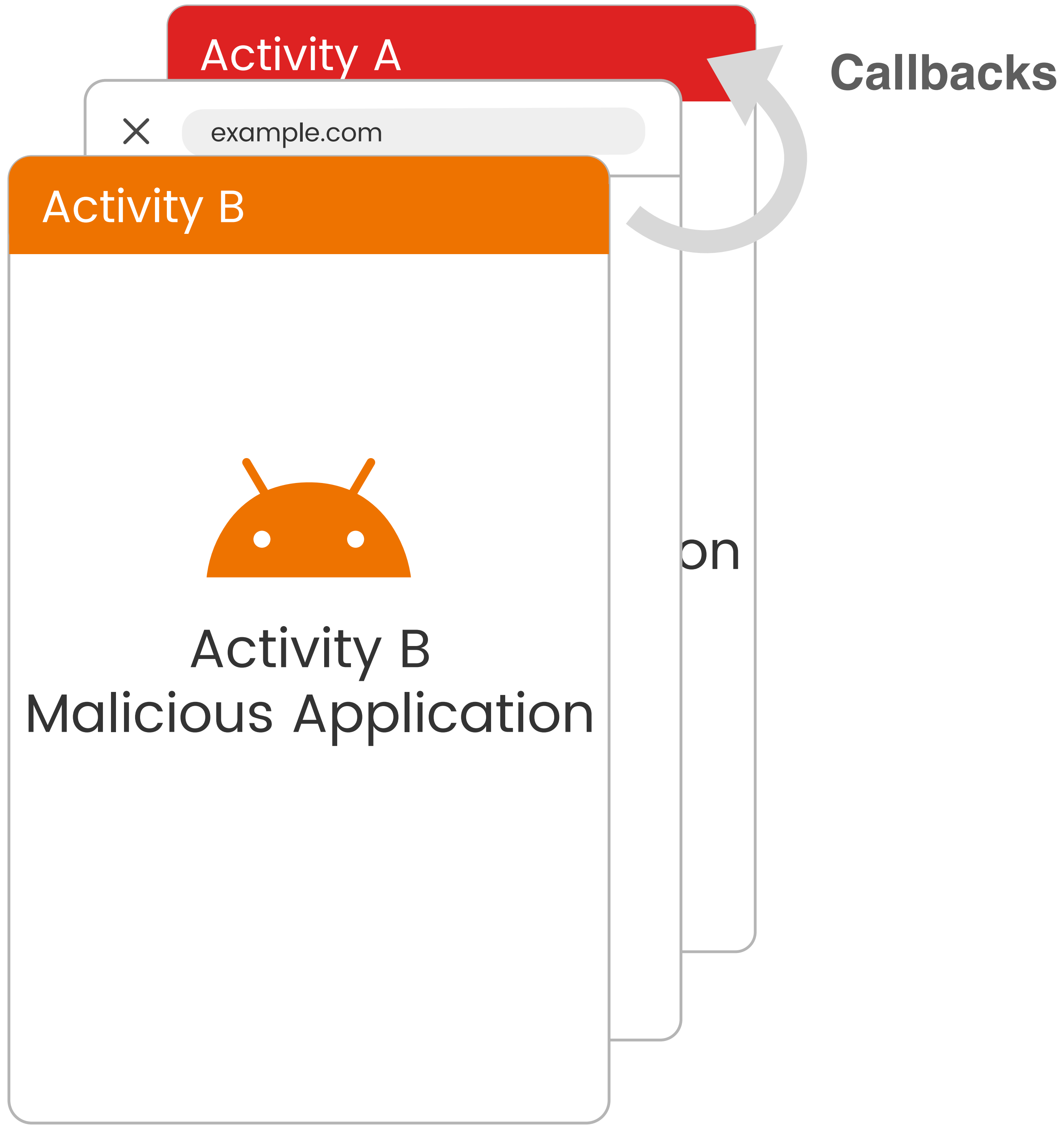
- Measure time between NAVIGATION_STARTED and NAVIGATION_FINISHED



Stealthiness

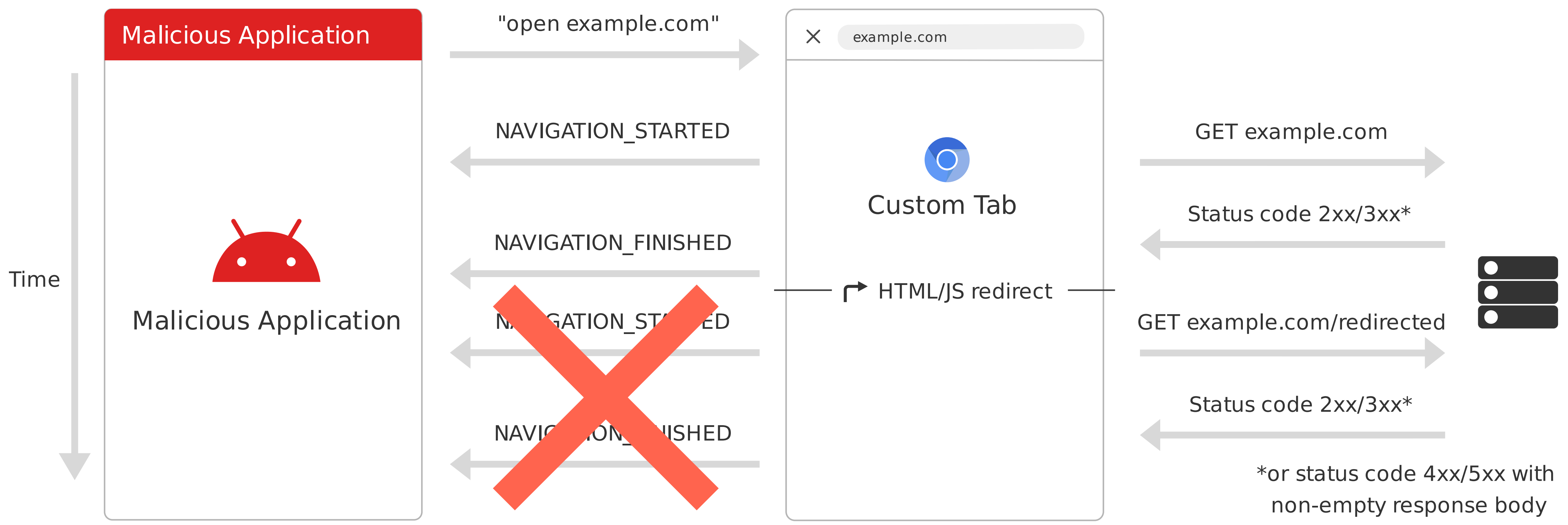


Normal Custom Tab launch

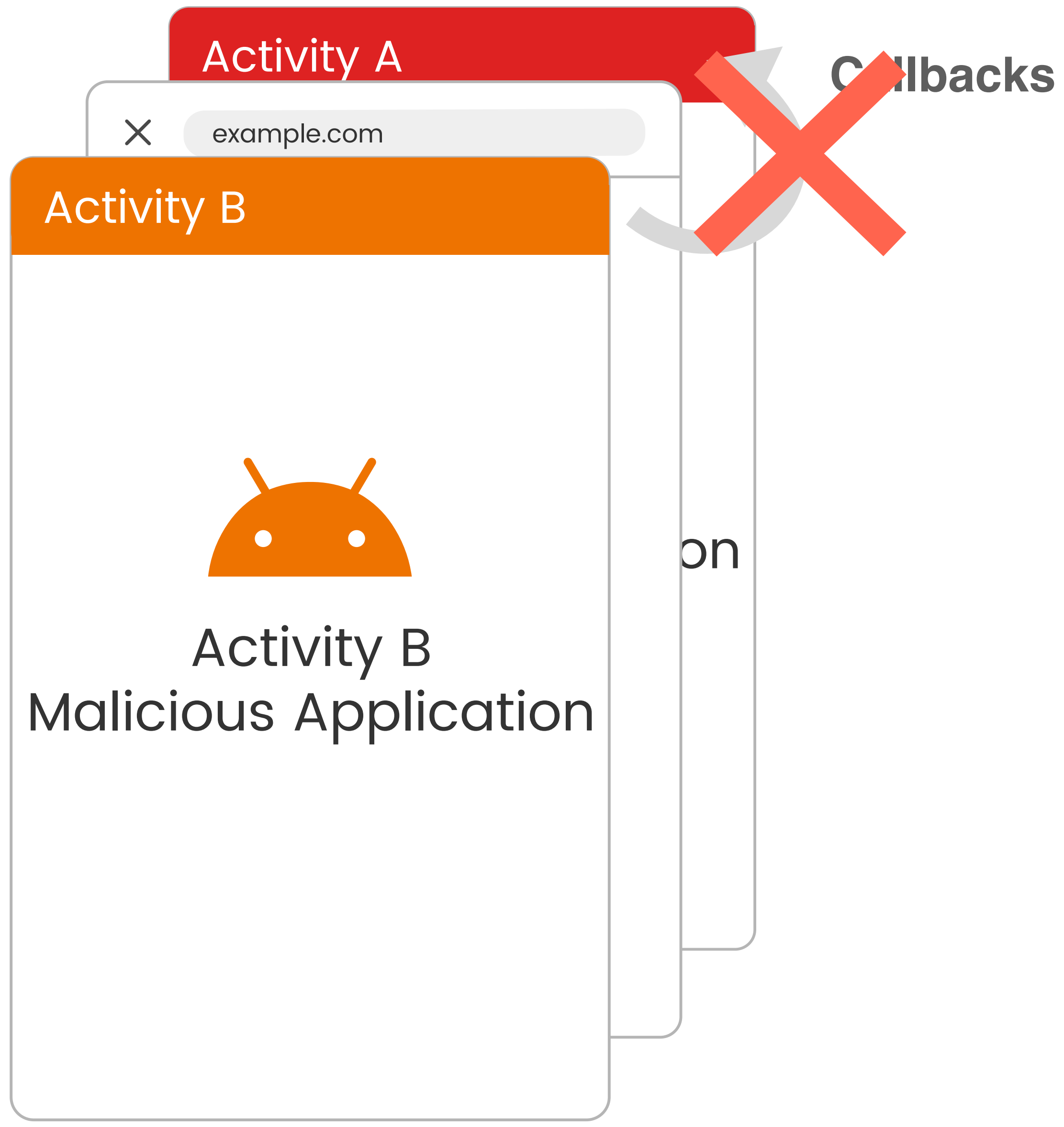


Hiding the Custom Tab

Mitigation



Custom Tab Provider:
Prevent callbacks on redirection
(prevents redirection-based attack)



Android OS:
Restrict callbacks to Custom Tabs in the foreground
(prevents stealthy attack)

Security Implications

- Opening website in Custom Tab is top-level navigation
- Cross-origin attack-targeted mitigations useless
- Allows to bypass
  - SameSite cookies
  - Framing Protection
  - Cross-Origin-Opener-Policy
  - Fetch Metadata

Headers	
connection	close
accept-language	en-US,en;q=0.9,de-AT;q=0.8,de;q=0.7,en-AT;q...
accept-encoding	gzip, deflate, br
sec-fetch-dest	document
sec-fetch-user	?1
sec-fetch-mode	navigate
sec-fetch-site	none
accept	text/html,application/xhtml+xml,application...
user-agent	Mozilla/5.0 (Linux; Android 11; AC2003) App...
upgrade-insecure-requests	1
sec-ch-ua-platform	"Android"
sec-ch-ua-mobile	?1
sec-ch-ua	" Not A;Brand";v="99", "Chromium";v="101", ...
host	webhook.site
content-length	
content-type	

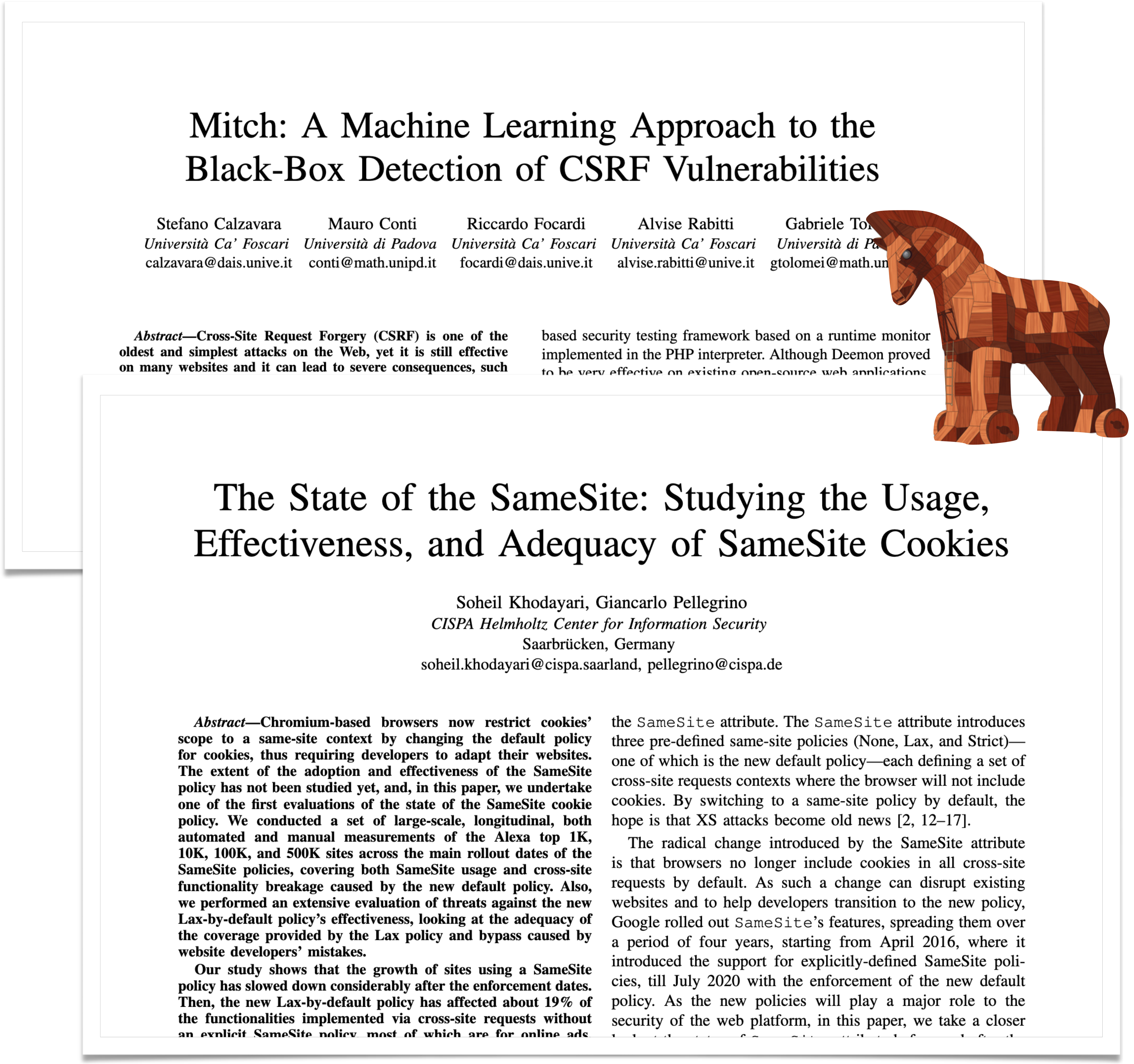
Chrome

Headers	
connection	close
accept-language	en-US,en;q=0.9,de-AT;q=0.8,de;q=0.7,en-AT;q...
accept-encoding	gzip, deflate, br
sec-fetch-dest	document
sec-fetch-mode	navigate
sec-fetch-site	none
accept	text/html,application/xhtml+xml,application...
user-agent	Mozilla/5.0 (Linux; Android 11; AC2003) App...
upgrade-insecure-requests	1
sec-ch-ua-platform	"Android"
sec-ch-ua-mobile	?1
sec-ch-ua	" Not A;Brand";v="99", "Chromium";v="101", ...
host	webhook.site
content-length	
content-type	

Chrome Custom Tab

Custom Tab CSRF

- 10.3% of state-changing requests still implemented using GET
- … sensitive state-changing POST requests can be exploited by changing to GET requests (e.g. IMDB, PayPal and Meetup)
- No detectable attack
- Allows to bypass even **SameSite strict cookies** on Chrome!



Mitch: A Machine Learning Approach to the Black-Box Detection of CSRF Vulnerabilities

Stefano Calzavara, Università Ca' Foscari, calzavara@dais.unive.it
Mauro Conti, Università di Padova, conti@math.unipd.it
Riccardo Focardi, Università Ca' Foscari, focardi@dais.unive.it
Alvise Rabitti, Università Ca' Foscari, alvise.rabitti@unive.it
Gabriele To…, Università di Pa…, gtolomei@math.un…

Abstract—Cross-Site Request Forgery (CSRF) is one of the oldest and simplest attacks on the Web, yet it is still effective on many websites and it can lead to severe consequences, such

based security testing framework based on a runtime monitor implemented in the PHP interpreter. Although Deemon proved to be very effective on existing open-source web applications

The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies

Soheil Khodayari, Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
soheil.khodayari@cispa.saarland, pellegrino@cispa.de

***Abstract*—Chromium-based browsers now restrict cookies' scope to a same-site context by changing the default policy for cookies, thus requiring developers to adapt their websites. The extent of the adoption and effectiveness of the SameSite policy has not been studied yet, and, in this paper, we undertake one of the first evaluations of the state of the SameSite cookie policy. We conducted a set of large-scale, longitudinal, both automated and manual measurements of the Alexa top 1K, 10K, 100K, and 500K sites across the main rollout dates of the SameSite policies, covering both SameSite usage and cross-site functionality breakage caused by the new default policy. Also, we performed an extensive evaluation of threats against the new Lax-by-default policy's effectiveness, looking at the adequacy of the coverage provided by the Lax policy and bypass caused by website developers' mistakes.**

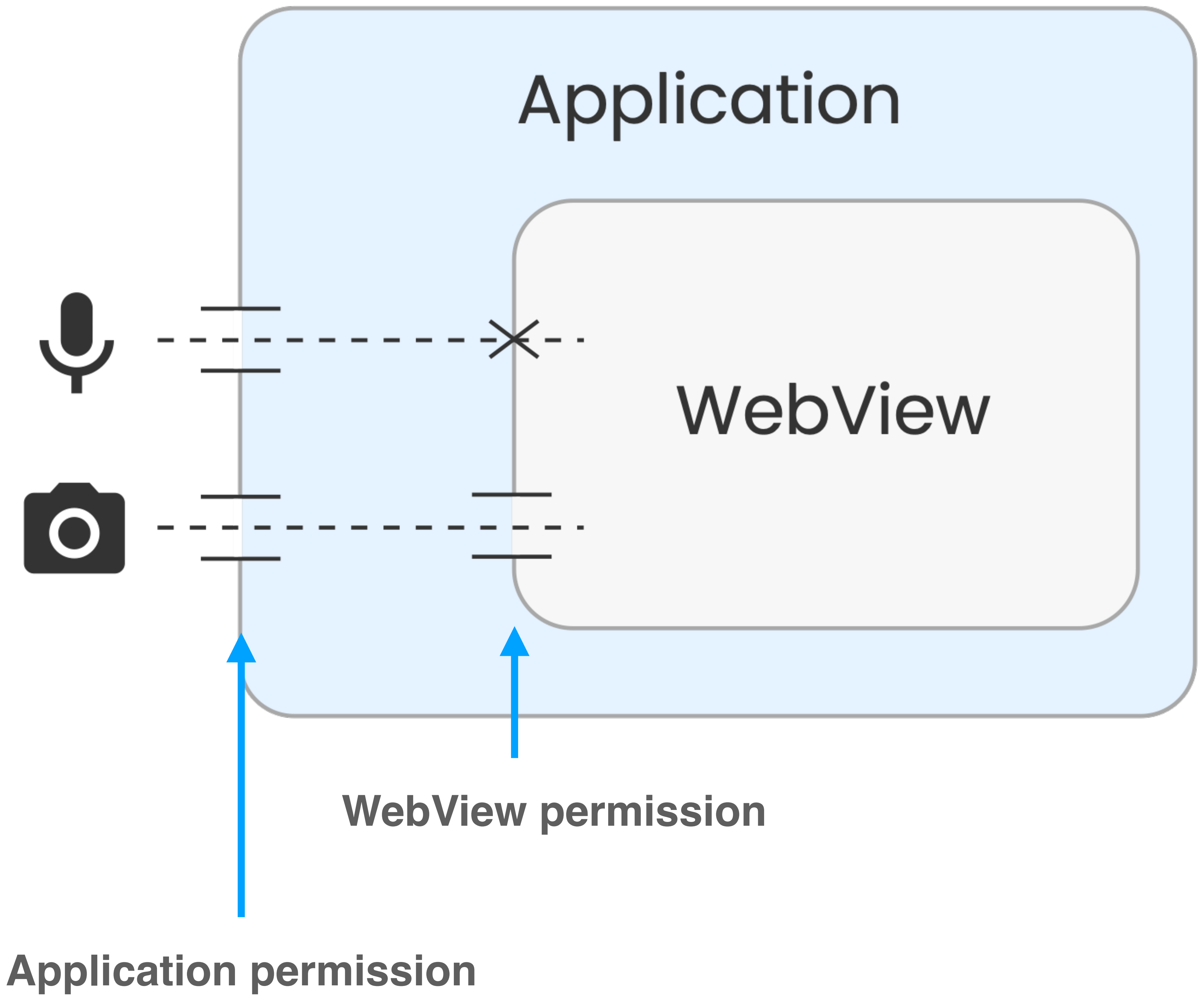
Our study shows that the growth of sites using a SameSite policy has slowed down considerably after the enforcement dates. Then, the new Lax-by-default policy has affected about 19% of the functionalities implemented via cross-site requests without an explicit SameSite policy, most of which are for online ads,

the SameSite attribute. The SameSite attribute introduces three pre-defined same-site policies (None, Lax, and Strict)—one of which is the new default policy—each defining a set of cross-site requests contexts where the browser will not include cookies. By switching to a same-site policy by default, the hope is that XS attacks become old news [2, 12–17].

The radical change introduced by the SameSite attribute is that browsers no longer include cookies in all cross-site requests by default. As such a change can disrupt existing websites and to help developers transition to the new policy, Google rolled out SameSite's features, spreading them over a period of four years, starting from April 2016, where it introduced the support for explicitly-defined SameSite policies, till July 2020 with the enforcement of the new default policy. As the new policies will play a major role to the security of the web platform, in this paper, we take a closer

Web View Attack 1/2

- Vulnerability in two popular WebView plugins for Android frameworks
  - React Native WebView
  - unity-webview
- Websites in WebView can access camera/ microphone, if
  - Application has permission
  - Application grants WebView permission
- Default: WebView permission denied



Web View Attack 2/2

- Two plugins by default grant permission to WebView
- Attacker loads malicious website into WebView of vulnerable app
- Access to camera & microphone
- Mitigation
  - Deny access by default
  - Implement access control mechanism by plugin developers
  - Show indicator when camera/microphone is used

Conclusion

- Custom Tab Attack
  - Abuse Custom Tab for XS-like attacks (state inference & CSRF)
  - Doesn’t trigger user-observable events
  - Defeats existing mitigations for XS attacks
- Web View Attack
  - Implementation flaw in Android framework plugins allows microphone/camera access to web attacker

Thank you!
Questions?

@beerphilipp

Backup: Preliminary Evaluation

- Analysed top 250 downloaded free applications on Google Play (247 successfully)
- 85 (34%) use Custom Tabs
- 57 (23%) use Custom Tabs Callback
- Web View attack app vulnerability:

Permissions	RN WebView	unity-webiew	Others
¬camera $\wedge$ ¬microphone	0	1 (< 1%)	113 (46%)
¬camera $\wedge$ microphone	0	0	28 (11%)
camera $\wedge$ ¬microphone	2 (< 1%)	0	32 (13%)
camera $\wedge$ microphone	5 (2%)	0	66 (27%)
camera $\vee$ microphone	7 (3%)	0	126 (51%)

Backup: Custom Tab Attack Code

```
val callback = object : CustomTabsCallback() {
    override fun onNavigationEvent(navigationEvent: Int, extras: Bundle?) {
        when(navigationEvent) {
            TAB_SHOWN -> {
                startActivity(Intent(this, OverlayActivity::class.java))
            }
            NAVIGATION_STARTED -> {
                onNavigationStarted()
            }
            NAVIGATION_FINISHED -> {
                onLoadingFinished()
            }
            NAVIGATION_FAILED -> {
                onLoadingFailed()
            }
            else -> { }
        }
    }
}

val connection = object : CustomTabsServiceConnection() {
    override fun onCustomTabsServiceConnected(name: ComponentName, client: CustomTabsClient) {
        session = client.newSession(callback)
        client.warmup(0)
    }
    override fun onServiceDisconnected(componentName: ComponentName?) { }
}

CustomTabsClient.bindCustomTabsService(context, packageName, connection)
val cctIntent: CustomTabsIntent.Builder = CustomTabsIntent.Builder(session).build()
cctIntent.launchUrl(context, Uri.parse(url))
```