Mobile Login Bridge: Subverting 2FA and Passwordless Authentication via Android Debug Bridge

Ahmed Tanvir Mahdad
Texas A&M University
College Station, TX, USA
mahdad@tamu.edu

Nitesh Saxena
Texas A&M University
College Station, TX, USA
nsaxena@tamu.edu

***Abstract*—Smartphones have become ubiquitous for a range of social, financial, and personal endeavors, as well as for accessing sensitive resources like confidential files from organizations. Nevertheless, this extensive usage has also made smartphones vulnerable to multiple security risks posed by malicious adversaries who intend to breach user accounts or steal personal information. Specifically, high-profile individuals or organizations are susceptible to becoming targets of targeted attacks. Previous research has identified various vulnerabilities that can compromise smartphones and access users' confidential information. A prominent example of such a vulnerability, known as the "Android Debug Bridge (ADB) vulnerability," is widely recognized as it enables an attacker to remotely access and manipulate an Android smartphone and perform malicious activities. However, the existing body of literature lacks a comprehensive examination of the implications of this vulnerability on modern authentication systems, web-based password managers, and financial and e-commerce applications.**

In this paper, we shed light on this area and evaluated the security of multi-factor authentication systems, browser-based password managers, and popular financial and e-commerce applications. For this purpose, we introduce the *BADAuth*[1] attack that exploits a set of ADB utilities. Our results reveal the susceptibility of secure authentication systems and browser-based password managers to a sophisticated one-time attack on a non-rooted device even with the latest Android version (Android 14.0). Furthermore, our research exposes the alarming ability of adversaries to access all passwords stored by browser-based password managers, thus paving the way for more severe attacks, including large-scale breaches within organizational settings. Additionally, our assessment underscores potential privacy and security risks for financial and e-commerce apps under *BADAuth* attacks, along with possible risk mitigation strategies.

I. Introduction

Smartphones have become an essential device for many online activities, including social networking, banking, and online transactions, as well as phone calls and text messages. As a result, they have become lucrative targets for cybersecurity breaches, due to the increasing use of mobile applications for financial activities. Malware is the primary method of conducting these attacks. Malware can infiltrate users' devices through various channels, including malicious applications, email attachments, and browser extensions.

[1]Named after "**B**ridge **A**ndroid **D**ebug for malicious **Auth**entication"

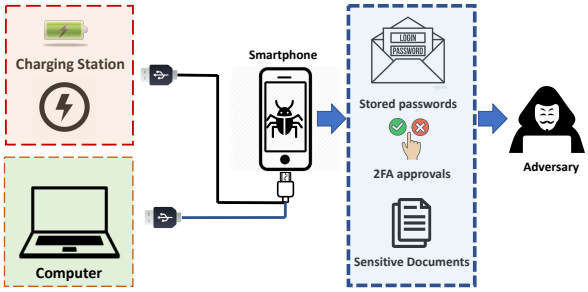


Fig. 1: Overview of *BADAuth* attack on Android smartphones.

Malware can infiltrate smartphones via USB connectivity, in addition to the typical infection methods. In such instances, attackers can take advantage of the Android Debug Bridge (ADB), a tool that facilitates communication and control between a computer and a smartphone, to infect the smartphone when the 'USB debugging' mode is enabled. This mode grants elevated permissions to the connected device, enabling attackers to carry out malicious activities.

Researchers have pointed out the possible malicious use of ADB and associated risks. For example, they leveraged the elevated permissions gained from `ADB Shell` and demonstrated event recording and injection, thus mimicking user actions without active user involvement. Mohamed et al. [1] and Gomez et al. [2] showed how adversaries can use record-and-replay of touch events to record and then inject touch events to mimic user actions. Researchers also reported private data infiltration, including authentication credentials such as passwords, using ADB commands like screenshot [3], [4], [5]. Additionally, Meng et al. [6] conducted app analysis on the Google Play store to detect apps that still use ADB commands in their workflow, posing a potential risk to app users.

In recent versions (Android 9.0 and 10.0), Android made some permission changes [7] that restrict injecting touch event data using `ADB Shell`. Furthermore, in recent versions, third-party applications are no longer authorized to view the contents of the "/data/local/tmp" directory, primarily used for running executables on the device. With these changes, record-

TABLE I: Attack Summary on 2FA and passwordless Schemes

Scheme	Variant	Attack Approach	Successful Attack
Security key	Built-in Security Key	Inside-adversary	✓
		Remote-helper	✓[1]
Push Notification	Confirm	Inside-adversary	✓
		Remote-helper	✓
	Compare-and-Confirm	Inside-adversary	✓
		Remote-helper	✓
	Select-and-Confirm	Inside-adversary	✓
		Remote-helper	✓
One Time PIN	SMS-OTP	Inside-adversary	✓
		Remote-helper	✓
	Software Token	Inside-adversary	✓
		Remote-helper	✓

✓- Successful, ✗- Not successful

[1] Remote-Helper is unable to defeat the built-in security key protected by the titan-m chip in Pixel phones, but it has been observed to be successful in exploiting other phone models.

and-replay attacks reported in [1], [2] will no longer work and pose a risk to users.

With the increasing prevalence of smartphones in daily life, they are becoming a more popular factor in multi-factor authentication. Modern authentication systems, such as security keys and push notification authentication, increasingly use smartphones as a possession factor. Although previous research has identified the risk of information theft [3], [4], [5], including passwords through screenshots and record-and-replay event attacks, there has been no evaluation of the security of modern authentication systems, browser-based password managers, and popular smartphone financial apps in the presence of the ADB vulnerability. Most importantly, no research has evaluated the present state of the risk of the ADB vulnerability in the latest Android version. Additionally, no effective mitigation recommendations have been suggested to prevent this attack.

To address this research gap, we design the attack framework *BADAuth*, which utilizes a series of ADB commands to generate touch events, button presses, and other events programmatically, eliminating the need for touch or button press event injection. Unlike previous record-and-replay attacks [1], [2], *BADAuth* is effective against the latest Android version (Android 14.0). We conducted an assessment of the security of modern authentication systems that utilize smartphones as possession factors, as well as browser-based password managers that sync password vaults with smartphone versions. We identified vulnerabilities that can lead to compromises in password vaults and authentication systems in the presence of *BADAuth*. Finally, we propose effective mitigation techniques that can safeguard against *BADAuth* and similar automated agents. The overview of the attack is shown in Figure 1.

Our Contribution: Our contribution to this paper is four-fold:

1) **Proposed alternative ADB-based attack framework, *BADAuth*, distinct from record-and-replay attacks**: Our proposed attack framework, *BADAuth*, employs ADB utilities to automate user actions on smartphones, in contrast to the record-and-replay attack which depends on touch event injection. This attack has the ability to function efficiently on all versions of Android, including the most recent one (Android 14.0).
2) **Evaluation and Demonstration of *BADAuth* attack on real-world 2FA and passwordless systems**: We evaluated the security of real-world 2FA systems in the presence of *BADAuth* that impersonates user actions on Android devices and demonstrated the results. We used two approaches: inside-adversary (where both the attack initialization and 2FA verifications were done in the same device) and remote-helper (where the attack was initiated outside of the device and verification was done from the target smartphone) in the evaluation process. The summary results are shown in Table I.
3) **Unveiling the vulnerabilities of browser-based password managers**: We have demonstrated the capability of *BADAuth* to exploit vulnerabilities in popular browser-based password managers, including Google Chrome. With *BADAuth*, an external adversary can easily export all stored passwords in clear text from browser-based password managers. This poses a significant security risk as it allows malicious entities to gain unauthorized access to passwords that may not have been used on the smartphone but are stored within the password manager.
4) **Security Analysis and Countermeasures for Popular Android Apps with Sensitive Financial and E-commerce Services**: Considering the capability of *BADAuth* that impersonates user actions on Android devices, we analyzed ten popular and highly rated Android apps that provide the sensitive financial services, such as banking and credit card management. We identified potential vulnerabilities in their workflow that can be exploited by *BADAuth* and suggested effective mitigation strategies that can prevent similar attacks.

The video demonstrations of the implemented attacks on 2FA and passwordless systems are presented at https://sites.google.com/view/badauth/home.

II. Background

Two-factor Authentication (2FA) Systems: In 2FA systems, in addition to the password, the user has to prove possession of a device known as a "possession-factor" device. To successfully defeat the 2FA systems, the adversary has to compromise both the user terminal (where the knowledge factor is compromised) and the possession-factor device.

Security Key: The security keys utilize the FIDO2 protocol for user presence verification [8]. They employ tamper-proof secure hardware, such as lightweight key-like devices like Yubikey, for secret key computation and storage. The integrity of the secured chip is maintained, preventing tampering or unauthorized access by the host OS or malicious programs. Another approach, known as the "Built-in Security Key," utilizes the smartphone itself as a security key. The key is computed and securely stored on the smartphone. Google Pixel phones incorporate a dedicated tamper-proof chip known as "Titan-m," while other devices utilize an isolated area within the phone's processor for the same purpose.

Push Notification Authentication: It is another recent addition to 2FA systems. Here, in addition to the password, the service sends a push notification to a pre-registered device (e.g., smartphone). Users have to establish their presence by tapping on the "Approve" button there. Recently, it has gained popularity as the primary 2FA system for online services (e.g., Google [9]), password managers (e.g., LastPass [10]), and organizational settings (e.g., Duo [11]).

One Time PIN (OTP): OTP can be generated in a smartphone app (or any hardware token) or communicated via SMS (Short Message Service). The user has to enter the OTP into the user terminal to complete the authentication.

Passwordless Authentication Systems: In passwordless authentication systems, they utilize possession factors (e.g., smartphones) as an alternative to passwords. Additionally, smartphone-based passwordless systems require the phone lock to approve a notification to authenticate.

Browser-based Password Managers: Password managers are essential tools for managing and safeguarding complex passwords. Generally, password managers are built-in functionality in the major browsers (e.g., Chrome, Firefox) that allow users to enable "sync" to use saved passwords across all their devices. Password managers can also be implemented by third-party providers (e.g., LastPass, 1Password). In this work, the primary focus is on the security of browser-based password managers in the context of our designed attack framework.

III. The Risk Associated with ADB

A. Android Permission Model

The Android OS utilizes the "Application Sandbox" model, which is derived from Linux-based operating systems, to ensure the security of app resources. This model effectively isolates app resources and restricts an app's access to resources that have not been explicitly allocated to it. This helps protect other apps and the operating system from unauthorized or malicious activities. Furthermore, the Android OS assigns different permission levels to apps based on their specific functionalities and requirements.

Install-time Permissions: During installation, an app is granted install-time permissions. App developers are required to declare necessary permissions, which are then automatically granted during the installation process. These permissions are categorized as "Normal Permissions," which are considered low-risk, and "Signature Permissions," which are granted when the signed certificate of the permission-seeking app matches with a pre-defined app.

Runtime Permissions: Runtime permissions are categorized as "Dangerous Permissions" as they grant an app access to data and resources beyond its sandbox, allowing actions on private user data. App developers are required to explicitly declare these runtime permissions, and users will always see a prompt to allow or deny such permissions.

B. How Malware Exploits ADB Permissions

Android Debug Bridge (ADB) consists of three components. The first component is the ADB client, located on the development machine, which is responsible for sending commands. The second component is the ADB Daemon, which runs as a background service on the device and executes the received commands. Lastly, the ADB server manages the communication between the client and the daemon.

The "ADB shell" and command-line ADB utilities grant elevated permission levels, allowing unrestricted access to device resources. This feature is intended to aid app developers during development without requiring explicit runtime permissions. While beneficial for developers, it can also be exploited by malicious entities for nefarious purposes.

For example, the ADB utilities "screencap" and "screenrecord" can be used to capture screenshots or record the screen without any sound or visual cues, unlike the default screenshot utility of Android. This capability poses a risk as it enables the recording of a user's activities during sensitive operations (e.g., logging into a bank account). Additionally, the ADB utilities "input tap," "input key event," and "input text" can inject touch events, button presses, and text input into text boxes, respectively. This allows an attacker to automate user actions, such as initiating authentication requests or approving 2FA prompts, without requiring any user involvement.

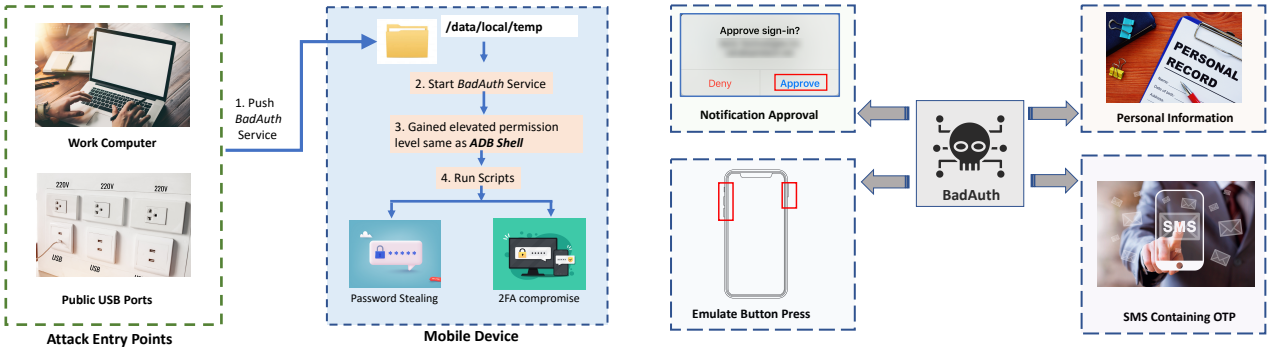
The ADB shell on an Android smartphone has full read/write permission to the directory `/data/local/tmp`. If the phone's USB debugging is enabled and it is connected to a malicious computer or port, an attacker could push an executable service to this directory. The executable would inherit the same elevated permissions as the ADB shell.

IV. Attack Approach and Design

A. Attacks, Adversary and their Motivation

*1) **Adversary Model**:* Our proposed adversary model encompasses two distinct types of adversaries. This model aims to provide a comprehensive understanding of the potential threats faced by organizations and individuals.

Adversary Type 1: These adversaries are motivated to infiltrate the systems of large organizations and acquire valuable and confidential data. They specifically target individuals within these organizations holding key roles (e.g., developers, systems admins), who are typically technical experts. The primary focus of these adversaries is to exploit vulnerabilities in their targets' smartphone browsers with the goal of obtaining sensitive organizational passwords, thereby compromising the organization's security. We assume that the target user has already logged into their browser profile and enabled password synchronization. Additionally, we assume that the adversary is already aware of the user's phone PIN, which they can acquire a smartphone PIN by employing well-known touch stroke logging attacks, as described in [12], [13]. These automated touch loggers can be utilized alongside *BADAuth* to record and transmit the target users' PIN to adversaries. We refer to the attack involving adversary A as *Attack A*.



(a) Overview of the workflow for malicious program injection and activation on an Android smartphone.

(b) An overview of malicious program capability on secure authentication systems and private information stored on mobile devices.

Fig. 2: An overview of malicious program injection process and attack capability.

Adversary Type 2: This category of adversaries focuses on advanced Android users who engage in activities such as using custom Android ROMs, performing Android testing, or serving as administrators for social networks used by prominent figures such as politicians and actors. Their primary objective is to gain unauthorized access to the personal accounts of these high-profile individuals, which typically employ advanced two-factor authentication (2FA) methods for enhanced security. We denote this attack as *Attack B*.

*2) **Attack Entry Points**:* First, we assume that the target users, being advanced users themselves, willingly enable the "Developer Options" on their Android devices. This is motivated by their interest in utilizing advanced tools to assess the performance of their developed applications or custom operating systems. These advanced users also include developers who use their smartphones on trusted work computers. This common practice is also evidenced by discussions among application developers in online forums [14]. They prefer real smartphones over emulators for more precise user experience evaluation [15]. Additionally, they typically test apps on multiple OS versions and devices [16]. These needs prompt them to use their own phones, with USB debugging enabled, for app testing. Since these workstations are trusted, they would not encounter any USB debugging prompts. This workflow is practical, given their daily work habits.

Additionally, adversaries possess the capability to persuade target users to install seemingly benign Android applications, which eventually prompt them to enable developer options and USB debugging as a prerequisite for unlocking enhanced functionality within these applications. It is important to note that, the adversaries do not have physical access to the target user's workstation.

We consider two scenarios for *BADAuth* infections on target devices, one for each type of adversary.

Scenario 1: We assume that adversaries are capable of persuading target users to install malicious program executables on their trusted workstations. This can be achieved through various deceptive means, such as convincing the target user to install seemingly harmless software on their host computer, such as a laptop, or enticing them to click on a malicious link that initiates the download of these malicious files and executables on their computer. In this particular scenario, the targeted users willingly connect their smartphones to their trusted workstations for working purposes (as discussed before), unaware of the presence of *BADAuth*. Consequently, the user does not encounter a prompt to allow USB debugging, further facilitating the adversary's purpose.

Scenario 2: In Scenario 2, we consider a situation where the targeted users unknowingly connect their smartphones to a malicious charging port located in a public setting, such as an airport. These compromised charging ports can be connected to a host computer under the control of adversaries, housing malicious programs and executables. Given that the targeted users are advanced Android users, we assume that they would have already activated the developer mode and enabled USB debugging on their devices.

Users are prompted to allow USB debugging when the host computer is unfamiliar to the target smartphone. A previous study [17] showed that only 17% of users pay attention to the Android permission prompts. Most users grant permissions quickly, exhibiting a skip-through behavior, to accomplish their desired task as soon as possible. For example, the desired task might be using a utility app while installing an application, or making an important call or message while charging the smartphone at the airport. Based on this observation, we assume that the target users will accept the USB debugging prompt without reviewing its contents carefully, and thus expose their devices to potential attacks.

*3) **Practicality of the Adversary Model**:* In this section, we assess the practicality of our assumptions in the adversary model.

Learning username and passwords: For *Attack A*, we assume that the attacker is not aware of the target user's authentication credentials, such as their username and password for any service. On the other hand, for *Attack B*, we assume that the attacker has knowledge of the target user's username (often a public email address) and password through one of the following methods: a previous Attack A, common cyberattacks, or known password vulnerabilities (e.g., phishing, keylogger, large-scale password leaks [18], [19], reuse of

leaked passwords on more sensitive sites). According to the ForgeRock Consumer Identity Breach Report, two billion records, including usernames and passwords, were stolen in 2021, representing a 35% increase compared to the previous year [20]. Therefore, obtaining a targeted user's username and password is a realistic threat rather than a strong assumption for adversaries.

Furthermore, this assumption aligns with previous works (e.g., work of Jacomme et al. [21]), which commonly adopt the attacker's knowledge of the target user's password for analyzing the efficiency of the second factor (possession factor) when the password is compromised.

Public USB charging port. USB charging ports, specifically those commonly found in airports, are a potential attack vector for cyberattacks on mobile devices. This security threat has been extensively recognized and studied by researchers (e.g., Tian et al. [22]). Several defense mechanisms have been proposed by researchers and practitioners to prevent or mitigate such attacks (e.g., [23]). Moreover, authoritative organizations such as the US Federal Communications Commission [24] and the US Federal Bureau of Investigation [25] have acknowledged this practical and significant threat posed by USB charging ports in today's context.

USB Debugging: We assume that the target users of both Attack A and Attack B are advanced Android users who activated the *developer option* on their smartphones, either for their own purposes or by following some instructions from a malicious source. This assumption is supported by a previous survey from the literature [26] that reported that 16.3% of users consistently have USB debugging turned on, while 41.9% of them are unaware of the status of USB debugging. As discussed in previous subsections, the adversary can also persuade unsuspecting users to activate the Developer option through any application installation guide and then enable USB debugging.

B. Understanding the Attack Mechanism

In our analysis, we used three devices: OnePlus 3T running Android 9.0, OnePlus 7T running Android 11.0, and Google Pixel 5 running Android 14.0. To implement the attack modules, we utilized the Android Native Development Kit (NDK), which allows the execution of C or C++ executable programs within an Android device.

Leveraging ADB Utilities to Design the Attack: Adversaries utilize standard ADB utilities such as `Input Tap`, `Input Swipe`, and `Input Keyevent` to simulate touch and button press events. Another utility, `Input Text`, is used for entering text in various textboxes, including password fields. It is also helpful in navigating to a specific web page, performing searches, and installing helper apps from Google Play. Additionally, adversaries can start and stop Android applications (e.g., browser, messaging app, OCR app, text editor app) on the smartphone during the attack workflow. For this purpose, they make use of ADB utilities such as `am` and `monkey`. To paste text copied to the clipboard, adversaries utilize the `keyevent 279` command. Furthermore, the `sleep` utility is used to introduce necessary delays.

TABLE II: Examples of ADB command chain to automate user actions (Google Pixel 5)

Task	Command Chain
Username and password submission	am start -a android.intent.action.VIEW -d http://twitter.com/login && sleep 2 input text username && sleep 1 && input tap 239 724 && sleep 2 && input tap 239 724 && sleep 1 && input text password && sleep 1 && input tap 942 1449
Screenshot capture	screencap -p /sdcard/example.png
Screen video recording	screenrecord –time-limit 70 /sdcard/azure.mp4
Application launch	am start -n com.azure. authenticator/com.azure.authenticator.ui. MainActivity && sleep 2
Stop application	am force-stop com.azure.authenticator

TABLE III: Adversary task, conditions, and permission

Adversary Task	**USB Connectivity**	**Permission**
BADAuth injection to device	Required	Required [1]
Touch event Injection	Not Required	Automatically Granted
Hidden Screenshot	Not Required	Automatically Granted
Hidden Screenvideo	Not Required	Automatically Granted

Automatically Granted - Uses ADB Shell Permission

[1] Only required at first-time connection, which also users tend to approve without checking [17]

Generate Binary Executable: Adversaries leverage the Android Native Development Kit (NDK) to compile attack programs into binary executables. These programs, coded in C, enable adversaries to simulate user actions programmatically, including generating screen tap events, emulating button presses, launching and terminating applications, capturing screenshots, and recording screen videos. The execution of these actions is orchestrated through a sequence of ADB commands linked together in a chain. Table II provides examples of such ADB command chains. Additionally, the adversary creates an additional shell script file (start.sh) to initiate the program for the first time.

***BADAuth* Infection on Device**: Adversaries place the executable and shell script file in a designated directory (`/data/local/tmp`) on an Android phone. Once the executable file is initiated, it becomes independent of the USB connection, allowing the script to execute malicious activities later without relying on the USB connection. The executable will persist until the device is restarted or the process is terminated. To maintain screen activity, adversaries can dispatch random tap events at regular intervals.

An overview of the injection of the malicious program and its attack capabilities is presented in Figure 2. Additionally, Table III outlines the adversary's capabilities at various stages of the attack workflow.

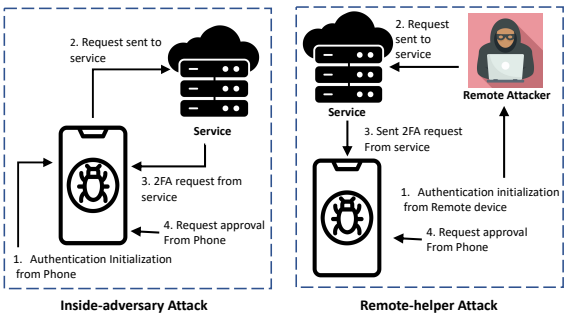


Fig. 3: Workflow of Inside-adversary and Remote-helper Attack

C. Attack Workflow

As mentioned previously, the adversary can initiate two types of attacks, referred to as *Attack A* and *Attack B*. *Attack A* specifically targets browser-based password managers, while *Attack B* constitutes a broader attack on secure 2FA and passwordless systems in general.

*1) **Attack A**:* The primary objective of this attack is to steal passwords stored across multiple devices within the same user profile of browsers. The attacker seeks to export all saved passwords or target specific ones, including organizational passwords, and transmit them to a remote adversary. We present a general workflow for the process of password theft from mobile browser apps. As a representative example, the steps of an attack on the Google Chrome password manager are discussed here:

Step 1: *BADAuth* starts the Google Chrome browser and navigates to *https://passwords.google.com*.

Step 2: It generates tap events to open settings and taps on ”Export Passwords”.

Step 3: In the next step, a verification prompt will appear that asks for the password. *BADAuth* inserts text with already known passwords in that field and submits it.

Step 4: All passwords are downloaded as a (.csv) file in the *Downloads* folder.

Step 5: *BADAuth* navigates to *https://mail.google.com*, types a new email, attaches the exported passwords, and sends them to an external adversary.

Step 6: *BADAuth* deletes the sent email from the ”Sent” folder of the email and deletes the downloaded file from the *Downloads* folder to remove attack traces.

BADAuth can steal all passwords saved across various devices, including those used for work, within a specific profile, and transmit them to external adversaries. This capability poses a significant risk of a large-scale organizational breach, as the attacker may exploit it as a gateway to infiltrate a secure network. An example of exported authentication credentials is illustrated in Figure 4.

*2) **Attack B**:* To implement *Attack B*, the *BADAuth* employs two approaches: the inside-adversary attack and the remote-helper attack. An overview of these approaches is depicted in Figure 3.

Inside-adversary attack on Security Key:

Step 1: The executable program initiates the attack when it detects no user activity for a specific period of time (e.g., 30 minutes).

Step 2: Next, the executable program sends an authentication request using previously collected credentials.

Step 3: The Android built-in security key system does not necessitate user presence verification through a button press or push notification when the authentication request originates from the same device. Instead, users are required to verify their phone lock (e.g., fingerprint) to approve the request. In the case of fingerprints, a fallback option, ”Use PIN,” is displayed on the screen. This can be exploited by *BADAuth*, enabling it to programmatically input the user’s known PIN, authenticate, and thereby bypass the security provided by Android’s built-in security key.

Titan-m chip introduced some extra security features (e.g., acts as a secure boot controller, checks the digital signature of boot image) to safeguard the user from malicious OS [27]. However, using the inside-adversary attack, the *BADAuth* can bypass the security added by the “Titan-m” chip in Google Pixel phones. We have shown an example of an inside-adversary attack on phones with Titan-m chip in our demonstration.

Inside-adversary attack on Push Notification Authentication:

Step 1: *BADAuth* would send an authentication attempt to the service. Anticipating the push notification to arrive a few seconds after the authentication attempt, *BADAuth* activates itself accordingly.

Step 2: When the push notification with the approve or deny button appears on the phone’s screen or in the notifications, *BADAuth* injects a tap event at a specific coordinate to approve the notification without requiring active user involvement.

Step 3: In the *select-and-confirm* variant, *BADAuth* captures a screenshot of the challenge, which includes all buttons with numbers and the actual identifier. It then analyzes the screenshot using a third-party OCR app and the built-in analyzer in the *BADAuth* service, generating a tap event for the correct button to approve it. A sample attack on this variant is shown in the demonstration.

Inside-adversary attack on SMS-OTP: Here, *BADAuth* can collect OTP by exploiting the ”tap-to-copy” feature in SMS notifications. This vulnerability was initially addressed by Lei et al. [28], and we incorporate this concept into our design, demonstrating an attack on SMS-OTP. The detailed steps are described below.

Step 1: After an authentication attempt, the SMS containing the OTP is usually received on the phone within a few seconds. The Android OS extracts the OTP from the SMS and displays it in the notification with a tap-to-copy button. *BADAuth* activates itself shortly after the authentication attempt and injects a tap event in the notification to copy the OTP to the clipboard.

TABLE IV: Security of Browser-based password managers - Observation from Attack A

Browser Name	Access Password Vault			Security During Action in Password Vault		
	Password Re-quired?	2FA required?	Phone Lock required?	All Password export in plaintext?	Display Single Password in plaintext?	Copy password in plaintext?
Google Chrome	✗	✗	✓	phone lock	phone Lock	No Security
Microsoft Edge	✗	✗	✓	N/A	phone lock	No Security
Mozilla Firefox	✗	✗	✓	N/A	phone lock	No Security
Opera	✗	✗	✓	N/A	phone lock	No Security

✓– Yes, ✗- No

Step 2: In the next step, *BADAuth* pastes the OTP using another ADB utility (`keyevent 279`) into the designated text field on the authentication form and submits it to complete the authentication.

Inside-adversary attack on Software Token OTP: In software token, OTPs are generated on the phone through a specific generator app (e.g., Google Authenticator). These authenticator apps employ the `FLAG_SECURE` flag, preventing users or any automated agent from taking screenshots or recording screen videos. To circumvent this security measure, *BADAuth* utilizes ADB utilities (e.g., "am") to initially launch the software token app and then copy specific OTPs using the "tap-to-copy" feature. This attack is shown in our demonstration, and the detailed steps are as follows:

Step 1: After sending the authentication request to the authentication service, *BADAuth* would open the targeted software token app (e.g., Google Authenticator) and exploit the tap-to-copy feature to copy the desired OTP.

Step 2: In the next step, *BADAuth* would paste the OTP (using `keyevent 279`) in the specific text field on the authentication form and inject a tap event to submit it and complete the authentication.

Remote-helper attack on Security Key: The Android built-in security key on Google phones (e.g., Pixel series phones starting from Pixel 3) requires a volume-down button press to confirm the user's presence. This physical volume-down button is directly linked to the Titan-m chip within Google Pixel phones. Consequently, attempting to emulate a button press using ADB utilities cannot transmit the response to the Titan-m chip, resulting in the failure of user possession establishment. Thus, the remote-helper attack on the security key is unable to compromise the security provided by the Titan-m chip.

Remote-helper attack on Push Notification Authentication:

Step 1: *BADAuth* injects a tap event on a specific position of the "Approve" button in the notification.

Step 2: For the select-and-confirm variant, the remote adversary would send the unique identifier to *BADAuth*, which it uses to generate a tap event on the correct button after the analysis.

Remote-helper attack on OTP: A remote attacker can instruct *BADAuth* to capture a screenshot of the OTP or tap on the tap-to-copy button (applicable to both SMS-OTP and software token). In the subsequent step, the attacker can communicate the extracted OTP or screenshot, which can then be used to carry out the malicious authentication attempt.

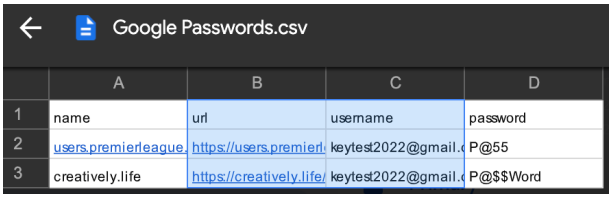


Fig. 4: Screenshot of CSV file exported from Chrome Password Manager

After accomplishing its attack objectives, *BADAuth* can eliminate all traces of the attack, including closing any apps opened for the attack and uninstalling helper apps.

V. Evaluation

A. Attack on Browser-based Password Managers

We investigate the security of mobile browsers utilizing cross-device password synchronization, facilitating users in accessing saved passwords across various devices. Our focus is on the potential threat posed by *BADAuth* capable of compromising password storage on mobile devices. We assess the security measures implemented by different browsers, including password re-verification, 2FA, and screen locks, aimed at preventing unauthorized access to the password vault. The summarized findings are presented in Table IV. Notably, we observe that none of the browsers necessitate password or 2FA re-authentication once the user is logged into the browser profile, but they do enforce a screen lock as a precaution to reveal saved passwords.

We identify a vulnerability in Google Chrome that allows an adversary to export all the user's saved passwords in plain text without any password re-verification. The only security measure required is a screen lock, which can be bypassed by *BADAuth* using downgrade attacks. The adversary can then transmit it to the external adversary.

In other browsers, *BADAuth* can utilize the tap-to-copy feature to capture desired stored passwords, subsequently saving and transmitting them to the remote adversary. We observed similar screen lock downgrading vulnerabilities in other browsers as well, which *BADAuth* can exploit to fulfill its objectives.

Based on these observations, it is evident that *BADAuth* can steal passwords from browser-based password managers' vaults.

B. Attack on 2FA Systems

Attack on Security Key: Among the security key variants, our specific focus is on the built-in security key that involves smartphones in its workflow. Typically, *BADAuth* can inject

TABLE V: Evaluation with Free Android Anti-malware Programs

Anti-malware Program	File Scan	Real Time Protection	Malicious File Warning	USB Debugging Warning
Avast mobile	✗	✗	✗	✓
Kaspersky Free Internet for mobile	✗	N/A	✗	✗
Lookout Mobile Security	✗	✗	✗	✓
BullGuard Mobile Security	✗	✗	✗	✗
Panda Dome Mobile Antivirus	✗	✗	✗	✓
Sophos Intercept X	✗	✗	✗	✓
AVG mobile	✗	✗	✗	✓
Bitdefender free antivirus	✗	N/A	✗	✗

✓– Detected, ✗- Not Detected

touch events to automatically approve user presence. However, our observations reveal that the Titan-m chip in Pixel smartphones prevents *BADAuth* from bypassing user presence verification. Here, to establish user presence, the user must physically press the 'Volume-down' button, which is directly connected to the chip, making it impossible for *BADAuth* to emulate by injecting a button press event.

However, in the case of an inside-adversary attack, where the authentication attempt is initiated from the same device, the security key does not require a volume-down button press. Instead, it only requires the establishment of a screen lock, easily achievable by *BADAuth* using a downgrading attack. Consequently, the security provided by Titan-m can be bypassed by *BADAuth* in such cases.

Attack on Push Notification Authentication: We examine the security of all push notification authentication variants. Generally, *BADAuth* can inject touch events to approve push notifications for both the confirm and compare-and-confirm variants. In the select-and-confirm variant, *BADAuth* possesses the capability to solve the challenge by taking a screenshot, analyzing it, and tapping the correct button. We demonstrated all these attacks in our demonstrations.

Attack on One-Time PIN (OTP): We demonstrate that *BADAuth* can overcome all variants of OTP authentication. It can copy the OTP from the notification or the messaging app and paste it into the OTP entry field or send it to the remote attacker. Additionally, it can access software tokens and copy the OTP from them. All these attacks are showcased in our demonstrations.

We present our attack summary in Table I.

C. Detectability

Anti-malware Programs: Anti-malware programs detect unwanted and harmful programs on devices by comparing known signatures, suspicious permissions, and activities. They display warnings about any suspicious apps and processes. We tested *BADAuth* against popular anti-malware apps for Android. We used Avast Mobile [39], Kaspersky Mobile Internet [40], Lookout Mobile Security [41], Avira Mobile Antivirus [42], BullGuard Mobile Security [43], Panda Dome Mobile Antivirus [44], Sophos Intercept X [45], AVG Mobile Antivirus [46], and Bitdefender Mobile Antivirus (Free Version) [47]. These anti-malware programs provide both file scanning and real-time protection (except Kaspersky and Bitdefender). However, none of them detected or warned about *BADAuth* during its operation. Some apps warned about USB debugging being enabled. Table V summarizes our observations.

Impact on Device Resources: Consuming a high amount of resources can trigger warnings from anti-malware programs against any process. To assess this, we measured the resource consumption of *BADAuth* in both idle and active states and observed that it utilizes a minimal amount of device and network resources. This should not trigger alarms in anti-malware programs. Detailed information about resource usage is provided in Table VII, indicating minimal CPU and memory consumption (nearly 0.01% of total available resources). The virtual image (amount of virtual memory used by the process), resident size (non-swapped physical memory used by the process), and shared memory size usage are also minimal, both in idle and peak states (when *BADAuth* injects touch events or takes a screenshot).

D. Application Analysis

To assess the threat posed by *BADAuth*, we conducted an analysis of its impact on real-world finance and e-commerce smartphone applications. We selected ten popular apps from the Google Play Store with over 10 million downloads and high user ratings. Our evaluation focused on their privacy protection mechanisms, authentication methods, and payment procedures, aiming to determine whether these apps can effectively maintain privacy and ensure user presence during sensitive operations in the presence of automated agents similar to *BADAuth*.

Privacy restriction. To mitigate the risk of silent screenshots from *BADAuth* or similar automated agents, financial apps should utilize the `FLAG_SECURE` to prevent screenshots. However, our analysis (see Table VI) revealed that none of the financial, e-commerce, and wallet apps we examined implemented restrictions on screenshots and screen video capturing. Consequently, *BADAuth* can effortlessly capture screenshots of account information, statements, recent order details, and transaction details from the background without notifying users.

Authentication.: One of the benefits of persistent user authentication is that it enhances the user experience for online shopping and social networking platforms. However, applications that involve sensitive financial transactions, such as banking services, typically enforce a short authentication session duration for security purposes. As shown in Table VI, the majority of the financial applications examined in this study require user authentication every time the application is launched. The primary authentication method for all applications is a username/password combination, but biometric authentication is also available as an alternative option. However, passwords are still required as a fallback mechanism in case biometric authentication fails.

According to the threat model presented in this paper, passwords cannot provide adequate security in the presence

TABLE VI: Risk analysis of popular finance, e-commerce and wallet application in the presence of *BADAuth*

App Category	App Name	Privacy Restriction		Authentication				Payment / Fund Transfer	
		Screenshot	Screen Record	Authentication Required?	Username/ Password	Screen lock	Screen lock downgrading option	Verification method	Screen lock?
Finance	Wells Fargo [29]	✗	✗	✓	✓	✓	Password	OTP	✗
	Paypal [30]	✗	✗	✓	✓	✓	Password	Captcha	✗
	Discover [31]	✗	✗	✓	✓	✓	Password	OTP	✗
	Remitly [32]	✗	✗	✗	✗	✗	N/A	✗	✗
	Worldremit [33]	✗	✗	✓	✓	✓	PIN	CVV	✗
E-commerce	Amazon [34]	✗	✗	✗	✗	✗	N/A	CVV	✗
	E-bay [35]	✗	✗	✗	✗	✗	N/A	CVV	✗
Wallet	Trust Wallet [36]	✗	✗	✓	✗	✓	Custom PIN	Other [2]	✗
	Coinbase [37]	✗	✗	✗	✗	✗	N/A	Other [2]	✗
	MetaMask [38]	✗	✗	✓	✗	✓	✗	Other [2]	✗

✓- Feature is present, ✗- Feature not present.

[2] These apps use third-party services for their transaction and payment. They have different payment and verification methods.

TABLE VII: Device resource consumption of the malicious program

Metric	Consumption (idle)	Consumption (peak)
CPU Percentage	0.01%	0.01%
Memory Percentage	0.01%	0.01%
Virtual Image	4.4 M	9.1 M
Resident Size	1.1 MB	1.4 MB
Shared Memory	880 Kb	1 MB

of malware. Biometric authentication may be a more secure alternative in such a scenario. However, if passwords or PINs are used as a fallback option, they can be exploited by *BADAuth* to circumvent biometric authentication security. One of the applications examined in this study (worldremit) employs a custom PIN (which may differ from the device PIN) as an alternative option, which may increase the difficulty for the attacker, but not eliminate the threat entirely.

Some of the financial and wallet applications do not mandate authentication every time (as shown in Table VI), which exposes them to the risk of severe financial losses and privacy breaches by *BADAuth*.

In comparison to financial applications, the majority of the wallet applications necessitate user authentication. However, they predominantly employ biometric authentication rather than username/password, which offers more security in light of *BADAuth*'s capabilities. Furthermore, their biometric fallback option is more secure, as Trust wallet demands a custom PIN to override the biometric authentication. Metamask does not have any fallback option whatsoever, which can effectively thwart the *BADAuth* attack.

Payment / Fund Transfer: In addition to user authentication for accessing the application, financial applications also implement an extra level of verification for payment or fund transfer, which entails OTP or captcha. However, according to our threat model and analysis, all variants of OTP are vulnerable to *BADAuth*. Here, Advanced captcha, such as image recognition, puzzles, or checkboxes, can help to prevent this attack effectively.

Our observations on payment and fund transfer on examined apps security in the presence of *BADAuth* are listed in Table VI. The payment that involves a credit card requires Card Verification Value (CVV), which is a standard practice. This analysis reveals that very few of these applications, such as PayPal and Metamask, employ secure methods of authentication and payment, such as a captcha and a biometric with no downgrade method, that can prevent an automated attack agent, such as *BADAuth*. Applications that use custom PIN, such as worldRemit and Trust wallet, can pose an extra challenge for the attacker. Other applications, which did not employ these secure methods, are susceptible to *BADAuth*.

VI. Related Work

Mohamed et al. [1] demonstrated ADB vulnerabilities in their work "SMASheD", which exploited ADB permissions to manipulate sensor data. They presented attacks on continuous authentication and device locks, such as pattern and PIN. However, Android 10.0 revoked the ADB shell permission to inject sensor events, making these attack scenarios ineffective on recent Android versions. In contrast, the *BADAuth* attack uses ADB utilities to design a different approach that can compromise modern 2FA and passwordless systems, potentially stealing private data, even on Android 14.0.

Lin et al. [5] discussed ADB vulnerabilities and demonstrated the collection of private information, such as contact lists, through screenshots. Another work, ScreenStealer [4], focused on using ADB utilities to steal private information, similar to [5]. Lin et al. also highlighted the malicious

use of the "screencap" utility and conducted an analysis of Google Play apps to identify potential malicious use of ADB vulnerabilities. Another work [6] analyzed screenshot apps, similar to [5]. In contrast, our work focuses on highlighting the risks associated with 2FA systems involving smartphones and analyzing some popular sensitive apps security in presence of *BADAuth*.

Hwang et al. [3] explored ADB vulnerabilities and the leakage of private information, with a focus on stealing messages, call information, and SIM information. In contrast, our work focuses on compromising secure 2FA systems. To steal OTPs during the attack, we utilized the tap-to-copy feature and the "dumpsys" utility, which is more effective for attacks on 2FA systems compared to the method presented in this work.

Gomez et al. [2] leveraged ADB vulnerabilities in their work "RERAN" to demonstrate record-and-replay in benign application automation processes. However, similar to the approach used in [1], this method is no longer effective in the latest versions of Android. In contrast, the *BADAuth* uses ADB utilities to inject touch and button press events instead of recording sensor data, which is effective even in the latest Android version.

Shrestha et al. [48] proposed a preventive measure against record-and-replay attacks by introducing noisy sensor data. This makes it challenging for attackers to use recorded user input to initiate an attack, similar to the approaches used in "SMASheD" [1] and "RERAN" [2]. However, as our approach does not rely on record-and-replay sensor data, this solution is not effective in defending against our attack.

VII. Discussion and Future Work

Mitigation Strategies: An effective mitigation strategy, for browser-based password managers, given the assumed threat model of credential leakage, involves implementing a secure screen lock, such as a fingerprint, which an automated agent cannot easily bypass. In cases where fingerprint or face ID fails due to device issues, these browsers could employ advanced CAPTCHA, which includes challenges, such as image recognition tasks, puzzles, or checkboxes, that are difficult for an automated agent to breach. This practice, common in financial apps, can potentially enhance the protection of sensitive password vaults by preventing malware attacks, even if credentials are exposed. While this might affect usability slightly, it is essential for the security of such scenarios.

In the case of 2FA and passwordless authentication methods, ensuring user presence is an important security step. This can be achieved through biometric verification, such as fingerprint and face ID, without downgrading option, which can effectively prevent automated touch event and button press generator attacks, including *BADAuth*. Additionally, secure chip implementations, such as *titan-m*, provide another secure means to establish the user presence.

Moreover, it is strongly advised that developers of push notification authentication applications mandate users to tap on the notification and open the application to complete approval process. This action would help to display the unique identifier and buttons within the application interface. This approach provides an opportunity to implement the `FLAG_SECURE` flag in their application, thereby preventing unauthorized screenshots and subsequently thwarting runtime button position analysis conducted by *BADAuth*.

In addition, it is recommended to deactivate the automatic copy-to-clipboard feature when an SMS containing a One-Time Password (OTP) is received. While this feature may enhance usability, it compromises security by potentially facilitating *BADAuth* to effortlessly capture the OTP. Therefore, despite the slight inconvenience it may cause, disabling this feature is a necessary trade-off for enhanced security.

It is recommended that users exercise caution when utilizing public USB charging ports in communal spaces, and instead, consider safer alternatives such as personal wall adapters or power banks. Furthermore, it would be beneficial for Android to incorporate built-in anti-malware programs into its system. These programs would actively monitor all processes and automatically terminate any process exhibiting a definitive signature (e.g., using ADB commands) or abnormal behavior (e.g., generating an excessive number of touch events, running apps, taking screenshots). This proactive approach would enhance the security of Android devices, providing users with a safer and more secure user experience.

Limitation: The process that executes *BADAuth* cannot restart itself once terminated by a user or a restart event. Consequently, the attacker has to complete the attack before the next restart. However, this limitation does not significantly restrict attackers, as most users do not frequently restart their smartphones, providing the *BADAuth* ample time to accomplish its task before the device is restarted. A recent user survey conducted by an online forum [49] indicates that 83% of users do not restart their phones daily, underscoring this tendency.

Moreover, the attack can be mitigated by establishing user presence through biometric verification methods such as fingerprint and face ID. Additionally, the attack cannot bypass the security provided by the titan-m chip in Google Pixel phones. However, we have demonstrated that the attack can be successful in real-life authentication systems using a downgrading attack (e.g., using a PIN instead of biometric) to bypass the security provided by both biometric verification and the titan-m chip.

Future Directions: Researchers can investigate potential loopholes in Android permissions that adversaries could exploit. Additionally, they could focus on designing a secure password manager migration process, thereby eliminating the need for the present export-import approach of password migration in plaintext. Furthermore, the development of anti-malware programs capable of effectively detecting and terminating threats posed by similar programs to *BADAuth* is another promising area of research.

Vulnerability Disclosure: We have reported the vulnerability through the Google Bug Hunters program (Issue ID: 352938791).

VIII. Conclusion

We examined the well-known ADB vulnerability and its potential to breach browser-based password managers, modern 2FA, and passwordless authentication systems. We also showed that it can capture hidden screenshots and screen videos to steal credentials and other sensitive data. This flaw threatens sensitive user accounts, such as banking apps and password managers. Our findings highlight the need for vigilance and appropriate safeguards against such attacks.

Acknowledgments

This work is funded in part by NSF grants: OAC-2139358, CNS-2201465 and CNS-2154507.

References

[1] M. Mohamed, B. Shrestha, and N. Saxena, "Smashed: Sniffing and manipulating android sensor data for offensive purposes," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 4, pp. 901–913, 2016.

[2] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 72–81.

[3] S. Hwang, S. Lee, Y. Kim, and S. Ryu, "Bittersweet adb: Attacks and defenses," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, pp. 579–584.

[4] S. M. Muzammal and M. A. Shah, "Screenstealer: addressing screenshot attacks on android devices," in *2016 22nd International Conference on Automation and Computing (ICAC)*. IEEE, 2016, pp. 336–341.

[5] C.-C. Lin, H. Li, X.-y. Zhou, and X. Wang, "Screenmilker: How to milk your android screen for secrets." in *NDSS*, 2014.

[6] M. H. Meng, G. Bai, J. K. Liu, X. Luo, and Y. Wang, "Analyzing use of high privileges on android: an empirical case study of screenshot and screen recording applications," in *International Conference on Information Security and Cryptology*. Springer, 2018, pp. 349–369.

[7] Android Developers, "Behaviour changes: All apps," April 2022, https://developer.android.com/about/versions/pie/android-9.0-changes-all.

[8] FIDO Alliance , "Fido2: Webauthn & ctap," 2022, https://fidoalliance.org/fido2/.

[9] Google . (2022) Sign in faster with 2-step verification phone prompt. https://bit.ly/2X3DyKL.

[10] Logmeln Inc. (2012) Lastpass - password manager & vault app. https://www.lastpass.com/.

[11] Duo. (2012) Duo two factor authentication and endpoint security. https://duo.com.

[12] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012, pp. 113–124.

[13] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang, "Accessory: password inference using accelerometers on smartphones," in *proceedings of the twelfth workshop on mobile computing systems & applications*, 2012, pp. 1–6.

[14] r/FlutterDev, "Do you use your own phone to test?" October 2022, https://www.reddit.com/r/FlutterDev/comments/ymaidj/do_you_use_your_to/.

[15] Praveena Monohar, "Emulator vs simulator vs real device testing: Key differences," March 2022, https://www.lambdatest.com/blog/emulator-vs-simulator-vs-real-device/.

[16] Sourojit Das, "What is android testing: Types, tools, and best practices," April 2023, https://www.browserstack.com/guide/what-is-android-testing.

[17] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the eighth symposium on usable privacy and security*, 2012, pp. 1–14.

[18] Cybernews – Latest Cybersecurity and Tech News, "Rockyou2021: largest password compilation of all time leaked online with 8.4 billion entries," July 2022, https://cybernews.com/security/rockyou2021-alltime-largest-password-compilation-leaked/.

[19] ——, "Comb: largest breach of all time leaked online with 3.2 billion records," July 2022, https://cybernews.com/news/largest-compilation-of-emails-and-passwords-leaked-free/.

[20] ForgeRock, "2022 forgerock consumer identity breach report," August 2022, https://www.forgerock.com/resources/2022-consumer-identity-breach-report.

[21] C. Jacomme and S. Kremer, "An extensive formal analysis of multi-factor authentication protocols," *ACM Transactions on Privacy and Security (TOPS)*, vol. 24, no. 2, pp. 1–34, 2021.

[22] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, C. Raules, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, M. Grace *et al.*, "Attention spanned: Comprehensive vulnerability analysis of {AT} commands within the android ecosystem," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 273–290.

[23] Mobile Hacker, "Nethunter hacker vi: Ultimate guide to hid attacks using rubber ducky scripts and bad usb mitm attack," August 2023, https://www.mobile-hacker.com/2023/08/08/nethunter-hacker-vi-ultimate-guide-to-hid-attacks-using-rubber-ducky-scripts-and-bad-usb-mitm-attack/.

[24] Federal Communications Commission, "'juice jacking': The dangers of public usb charging stations," October 2021, https://www.fcc.gov/juice-jacking-dangers-public-usb-charging-stations.

[25] Bruce Schneier, "Fbi advising people to avoid public charging stations," April 2023, https://www.schneier.com/blog/archives/2023/04/fbi-advising-people-to-avoid-public-charging-stations.html?

[26] B. Watson and J. Zheng, "On the user awareness of mobile security recommendations," in *Proceedings of the SouthEast Conference*, 2017, pp. 120–127.

[27] Jerry Hildenbrand, "What is the titan security module," January 2020, https://www.androidcentral.com/what-titan-security-module.

[28] Z. Lei, Y. Nan, Y. Fratantonio, and A. Bianchi, "On the insecurity of sms one-time password messages against local attackers in modern mobile devices," in *Proceedings of the 2021 Network and Distributed System Security (NDSS) Symposium*, 2021.

[29] Wells Fargo, "The wells fargo mobile app," November 2023, https://www.wellsfargo.com/mobile-online-banking/apps/.

[30] Paypal Pte. Ltd., "Paypal app: Send and manage your money," November 2023, https://www.paypal.com/ao/webapps/mpp/mobile-apps.

[31] Discover Financial Services, "Disvcover mobile," November 2023, https://play.google.com/store/apps/details?id=com.discoverfinancial.mobile.

[32] Remitly Inc., "Remitly: Send and recieve money," November 2023, https://www.remitly.com.

[33] WorldRemit Corp., "Money transfer app: Worldremit," November 2023, https://www.worldremit.com.

[34] Amazon.com, Inc. Amazon.com: Online shopping for electronics, apparels, computer, books & dvd and more. [Online]. Available: https://www.amazon.com

[35] eBay Inc., "Life is easier in the ebay app," November 2023, https://pages.ebay.com/mobile-app/.

[36] trustwallet.com, "Best cryptowallet app for web2,nft, and defi," November 2023, https://trustwallet.com.

[37] Coinbase, "Download coinbase wallet," November 2023, https://www.coinbase.com/wallet/downloads.

[38] Metamask- A consensys formation, "Download metamask: Blockchain mobile app," November 2023, https://metamask.io/download/.

[39] Avast Software s.r.o. (2022) Free android antivirus app – avast mobile security. https://www.avast.com/en-us/free-mobile-security.

[40] Kaspersky Lab. (2022) Kaspersky internet security for android. https://usa.kaspersky.com/android-security-free.

[41] Lookout, Inc. (2022) Lookout personal for android. https://www.lookout.com/products/personal/android.

[42] Avira Operations GmbH & Co. KG. (2022) Avira antivirus security for android. https://www.avira.com/en/free-antivirus-android.

[43] BullGuard. (2022) Free bullguard mobile security. https://www.bullguard.com/products/bullguard-free-android-mobile-security.aspx.

[44] Panda. (2022) Antivirus for android- panda security. https://www.pandasecurity.com/en/homeusers/android-antivirus/.

[45] Sophos Ltd. (2022) Sophos intercept x for mobile for android. https://www.sophos.com/en-us/products/free-tools/sophos-mobile-security-free-edition.aspx.

[46] Avast Software s.r.o. (2022) Avg antivirus for android. https://www.avg.com/en-us/antivirus-for-android.

[47] Bitdefender. (2022) Bitdefender antivirus free for android. https://www.bitdefender.com/solutions/antivirus-free-for-android.html.
[48] P. Shrestha, M. Mohamed, and N. Saxena, "Slogger: Smashing motion-based touchstroke logging with transparent system noise," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2016, pp. 67–77.
[49] OnePlus Community, "How often do you believe it's acceptable to reboot your phone?" November 2022, https://community.oneplus.com/thread/1203882664923758594.