A11y Attacks: Exploiting Accessibility in Operating Systems

Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee
School of Computer Science, College of Computing
Georgia Institute of Technology, Atlanta, GA, USA
{yeongjin.jang,csong84,pchung,tielei.wang,wenke}@cc.gatech.edu

ABSTRACT

Driven in part by federal law, accessibility (a11y) support for disabled users is becoming ubiquitous in commodity OSs. Some assistive technologies such as natural language user interfaces in mobile devices are welcomed by the general user population. Unfortunately, adding new features in modern, complex OSs usually introduces new security vulnerabilities. Accessibility support is no exception. Assistive technologies can be defined as computing subsystems that either transform user input into interaction requests for other applications and the underlying OS, or transform application and OS output for display on alternative devices. Inadequate security checks on these new I/O paths make it possible to launch attacks from accessibility interfaces. In this paper, we present the first security evaluation of accessibility support for four of the most popular computing platforms: Microsoft Windows, Ubuntu Linux, iOS, and Android. We identify *twelve* attacks that can bypass state-of-the-art defense mechanisms deployed on these OSs, including UAC, the *Yama* security module, the iOS sandbox, and the Android sandbox. Further analysis of the identified vulnerabilities shows that their root cause is that the design and implementation of accessibility support involves inevitable trade-offs among compatibility, usability, security, and (economic) cost. These trade-offs make it difficult to secure a system against misuse of accessibility support. Based on our findings, we propose a number of recommendations to either make the implementation of all necessary security checks easier and more intuitive, or to alleviate the impact of missing/incorrect checks. We also point out open problems and challenges in automatically analyzing accessibility support and identifying security vulnerabilities.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Accessibility; Assistive Technology; Attacks

1 Introduction

On August 9, 1998, the United States Congress amended the Rehabilitation Act of 1973 to eliminate barriers in electronic and information technology for people with disabilities [32]. Effective June 21, 2001, the law is enforced on the development, procurement, maintenance, or use of electronic and information technology by the federal government [21]. Driven by this requirement, OS vendors [2, 15, 24] have included accessibility features such as on-screen keyboards, screen magnifiers, voice commands, screen readers, etc. in their products to comply with federal law.

Assistive technologies, especially natural language voice processors, are gaining widespread market acceptance. Since the iPhone 4S, Apple has included in iOS a voice-based personal assistant, Siri, which can help the user complete a variety of tasks, such as placing a call, sending a text, and modifying personal calendars. Google also added a similar feature, Voice Action, to its Android platform. Furthermore, wearable devices such as Google Glass use voice as the main interaction interface.

In general, adding new features into modern complex OSs usually introduces new security vulnerabilities. Accessibility support is no exception. For example, in 2007, it was reported that Windows Vista could be compromised through its speech recognition software [28]; in 2013, a flaw was discovered in Siri that allowed the bypass of an iPhone's lock screen to access photos and email [11]. As accessibility features are being used by more and more people, security issues caused by such vulnerabilities can become more serious.

In this paper, we present the first security evaluation of the accessibility support of commodity OSs. Our hypothesis is that alternative I/O subsystems such as assistive technologies bring a common challenge to many widely deployed security mechanisms in modern OSs. Modern OSs support restricted execution environments (e.g., sandboxes) and ask for the user's approval before applying a security sensitive change to the system (e.g., User Access Control (UAC) on Windows [25] and remote view on iOS [4]). However, accessibility support usually offers interfaces to programmatically generate user inputs, such as keystrokes and mouse clicks, which essentially enables the interface to act as a human being. Consequently, it might be possible to bypass these defense mechanisms and abuse a user's permissions by generating synthesized user inputs. Similarly, attackers may also be able to steal security sensitive information displayed on screen through the accessibility interfaces.

To verify our hypothesis, we examined the security of accessibility on four commodity OSs: Microsoft Windows 8.1, Ubuntu 13.10, iOS 6, and Android 4.4. We were able to identify *twelve*[1] attacks[2] that can bypass many state-of-the-art defense mechanisms deployed on these OSs, including UAC, the Yama security module, the iOS App sandbox, and the Android sandbox.

When designing new interfaces that provide access to computing systems, one must ensure that these new features do not break existing security mechanisms. However, current designs and implementations of accessibility support have failed to meet this requirement. Our analysis shows that current architectures for providing accessibility features make it extremely difficult to balance compatibility, usability, security, and (economic) cost. In particular, we found that security has received less consideration compared to the other factors. Under current architectures, there is not a single OS component that has all the information necessary to enforce meaningful security policy; instead, the security of accessibility features depends on security checks implemented in the assistive technology, the OS, and the applications. Unfortunately, in our evaluation, we found that security checks are either entirely missed or implemented incorrectly (or incompletely) at all levels. Based on our findings, we believe a fundamental solution to the problem will involve a new architecture that is designed with security in mind. Proposing this new architecture is beyond the scope of our work. Instead, we propose several recommendations that work under current architectures to either make the implementation of all necessary security checks easier and more intuitive, or to alleviate the impact of missing/incorrect checks. We also point out some open problems and challenges in automatically analyzing a11y support and identifying security vulnerabilities.

In summary, this paper makes the following contributions:

- We performed a security evaluation of accessibility support for four major OSs: Windows, Ubuntu Linux, iOS, and Android;

- We found several new vulnerabilities that can be exploited to bypass many state-of-the-art defense mechanisms deployed on these systems, including UAC and application sandboxes;

- We analyzed the root cause of these vulnerabilities and proposed a number of recommendations to improve the security of a11y support;

- We showed that the current architectures for providing accessibility features are inherently flawed, because no single OS component is able to implement a security policy: *security checks at the assistive technology, the OS, and the application must be implemented correctly; failure in any of these checks introduces vulnerabilities*.

The rest of this paper is organized as follows. Section §2 gives a brief background of accessibility support. Section §3 discusses security implications of accessibility. Section §4 presents the evaluation results, i.e., new vulnerabilities and attacks. Section §5 discusses the limitations of our attacks, the root cause of the vulnerabilities, and complexity of the problem. Section §6 compares this work with related works. Section §7 concludes the paper.

2 Overview of Accessibility

In this section, we give a brief overview of accessibility in operating systems, and present definitions of terminologies used in this paper.

[1]We discovered eleven new attacks, and we cover an attack for Siri that was released in public as exploitation of accessibility in OS.
[2]**Disclosure:** we reported all vulnerabilities that we found to the OS vendors.

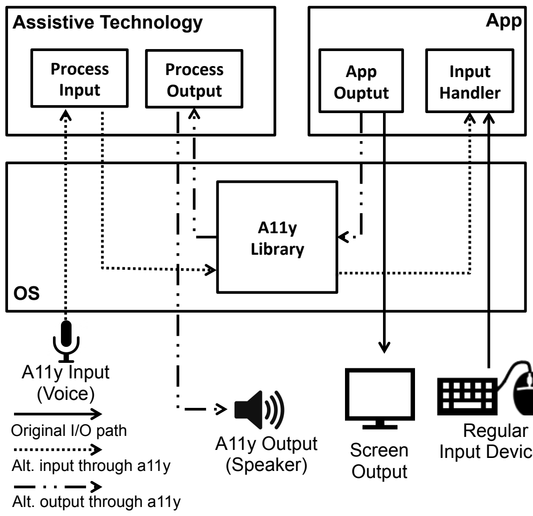


Figure 1: General architecture for implementing accessibility features. Supporting an accessibility feature creates new paths for I/O on the system (two dotted lines), while original I/O from/to hardware devices (e.g., keyboard/mouse and screen) is indicated on the right side.

2.1 Accessibility Features

In compliance with the amended Rehabilitation Act, software vendors have incorporated various accessibility features into their systems. In this paper, we define *computer accessibility (a11y) features* as new I/O subsystems that provide alternative ways for users with disabilities to interact with the system. For example, for visually impaired users, text-to-speech based Narrator (on MS Windows), VoiceOver (on OS X), and TalkBack (on Android) provide an output subsystem that communicates with the user through speech. For hearing impaired users, accessibility features like captioning services turn the system's audio output into visual output. Similarly, some systems can alert the user about the presence of audio output by flashing the screen. For users with motor disabilities, traditional mouse/keyboard based input systems are replaced by systems based on voice input. In general, we can see these accessibility features as implemented within an OS architecture in Figure 1.

There are also accessibility features that use traditional I/O devices (e.g., the screen, mouse and keyboard), but make them easier for users with disabilities to interact with the system. Examples of such features include: 1) Magnifier in Windows, which enlarges certain portions of the screen; 2) High Contrast in Windows, which provides higher contrast for easy distinction of user interfaces; and 3) on-screen keyboard, sticky keys, filter keys, assisted pointing, and mouse double-click speed adjust to allow input requiring less movement.

2.2 Accessibility Libraries

In addition to pre-installed accessibility features, most OS vendors provide libraries for third parties to implement their own accessibility features. This makes it possible to create new alternative I/O subsystems based on other I/O devices (e.g. a braille terminal). In this case, the assistive technology part in Figure 1 is a program developed by third party. Examples of these libraries include 1) UI Automation in Microsoft Windows, 2) the accessibility toolkit (ATK) and Assistive Technology Service Provider Interface (AT-SPI) in Ubuntu Linux, 3) AccessibilityService and related classes in Android, and 4) the (public) NSAccessibility and (private) UIAutomation frameworks in iOS.

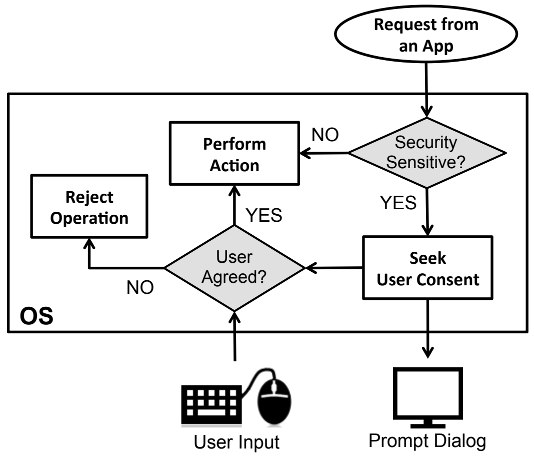


Figure 2: Traditional mechanism to seek user consent before performing privileged operations.

For all the discussions that follow, we will refer to these libraries as *accessibility libraries*. In general, the accessibility libraries provide the following capabilities as APIs:

1. Notifications on changes to the system's display (e.g. new window popped up, content of a window changed/scrolled, change of focus, etc.);
2. Ways to probe what is displayed on various UI elements (e.g. name of a button, content of a textbox, or static text displayed);
3. Ways to synthesize inputs to various UI elements (e.g. click a button to place text into a textbox).

2.3 Assistive Technologies

For the rest of this paper, we will use the term *assistive technology* (AT) to refer to the logic that runs in user space to provide any of the following functionality:

- (F1) processing user input from alternative input devices, "understanding" what the user wants and turning it into commands to the OS for control of other applications (or the OS itself);
- (F2) receiving information about the system's output and presenting it to users using alternative output devices.

Usually an assistive technology makes use of an accessibility library to obtain required capabilities for implementing a new accessibility feature.

3 Security Implications of A11y

In this section, we discuss new attack paths due to accessibility features in computing systems and correspondingly the required security checks for securing accessibility support. For the rest of this paper, we adopt the threat model where the attacker controls one user space process with access to the accessibility library, and we do not assume any other special privilege for this malicious process.

3.1 New Attack Paths

The first functionality (F1) of AT allows users to control the system through alternative input devices, which is inherently dangerous from a security perspective. While modern OSs provide increasingly restricted isolation between applications, accessibility support provides a way to bypass this isolation and control other applications on the system.

To prevent malware from abusing security sensitive privileges of the user, OSs also deploy defense mechanisms such as User

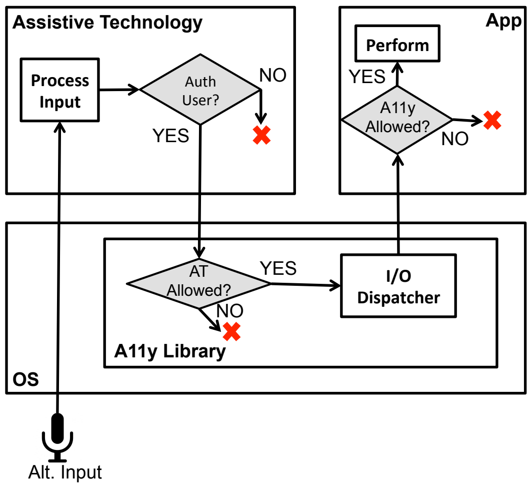


Figure 3: Required security checks for an AT as a new input subsystem. User input is passed to the AT first, moved to OS through accessibility libraries, then synthetic input is delivered to the application. Grayed boxes indicate security checks required by each entity that receives the input.

Account Control (UAC) [25] in Windows, remote view [4] in iOS, and ACG [30], with the policy of "ask for user consent explicitly before performing dangerous operations" (see Figure 2). However, since user consent is usually represented by a certain input event (e.g., click on a button), the capability to programmatically generate input events also breaks the underlying assumption of these security mechanisms that input is always the result of user action.

The ability of AT to monitor and probe the information currently being displayed on screen (F2) is also problematic because it provides a way to access certain security sensitive information, e.g., plaintext passwords usually not displayed on screen (e.g., most OSs show only scrambled symbols in the password box).

Based on the above observations, we argue that accessibility interfaces provide malware authors with these new paths of attacks:

- (A1) Malware implemented as AT penetrates the OS security boundary by obtaining new capabilities of controlling applications;
- (A2) Malware exploits the capability of generating interaction requests to bypass defense mechanisms or escalate its privilege;
- (A3) Malware exploits the capability of monitoring and probing the UI output to access otherwise unavailable information.

3.2 Required Security Checks

To evaluate how a platform could be secure against these new attack paths, we propose two reference models of required checks: one for handling alternative input (Figure 3) and the other for handling output (Figure 4).

The key to securely handling alternative input is to validate whether the input is truly from the user. To achieve this goal, we argue that three checks (gray boxes in Figure 3) along the input path are necessary: within the AT, in the OS, and at the application level.

First, an AT should validate whether the input is from the user. Otherwise, attacks can be launched by synthesizing the input format of this AT. For example, malware can transform malicious operations into synthetic voice (e.g., via text-to-speech, TTS) and drive the natural language user interface to control other applications (A1) or escalate its privilege (A2).

Second, since not all ATs can be trusted (e.g., those provided by a third-party), the OS should have control over what applications an

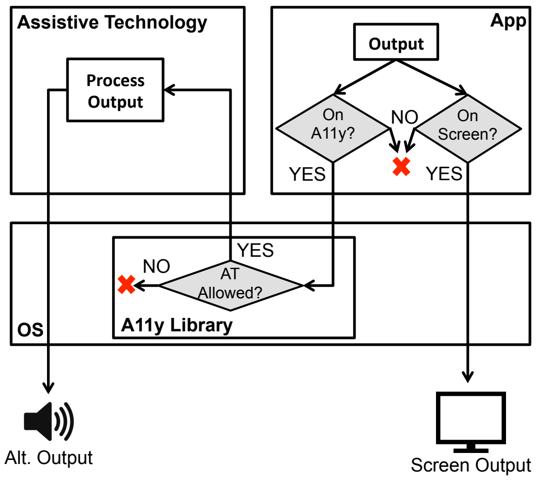


Figure 4: Required security checks for an AT as a new output subsystem. Application is required to decide which input can transit through the accessibility library, then the AT receives the output to deliver it to the user. Grayed boxes indicate the checks required by OS and the application.

AT can control. For example, interaction requests from untrusted AT to security sensitive processes such as system services and system settings should not be forwarded. Otherwise, privilege escalation would be feasible (A2). In addition, the access control policy should be consistent with other access control mechanisms to prevent a malicious AT from obtaining new capabilities (A1).

Third, the OS should provide the flexibility to allow an application to specify a fine-grained security policy on how to handle interaction requests from an AT. More specifically, the OS should 1) allow the application to distinguish input from real hardware and input from AT; and 2) allow the application to set its own callback functions to handle input events from AT. More importantly, when no customization is provided, the default setting should align with the platform's default security policy.

These three checks are complementary to each other for the following reasons. First, for AT-like natural language user interfaces for motor disabled people, it has to be able to control all applications and the underlying system; the only viable check is within the AT itself. Second, as not all ATs are trustworthy, the OS-level check is necessary to prevent malicious AT from compromising the system. Third, OS-level access controls are not aware of the context of each non-system application, so the application level check provides the last line of defense for an application to protect itself from malicious ATs (A1).

Similarly, to securely handle alternative output and prevent information leakage (A3), two checks (gray boxes in Figure 4) should be performed. The application level check allows the application to specify what information is sensitive so it will not be available to AT. Again, we must emphasize that when no customization is provided, the default setting should align with the platform's security policy. The OS-level check prevents untrusted ATs from acquiring sensitive information specific to the system.

4 Security Evaluation of A11y

In this section, we first describe our evaluation methodology, and then present the results of the security evaluation on major platforms: Microsoft Windows, Ubuntu Linux, iOS, and Android. The specific versions of the evaluated systems are: Windows 8.1, Ubuntu 13.10, iOS 6 and Android 4.4 on the Moto X [3].

[3]For Windows, Ubuntu, and Android, we tested the latest release version as of November 2013. Attacks still work for the current release versions. For iOS, we tested iOS 6.1.4, the latest iOS 6 at the time the research was performed.

4.1 Evaluation Methodology

Given an OS platform, we evaluate the security of the accessibility features it offers as follows:

1. We studied the availability of the built-in assistive technologies and the accessibility library on the platform. For built-in assistive technologies, we focused on the availability of a natural language user interface because it provides the most powerful control over the system. For the accessibility library, we focused on whether an application needs special privileges to use the library; if so, how such privileges are granted.

2. Using our input validation model (Figure 3), we examined the input handling process of the analyzed platform. When a check is missing or flawed, we try to launch attacks exploiting the missing or flawed check. Specifically, if the built-in natural language user interface lacks input validation or if the validation can be bypassed, we try to escalate our malware's privilege through synthetic voice. If the OS-level check is missing and there is a security mechanism that requires user consent, we try to escalate our malware's privilege by spoofing the mechanism with synthetic input. If the OS-level check is *not* missing, we assess whether its access control policy is consistent with other security mechanisms; if not, we evaluate what new capabilities become available. If the application level check is missing or flawed, we examine whether accessibility support provides us new capabilities.

3. Using our output validation model (Figure 4), we examined the output handling process of the analyzed platform. If the OS-level check is missing, we try to read the UI structure of other applications. If the application level check is missing, we examine whether new capabilities become available. In particular, since most of displayed information is available through screenshots, we try to steal a password because it is usually not displayed in plaintext. We assume obtaining any other (potentially sensitive) information as plaintext via AT is no harder than reading a password.

4.2 Availability of Accessibility Features

Table 1 summarizes the availability of a natural language user interface and accessibility libraries on the four platforms. Natural language user interfaces are available on all platforms except Ubuntu; accessibility libraries are available on all studied platforms [4].

Platform	Natural Language User Interface	Accessibility Libraries
Windows	Speech Recognition*	UIAutomation
Ubuntu	None	ATK, AT-SPI
iOS	Siri	UIAutomation*
Android	Touchless Control*	AccessibilityService*

Table 1: Accessibility libraries and natural language user interface on each platform. * indicates the feature requires special setup/privilege.

For natural language user interfaces, both Speech Recognition and Touchless Control for the Moto X[5] require initialization (training)

[4] On iOS, there is no accessibility library, but the UIAutomation framework provides most capabilities that we require.

[5] For the natural language user interface on Android, we try to analyze Touchless Control which is only available on the Moto X, due to the lack of a privileged natural language user interface in Android by default.

before first use. Siri can be enabled without any setup. Although Speech Recognition on Windows requires initialization, this step can be bypassed by modifying the values of a registry sub-key at `HKEY_CURRENT_USER\Software\Microsoft\Speech`. Since this key is under `HKEY_CURRENT_USER`, it is writable by any unprivileged process.

For accessibility libraries, both desktop environments (Windows and Ubuntu Linux) have no privilege requirements for using the libraries, thus they are available to any application.

On iOS, the UIAutomation framework, though not a full-fledged accessibility library, provides the functionality to send synthesized touch and button events. Since this framework is part of the private API set, its usage is forbidden by apps in the Apple App Store. However, as demonstrated in [33], the enforcement can be bypassed.

Unlike other platforms, Android's accessibility library (AccessibilityService) is available only after the following requirements are met: first, the app must declare use of the permission `BIND_ACCESSIBILITY_SERVICE`. Second, the user must explicitly enable the app as an accessibility service. When changing accessibility settings, a user is prompted with a dialog that displays what kind of capabilities will be granted to the AccessibilityService, which is very similar to the app permission system. Nonetheless, users are prone to enable permissions when apps provide step-by-step instructions. In particular, we find that there are more than 50 apps on the Google Play store that declare use of permissions for AccessibilityService, and two of them [14, 29] have been downloaded by more than ten million users combined. Both are rated 4.3 out of 5 stars.

4.3 Vulnerabilities in Input Validation

Platform	Assistive Tech. Check	OS Level Check	Application Level Check
Windows	None	UIPI*	None
Ubuntu	N/A	None	None
iOS 6	None	None	None
Android	Authentication	Permission*	None

Table 2: Status of input validation on each platform. * indicates the check enforces a security policy that is different from other security mechanisms.

Table 2 summarizes the examination results of each platform when checked against our input reference model (Figure 3). There are two common problems across all analyzed platforms.

Missing or flawed input validation within AT. Natural language user interfaces usually have more privileges than normal applications; most of them lack authentication for voice input. Moreover, some accept self-played input (sending audio from the built-in speaker to microphone), making it possible to inject audio input through text-to-speech (TTS). Although Touchless Control on the Moto X tries to authenticate its input, the authentication can be easily bypassed with a replay attack. As a result, an attacker can obtain the privileges of the natural language user interface (attack #1, #5, #9).

Control of other applications. At the application level, no platform provides a precise way to check whether the input event is from the hardware or from the accessibility library. Moreover, at the OS level, although Windows and Android have access controls for AT, their protections are not complete. This allows a malicious AT to control most applications the same way as a human user would. Specifically, a malicious AT can send input events to make other applications perform security sensitive actions (attack #4, #6, #7, #10) and spoof security mechanisms that require user consent (attack #2, #3, #8).

Implementation of attacks. We tested all Windows-based attacks by implementing proof-of-concept malware. For controlling apps on Ubuntu Linux, iOS 6, and Android, we checked the capability of sending synthetic input to other applications by writing sample code for sending basic user interactions such as clicking a button, and writing content into a textbox. For iOS, we also wrote code to test for special UI windows such as passcode lock, password dialog, and remote view. For Touchless Control, we implemented sample malware that records sound in the background; we then sliced the authentication phrase from it manually, and replayed the slice within the malware. For Siri, we manually performed the same attack.

4.3.1 Windows

The OS-level check applied to the accessibility library on Windows is called User Interface Privilege Isolation (UIPI) [23]. UIPI is a mandatory access control (or mandatory integrity control (MIC) in Microsoft's terminology) that sets an integrity level (IL) for every process and file, and enforces a relaxed Biba model [5, 22]: no write/send to a higher integrity level. The integrity levels (IL) are divided in 5 categories: *Untrusted*, *Low*, *Medium*, *High*, and *System*.

Regular applications run at Medium IL, while processes executed by an active administrator runs at High IL. As an MIC, the IL of a process is inherited by all of its child processes, and takes the minimum privilege when two or more ILs are applied on the process.

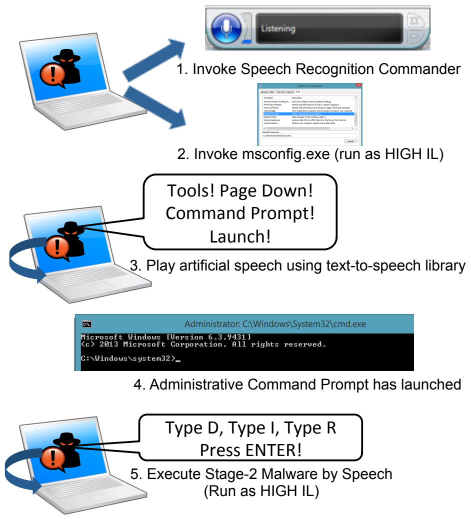
UIPI prevents attackers from sending input to higher IL processes. For example, malware cannot spoof UAC through a synthesized click because normal programs including malware run at either Medium IL (when launched by the user) or Low IL (when launched by browser, i.e., drive-by attacks), while the UAC window runs at System IL. Furthermore, malware cannot take control of applications that are executed by the administrator, which has a higher IL (High IL).

Unfortunately, the protection provided by UIPI is not complete: since most applications are running at the same Medium IL as malware, UIPI allows malware to control most other applications via AT.

Furthermore, the lack of security checks at the assistive technology and application levels results in more vulnerabilities: missing input validation in the built-in natural language user interface allows privilege escalation attacks through Speech Recognition (attack #1); missing application level checks enables escalation of privilege (attack #2), and theft of user passwords (attack #3).

Attack #1: privilege escalation through Speech Recognition. Control of Speech Recognition is security sensitive for several reasons. First, although there is a setup phase, it can be bypassed as mentioned in Section §4.2. After setup, any process can start Speech Recognition. Second, Speech Recognition always runs with administrative privilege (High IL) regardless of which process runs it. This allows it to control almost all other applications on the system, including applications running with administrative privileges. Because of these "features" of Speech Recognition and the problems mentioned previously (i.e., no input validation, and accepting self-played voice), malware running at Medium or even Low IL can escalate itself to administrative privilege through synthetic voice.

Figure 5 shows the workflow of the privilege escalation attack from a Medium IL malware. The first step is to launch Speech Recognition through `CreateProcess()` with the argument `sapisvr.exe -SpeechUX`. Second, the malware launches the `msconfig.exe`

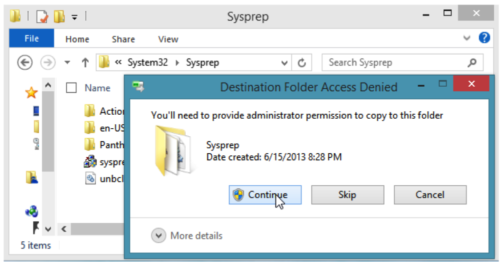

Listening
1. Invoke Speech Recognition Commander
2. Invoke msconfig.exe (run as HIGH IL)
Tools! Page Down! Command Prompt! Launch!
3. Play artificial speech using text-to-speech library
Administrator: C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 6.3.9431]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Windows\system32>_
```

4. Administrative Command Prompt has launched
Type D, Type I, Type R Press ENTER!
5. Execute Stage-2 Malware by Speech (Run as HIGH IL)

Figure 5: Workflow of privilege escalation attack with Windows Speech Recognition.



Destination Folder Access Denied

You'll need to provide administrator permission to copy to this folder

Sysprep
Date created: 6/15/2013 8:28 PM

Figure 6: Dialog that pops-up when Explorer.exe tries to copy a file to a system directory. The dialog runs at the same Medium IL as `Explorer.exe`. Thus, any application with Medium IL can send a synthetic click to the "Continue" button, and proceed with writing the file.

application through `CreateProcess()`. Since `msconfig.exe` is an application for an administrator to manage the system configuration, it automatically runs at High IL. While malware cannot send input events to this process (prevented by UIPI), Speech Recognition can. After launching `msconfig.exe`, the malware can use voice commands to launch a command shell by choosing an item under the tools tab of `msconfig.exe`. This is accomplished by playing a piece of synthetic speech "Tools, Page Down, Command Prompt, Launch!". Once the command shell that inherits the High IL from `msconfig.exe` is launched, the malware then says "cd" to its directory, says its own executable name and "Press Enter" to be executed with administrative privileges.

Attack #2: privilege escalation with `Explorer.exe`. `Explorer.exe` is a special process in Windows that has higher privilege than its running IL. Unlike other Medium IL processes, `Explorer.exe` has the capability of writing to High IL objects such as the `System32` directory. Although this capability is protected by a UAC-like dialog (Figure 6), i.e., `Explorer.exe` asks for the user confirmation before writing to a system directory of Windows, the dialog belongs to `Explorer.exe` itself. Since this action requires user consent, the application should check whether the input comes from the user or AT. However, there is no such check. As a result, malware can overwrite files in system directories by clicking the confirmation dialog through the accessibility library.

Some system applications in system directories are automatically escalated to the administrative privilege at launch. On Windows, when a process tries to load a DLL, the dynamic linker first looks for the DLL from the local directory where the executable resides.

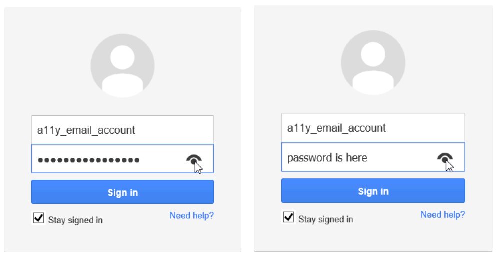


Figure 7: Password Eye on the Gmail web application, accessed with Internet Explorer 10. In Windows 8 and 8.1, this Eye is attached to password fields not only for web applications, but also regular applications. By left-clicking the Eye, the box reveals its plaintext content.

Once malware injects malicious DLLs into the directory containing these applications, it can obtain the administrative privilege when the applications are run, thus bypassing UAC. An example of such an application is `sysprep`, which will load `Cryptbase.DLL` from the local directory. By sending synthetic clicks to `Explorer.exe` and injecting a malicious `Cryptbase.DLL`, malware can achieve privilege escalation.

Attack #3: stealing passwords using Password Eye and a screenshot. On Windows, passwords are protected in several ways. They are not shown on the screen; and even with real user interactions, the content in a password box cannot be copied to the clipboard. Furthermore, as will be described in detail in Section §4.4, it is also not possible to retrieve password content directly through the accessibility library. However, the lack of input validation on the password box UI component opens up a method of stealing the plaintext of a password.

Starting with Windows 8, Microsoft introduced Password Eye as a new UI feature to give visual feedback to users to correct a typo in a password input box (Figure 7). This "Eye" appears when a user provides input to a password box, and clicking it will reveal the plaintext of the password. Unfortunately, since Password Eye cannot distinguish hardware input from synthetic input, malware can click it as long as UIPI permits. Again, since most applications run at the same IL as malware, malware can send a left-click event to reveal the content of the password dialog (Figure 7), and can extract it from a screenshot.

4.3.2 Ubuntu Linux Desktop

Since Ubuntu does not have a built-in natural language user interface, we only consider the attacks enabled by missing checks in the OS or an application. The missing check at the OS level allows malware to control any application and thus break the boundary enforced by other security mechanisms (attack #4). The missing check at the application level does not provide additional capabilities beyond those already provided by the missing OS level check.

Attack #4: bypassing the security boundaries of Ubuntu. Since neither the OS nor applications authenticate input, malware can send synthetic input to any application in the GUI, i.e. the current X Window display. The display here does not mean the physical display (i.e., a monitor screen) of the device; rather, it refers to the logical display space (e.g.,`:0.0`) of the X Window Server.

In this setting, the lack of security checks for input breaks two security boundaries in Ubuntu. The first violation is regarding user ID (UID) boundaries. Regardless of the UID of the display service, a launched process will run with the UID of the user who launched it. For example, if a non-root user runs a GUI application with sudo (e.g., `sudo gparted` or a GUI shell with root privileges), the application runs in the same display space of the non-root user account, even though it runs as the root UID. Since AT-SPI allows control of any application on the display, malware with a non-root UID can send synthetic input to control other applications, even those with root privileges.

Second, process boundaries can be bypassed by sending synthetic input. Starting with Ubuntu 10.10, Ubuntu adopted the *Yama* security module [8] to enhance security at the kernel level. In particular, one feature in *Yama* prohibits a process from attaching to another process using the *ptrace* system call, unless the target process is a descendant of the caller. Thus, a process cannot attach or read/write other processes' memory if the target is not created by itself or its descendants. However, malware can bypass this restriction: it can write values or perform UI actions to change application status through synthetic inputs or interfaces available by AT-SPI such as `settextvalue()` and `invokemenu()`.

4.3.3 iOS

iOS 6 lacks security checks at all levels. Missing input validation in its natural language user interface, Siri, allows an attacker to abuse its privileges to perform sensitive operations and access sensitive information (attack #5). Furthermore, missing OS-level checks allows malware to 1) bypass sandbox restrictions to control other apps (attack #6), 2) spoof the remote view mechanism to programmatically authorize access permissions to sensitive resources (attack #7), and 3) bypass password protection (attack #8). Finally, since there are no available checks at the application level, synthetic input from a malicious app cannot be prevented or detected by the targeted application.

Attack #5: bypassing passcode lock using Siri.[6] iOS allows several security-sensitive actions to be carried out through Siri even when the device is locked with a passcode. Such actions include making phone calls, sending SMS messages, sending emails, posting messages on Twitter and Facebook, checking and modifying the calendar, and checking contacts by name. Since there is no input validation, any attacker who has physical access to the iOS device can launch the attack without any knowledge of the passcode.

Attack #6: bypassing the iOS sandbox. App sandboxing [3] in iOS enforces a strict security policy that strongly isolates an app from others. The data and execution state of an app is protected so that other apps cannot read or write its memory, or control its execution (e.g., launching the app). However, the lack of OS-level security checks on accessibility makes it possible for malware to control other apps by sending synthesized input. With synthetic touch, malware can perform any actions available to a user, such as launching other apps, typing keystrokes, etc. That is, malware can *steal* capabilities of other apps across the app sandbox.

Attack #7: privilege escalation with remote view. In addition to app sandboxing, iOS protects its security sensitive operations with the remote view mechanism [4]. Protected operations include sending email, posting on Twitter or Facebook, and sending SMS. Remote view works as follows: when an app tries to access any protected operation, the underlying service (which is a different process) pops up a UI window to seek user consent. For example, if an app wants to send an email, it invokes a remote function call to the email service, which would then pop up a confirmation window.

[6] We note that this attack on Siri was not originally discovered by us. The attack has been publicly known since September 2013 [11], but we include this in the paper due to the importance of its security implications on built-in AT.

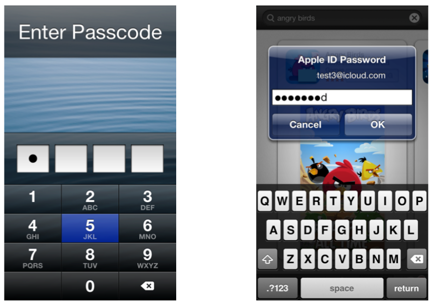


Figure 8: Screenshot of passcode and password input in iOS. For passcode (left), typed numbers can be identified by *color differences* on the keypad. For the password (right), iOS always shows the last character to give visual feedback to the user.

The email message can only be sent after the user clicks the "Send" button in the pop-up window.

Remote view is considered an effective defense mechanism to prevent misuse of sensitive operations. However, the lack of input validation in iOS allows malware to send synthetic touches to spoof user input to remote view and execute these privileged operations.

Attack #8: bypassing password protection on iOS. Another protection mechanism in iOS is passwords. This is utilized in two system apps: the lock screen and the App Store.

The lock screen prevents any unauthorized access to the device and is applied not only to UI events, but also to security data such as KeyChain and encrypted files. Moreover, once the screen is locked, all touch events are blocked; thus malware is no longer able to manipulate apps other than the lock screen.

The App Store asks for an Apple ID and password for each purchase. Although malware can generate "clicks" to initiate the purchase, without knowing the password, it is not possible to finish the transaction.

Unfortunately, since iOS always displays the last character of a passcode/password in plaintext (Figure 8) and background screenshots can be taken through the private API call `createScreenIOSurface` in `UIWindow` class, it is possible to steal the user's passcode and password. With a stolen password, since both the lock screen and the App Store accept synthesized input, malware can programmatically unlock the device and make malicious transactions.

4.3.4 Android

The Android platform has the most complete input validation among the four evaluated platforms. First, *Touchless Control* [27], a natural language user interface for the Moto X, utilizes voice authentication: the user is required to register his/her voice with Touchless Control at first boot; the app then constantly monitors microphone input for the fixed authentication phrase "OK Google Now" from the user. Once the command is captured, it checks whether the phrase matches the voice signature extracted from the registered phrase; if so, it then launches the Google Now application to execute a voice command. Nonetheless, like other non-challenge-response-based authentication, this voice authentication is vulnerable to *replay attacks* (#9).

Second, as discussed in Section §4.2, Android requires explicit user consent to acquire accessibility capabilities. However, its protection is incomplete. Specifically, Android has no runtime security check for AT. Once an app is allowed to be an AT, it can leverage the accessibility library to create a new inter-process communication (IPC) channel that is not protected by the ordinary Android permission system (#10). As a result, a malicious AT can easily achieve the same effect as capability leakage attacks [7, 9, 13, 16, 34] and information leakage attacks [18, 36]. Moreover, unlike UIPI, Android's

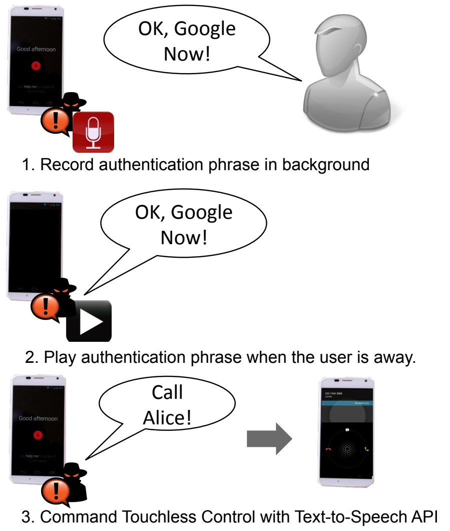

OK, Google Now!
1. Record authentication phrase in background
OK, Google Now!
2. Play authentication phrase when the user is away.
Call Alice!
3. Command Touchless Control with Text-to-Speech API

Figure 9: Workflow of the attack on the Moto X's Touchless Control. Malware in the background can record a user's voice, and replay it to bypass voice authentication.

OS level access control on accessibility does not protect system apps. In particular, we found that our malware can change system settings through AT, which offers us many powerful capabilities.

The only missing check in Android is at the application level. Similar to the iOS case, we did not find new capabilities beyond what is enabled due to inconsistent OS-level checks.

Attack #9: bypass Touchless Control's voice authentication. Fragile authentication for AT leads to a vulnerability in Touchless Control on the Moto X. In particular, voice authentication can be bypassed by a replay attack shown in Figure 9. First, an attacker can build malware as a background service that constantly monitors sound from the microphone. As the phrase "OK Google Now" is the only authentication phrase, the user is likely to repeat it frequently. The malware can easily record the authentication phrase. Once recorded, the malware can play the recorded phrase through the device speaker to activate Touchless Control. Since Touchless Control accepts self-played sound from the speaker to the microphone, it subsequently launches Google Now. After this, the malware can play arbitrary commands using the default TTS library for Google Now. Since there is no further authentication for the command phrase, the malware can utilize a variety of commands to make phone calls, send SMS, send email, set alarms, and launch applications.

Attack #10: bypassing Android sandboxing and more. Sandboxing in Android [1] provides isolation between apps to protect memory and filesystem access, and prohibits an app from interfering with the execution of other apps. Furthermore, its permission system restricts an app's access to sensitive resources.

However, once an app is activated as an AT, there are no further restrictions. A malicious AT can then read UI structure (including location, type, text, etc.) of the whole system and deliver user actions to any UI element, such as the click of a button, a scroll up or down, a cut/copy/paste of text, a change of focus, and expand/dismiss of UI. Therefore, malware can control other apps as if it is the user. Malware can abuse the permissions of other apps, e.g., even without network permission, our malware can control the Gmail application to exfiltrate stolen data.

In addition, malware can change system settings such as user-configurable settings, and install/uninstall apps. Moreover, malware can programmatically enable developer mode (e.g., by sending 7 synthetic clicks) which can put a device at risk for further infection.

4.4 Vulnerabilities in Output Validation

Table 3 summarizes the evaluation results of each platform compared against our output reference model in Figure 4. iOS does not support alternative output, so its result is omitted in this section.

Platform	Reading of UI Structure	Password Protection
Windows	UIPI	Yes
Ubuntu	None	Yes*
iOS	N/A	N/A
Android	None	Settings*

Table 3: Status of output validation on each platform. * means the check enforces an inconsistent security policy.

Across all platforms, only Windows enforces an OS-level check (UIPI) for output. However, since UIPI does not have any protection among applications in the same IL, Windows suffers from the same UI-level attacks described below.

Reading UI state of other applications. All platforms except iOS allow an AT to access UI structures. The library provides not only the metadata for the UI such as type of element, location, and size, but also the content of the UI element. Hence, a malicious AT can monitor other applications in a fine-grained manner. For example, malware can detect the current state of the target application using 1) available UI structures, 2) UI events such as change of focus, movement of window, change of contents, and 3) user interaction events. With these capabilities, malware can spy on every action a user takes, as well as maintain an accurate status of an application.

All three platforms (Windows, Ubuntu, and Android) protect the plaintext content of a password in a password dialog box by default. However, in Ubuntu, AT-SPI fails to block all paths for retrieving the plaintext of password (#11). Android can be configured to allow reading keystrokes on password dialog boxes; this can be enabled by malware implemented as an AT (as mentioned in #10).

Implementation of attacks. For extracting passwords in Ubuntu (attack #11), we implemented proof-of-concept malware that looks for authentication windows, obtains the plaintext, and prints out the plaintext on the console using AT-SPI. For attack #12, we implemented malware that enables the speaking of passwords via accessibility services and registers itself as the TTS subsystem for the accessibility service. In this twofold manner, malware can receive and transmit the contents of a password to an attacker.

4.4.1 Windows

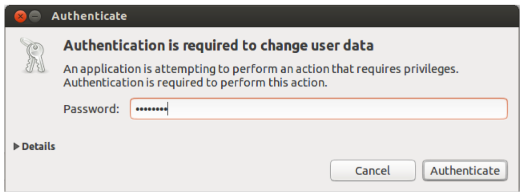
With UIPI, Windows is the only platform where the OS applies access control on the reading of UI structures. Although UIPI prohibits accessing the structures of an application that has higher IL than the caller, access on the same or lower IL is still permitted.

The application level output check exists for password boxes by default, which disallows 1) obtaining the password via `WM_GETTEXT` or `ValuePattern` in UI Automation, and 2) copying the password via "`WM_COPY`"or by generating a Ctrl-C input event. Therefore, malware cannot steal passwords through the accessibility library.

4.4.2 Ubuntu Linux Desktop

In Ubuntu, the application level check for passwords exists, but its implementation (in ATK) is inconsistent with the UI (in GTK).

Attack #11: stealing sudoer passwords from authentication dialogs. On Ubuntu, we found a password stealing vulnerability using


Authenticate

Authentication is required to change user data

An application is attempting to perform an action that requires privileges. Authentication is required to perform this action.

Password: ••••••••

Figure 10: Administrator authentication dialog of GNOME on Ubuntu 13.10. It asks for the password of the current user to gain root permissions.

AT-SPI. The security checks at the OS level are incomplete. For a password box, there exists an API call, `gettextvalue()`, on the Linux Desktop Testing Project (LDTP, a wrapper over AT-SPI and ATK). It throws a "Not Implemented" exception when called, meaning that reading passwords through this API is unavailable. However, AT-SPI missed security checks on a critical accessibility function of a password box: `copytext()`. Although physically or synthetically pressing Ctrl-C does not copy the value of a password box, `copytext()` from AT-SPI does copy the plaintext of a password to the clipboard. The clipboard then can provide the plaintext content of a password. Figure 10 shows a sudo dialog that is vulnerable to this attack. Once the sudoer's password is acquired in this manner, malware can easily gain root privileges.

4.4.3 Android

While Android prohibits reading of password content from its accessibility service, this can be disabled via user preferences. In conjunction with the vulnerability of input validation (attack #10), this restriction can easily be bypassed.

Attack #12: keylogger on Android. Although Android provides protections for accessing the plaintext of a password, incomplete protections at the OS-level lead to a vulnerability. Once an app is enabled as an AT (see Attack #10 for detail), the app can change any settings on the device without user consent. Android provides an option called "Speak passwords" in its accessibility settings. If enabled, keystrokes on a password box are delivered through the text-to-speech (TTS) processor. We register malware as a TTS output application. Once registered, the malware can receive password contents via the OS-level accessibility service.

5 Discussion

In this section, we explain how accessibility libraries are making it possible to implement our attacks, discuss the limitations of our attacks, analyze the root causes of the vulnerabilities, and consider open problems for future work.

5.1 Complexity of Accessibility Attacks

As we mentioned in section §2, accessibility libraries provide three capabilities: 1) obtaining events representing UI change, 2) providing a way of programmatically probing/accessing UI widgets, and 3) synthesizing inputs to UI widgets.

With these functionalities, an attacker can create malware capable of performing successful attacks with a degree of relative ease when compared to other non-AT methods that achieve the same ends. As an example, we will describe how the "Password Eye" attack (#3) can be implemented using accessibility libraries. To achieve the "Password Eye" attack, malware needs to: 1) detect when the user types a password, 2) identify the UI "eye" and click on it, and 3) locate the password field to grab its text in a screenshot. To determine whether the user is typing password, we can use the first capability of the accessibility libraries to keep track of which UI component is currently focused. In particular, on the Windows platform, this can be easily achieved by registering an event handler in the platform's accessibility framework that receives the focused UI element at any change of focus. After being handed a focused element, we can check whether the element is a password box with an "eye" by assessing its properties reported by the accessibility libraries. For example, a `TRUE` value of `isPassword` property indicates a password box. Once we determine that the focused element is a password box, we can use the second capability of the accessibility libraries to get the "eye" button. In particular, since we know the relative position of the focused text box and the "eye" button, we can walk the UI widget tree provided by the accessibility library and calculate the position of the "eye". Then, we can use the third capability to click it. Finally, the handle to the focused password box we obtained in the first step can also be used to retrieve the location of the box on screen, and allow us to grab the actual password typed from a screenshot. A point worth noting here is that developing attacks using accessibility libraries is very similar to how one manipulates DOM (Document Object Model) objects using Javascript in a web page.

One may point out that the same attack can be achieved on the Windows platform by sending traditional Windows Messages (such as `WM_CLICK`), or using tools such as AutoIt. However, we argue that the use of the accessibility library greater ease and reliability. In particular, without the first capability of the accessibility libraries, one may need to constantly probe the current state of UIs to determine if the user is typing a password. Secondly, while it maybe trivial to use a hardcoded coordinate to click the "eye" button in a testing environment such as AutoIt, this strategy will be very fragile in a real attack; factors such as variation in screen size and resizing/moving of the target window may break the hardcoded approach in a real attack. Using hardcoded locations to extract a password from screenshots will face a similar issue. Even though it may be possible to reliably implement our attacks without accessibility libraries, this implementation would be more complex and require greater effort on the part of the author.

5.2 Limitations of the Attacks

Since attacks through accessibility libraries perform actions over user interfaces, they have an inherent limitation in that they are not stealthy. For example, if the target application is running in the foreground when an attack is unfolding, the user may recognize visual cues of the attack, such as button presses, opening of a new UI window, etc. Furthermore, attacks via voice commands play sounds, and are thus audible; or they fail if the speaker is turned off.

However, we argue that these attacks *can* be launched in a stealthy way. First, malware can detect whether the user is using the device or not. For desktop machines, the presence of a user can be detected by monitoring physical keystroke or mouse movement. Malware can exploit a time period when the user is absent to launch UI-intensive attacks. If necessary, the malware can blank the screen when launching the attack, because screen sleep after some period of non-use is a natural and expected behavior of the system. For mobile devices, prior researches [17, 31, 35] discussed how to track the user's behavior using an app on the device. With the help of various sensors, such as the camera, face proximity detector, GPS location, accelerometer, etc., malware can determine when the user is not watching the screen, away from the device, or when the device is in the user's pocket. It can then launch an attack without being exposed.

Second, UI actions can be delivered in the background for some platforms. Thus, an attack can be carried out even when the user is actively using the device. In Windows, once a handle to a UI widget is obtained while it is in the foreground, it can still be manipulated even when it is in the background or minimized. In Linux, probing the UI of a minimized application is possible. Furthermore, in the worst case, malware can move a window to nearly off the screen, so that the user does not notice any UI change. In our experiment, if any pixel of an app is visible on the screen[7], there is no limitation on probing or performing actions on it.

Third, it is possible to make the attacks on natural language user interfaces stealthy with the help of hardware. Common audio devices such as the Realtek HD Audio device and other sound card devices' drivers provide functionality called `Stereo Mix`. `Stereo Mix` sends the output of system sound to an internal microphone input. Enabling this functionality does not require any special privilege. Malware can play audio *internally* to deliver text-to-speech audio to a natural language user interface. The attack succeeds without outputting audio to speakers, and also works when there is no speaker device at all.

Finally, our experience with OS vendors shows that these threats will be taken seriously. In May, 2013, before presenting an attack [19] that takes advantage of private APIs for synthesizing touches and taking screenshots on iPhone, we informed Apple of our attack. In Aug, 2013, the exploited vulnerabilities were removed from the then newly-released iOS 7.

5.3 Root Causes, and Design Trade-offs

We strongly believe that to fundamentally eliminate a11y-related vulnerabilities, a new architecture for providing accessibility features is necessary. However, proposing such an architecture is out of the scope of this paper; instead, we present the findings of our root cause analysis to illustrate why security checks spread across the AT, OS, and application tend to fail, and to show some of the trade-offs taken in the current implementation of accessibility features.

The first identified root cause is the emphasis of availability/compatibility of a11y support in all the studied systems. In every case we have studied, native UI widgets include logic to handle requests from accessibility libraries, and UI widgets provided by OS are usually built to reuse the same interfaces/channels to handle both real user inputs and a11y inputs. As a result, it is very hard for an application to distinguish a11y inputs from real user inputs. This design choice enables many attacks by accepting and processing synthesized input as if it is a real input (A2[8]). For instance, in Android, physically tapping an UI widget with a finger will invoke the `performClick()` function. Equally, on an a11y request, the same `performClick()` function is invoked (see Example 1 in Appendix for details). In Windows, just like user real input, clicks generated by `UIAutomation` are delivered as a Windows Message `WM_CLICK`. Similarly, for Ubuntu and iOS, a11y requests take the same path as I/O requests within the UI widget. While this means all applications that use the native UI widgets automatically and naturally work with the requests from accessibility libraries, such design also imposes a default security policy that makes every widget available to all ATs. As we can see in attack #2 and #3, this is too permissive a policy. Furthermore, in all the studied systems, if the application/UI developers were to instead implement their own policy regarding how an application should process requests from accessibility libraries, they would have to implement their own UI widgets (usually by "subclassing" the native ones), and this comes with a non-trivial cost.

Second, from both technical and economic perspectives, it is challenging to perform complete validation and authentication for certain inputs introduced by AT. As a result, new attack vectors become available due to missing security checks on processing input (A2) and output (A3) requests from ATs or accessibility libraries. For example, in attack #11, simply pressing Ctrl-C will call `gtk_entry_copy_clipboard` in which there is a security check for preventing text in a password field from being copied (see Example 2 in Appendix for details). However, a different function `copytext()` will be executed in ATK, which takes a different execution path without security checks, potentially leading to password leakages. We suspect that the ATK code was added to the OS by a group of developers who were not aware of the principles of input validation and complete mediation, or that the ATK code was added to the OSs only recently and has thus not been through rigorous security code review and testing when compared to older portions of the OS.

There are also technical and economic reasons for a lack of validation and authentication. For example, for the cases of attack #1 and #9, the AT needs to check whether the voice input actually comes from a real user, and also needs to further authenticate the authorized user. Voice based validation and authentication requires non-trivial technical support, with potentially high research and development costs.

Finally, to improve the usability of ATs, OSs usually have weak access control on accessibility libraries; while this makes the installation and use of ATs (their intended purpose) easy, it is not a good security practice. In particular, accessibility libraries can usually be accessed by any application on a system. For example, in Windows, iOS 6, and Linux, any program can be an AT without any authorization. This also opens paths for attack so that any (malicious) program can abuse accessibility functionalities to launch the attacks described in this paper. The exception is Android; it has a setup menu for enabling an app's use of the accessibility library, though this check is only performed at initial app setup.

5.4 Recommendations and Open Problems

Based on the root cause analysis in Section §5.3, we present recommendations on how to alleviate (if not eliminate) the security risks created by a11y support. Our recommendations are intended to work with the current architecture for supporting accessibility features, and thus are limited by the inherent difficulties that come with this architecture; nonetheless, we believe they will help the community to improve security for a11y before the introduction of a complete a11y security policy occurs. We will also discuss some open problems involved in implementing these recommendations.

Our first recommendation is to have fine-grained access control over which program can access specific functionality of the accessibility library. From our study, we find that both Linux and iOS have no such access control at all, while Windows allows all programs to use the accessibility library to control/read the content of any other program with the same integrity level. Android appears to be the only system that has access control policy specific for the accessibility library: the user has to specifically grant the AT the privilege to use the accessibility library. However, once this privilege is granted, the AT has full access to all the capabilities of the accessibility library. In many cases, this violates the principle of least-privileged access. For example, a screenreader will only need to read the content of other apps through the accessibility library, but it does not need to be able to interact with other apps. Based on this observation, we recommend the privilege of using the accessibility library be at least split into two, one for reading the content of other apps and one for the more privileged capability of interacting and controlling other apps. While this may present an extra hurdle for users who need AT, it will only incur a one-time set up cost,

[7]For example, only one pixel of the window is visible, while all others are invisible.

[8]Please refer to section §3.1 New Attack Paths for details.

which we feel is an acceptable trade-off for the extra security against misuse of the accessibility library.

Our second recommendation is to provide mechanisms for a UI developer to flag how different widgets in their UI will handle various requests from the AT, rather than requiring the UI developer to handle this task themselves. For example, in many UI libraries, a developer can flag a text field as a password field, and the underlying logic of the UI will make the content in the field invisible to both the display and the ATs. However, this generally appears to be the only instance of such a flag, and it only applies to text fields. We believe more such flags should be available to specify various a11y related security policies, and such flags should be made applicable to various kinds of widgets (e.g. attack #2 and #3 can be easily eliminated if a security flag is applicable to buttons). As future work, we will study what kind of a11y related security policies UI developers usually need to specify, and what language features are needed for specifying such policy as attributes of widgets in the UI.

Our final recommendation can be considered a new security component in the current a11y architecture, and can significantly limit the damage caused by exploitation of a11y-related vulnerabilities. We propose to extend accessibility support to user-driven access control mechanisms like UAC in Windows or Remote View in iOS. While this recommendation may not be directly derived from our root cause analysis, we believe it will fundamentally eliminate many a11y related security issues discussed in this paper. In particular, OS vendors should develop versions of access control mechanisms to support various disabilities. For example, for visually impaired users, the system can read out (through the speaker) the message seeking permission, and have the user confirm or abort by clicking the "F" or "J" button on the keyboard (which are tactilely different from all other keys on the keyboard), and for the users who lack fine motor skills, the permission granting can be driven by voice recognition. We note that while this approach is not general enough to support the need for all users with different kinds of disabilities, it will significantly improve the security for all users that are covered. Furthermore, in the case of voice recognition, the introduction of a mechanism specifically designed for seeking vocal permission may significantly simplify the task of authenticating user input (only "yes" or "no" need be verified, rather than performing general voice recognition), and thus move the burden of performing voice recognition from the AT developer to the OS vendor (who may have more resources to research and develop a mechanism that is robust against attack #1 and #9).

Finally, we acknowledge that our analysis requires significant manual effort and reverse engineering work and thus is not exhaustive. We will leave it as an open problem to design systems that can automatically find a11y related vulnerabilities. We believe this will be a challenging problem for the several reasons. First, automatically detecting a11y functions and analyzing their related vulnerabilities requires whole system analysis. Since an a11y request is regarded as an I/O event, it is processed asynchronously. As a result, it is very hard to find entry points. The complicated execution of a11y logic extends to many different low-level modules, which usually make use of many (function) pointers. Proprietary OSs do not provide source code, and so researchers can only perform analysis with the compiled binary, which makes the task even harder. Second, unlike general programming errors, confirming a11y related vulnerabilities requires a deep understanding of the semantics of an application, which significantly limits the scalability of such analysis. We hope that our work can motivate further studies toward this direction.

6 Related Work

Attacks on Windows. In 2007, it was reported that an attacker could control a Windows Vista machine by playing sound to Speech Recognition [28]. However, since the attack could not bypass UAC and assumed Speech Recognition was already enabled, it was considered a minor bug at that time. Compared with this attack, our attack (attack #1) does not require Speech Recognition to be enabled before the attack, and we can bypass UAC on Windows 7 through 8.1 (due to policy changes in UAC [26]).

Just before the release of Windows 7, there was a UAC bypass attack [10] that exploited the special capability of `Explorer.exe` to write to system directories. In this attack, a malware process will attach to `Explorer.exe`, inject code, and exploit its capability to write to system directories. Our attack #2 follows the same strategy, but instead of using low level function `WriteProcessMemory()` to inject code into `Explorer.exe`, we used the accessibility library to simply click the "OK" button.

Attacks on iOS. Recently, it was reported [11] that Siri in iOS 7 can be exploited to bypass the lock screen and send email, SMS, post on Twitter and Facebook, make phone calls, etc. We referred to this attack as attack #5 in the vulnerability section.

Although the accessibility library is a private API that is not usable by regular app developers, the threat is real. Last year, an attack [33] showed that it is possible to circumvent the Apple App Store review process by successfully publishing an App Store app that invoked private API calls.

Attacks on Android. In Android, there have been many attacks on the permissions [7, 9, 13, 16, 34] and private information [18, 36] of an app that demonstrate data leakage through Android's IPC channel. To address these problems, many mechanisms [6, 12, 16, 20] have been proposed. Unfortunately, since all of the proposed mechanisms were focused on the official IPC channel, they are not able to prevent attacks through accessibility libraries. Furthermore, our attacks can steal the capabilities and private information of other apps.

7 Conclusion

In compliance with the amendment to the Rehabilitation Act of 1973, software vendors have been continuously adding accessibility features to their OSs. As the technology advances, accessibility features have become complex enough to comprise a complete I/O subsystem on a platform. In this paper, we performed an analysis of the security of accessibility features on four popular platforms. We discovered vulnerabilities that led to twelve practical attacks that are enabled via accessibility features. Further analysis shows that the root cause of the problem is due to the design and implementation of a11y support requiring trade-offs between compatibility, usability, and security. We conclude with proposing several recommendations to either make the implementation of all necessary security checks easier, or to alleviate the impact of incomplete checks.

Acknowledgments

The authors would like to thank the anonymous reviewers and our shepherd, Trent Jaeger, for their help and feedback, as well as our operations staff for their proofreading efforts. This material is based upon work supported in part by the National Science Foundation under Grants No. CNS-1017265, CNS-0831300, and CNS-1149051, by the Office of Naval Research under Grant No. N000140911042, by the Department of Homeland Security under contract No. N66001-12-C-0133, and by the United States Air Force under Contract No. FA8650-10-C-7025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the

National Science Foundation, the Office of Naval Research, the Department of Homeland Security, or the United States Air Force.

References

[1] Android Developers. Security tips. http://developer.android.com/training/articles/security-tips.html.

[2] Apple, Inc. Accessibility. http://www.apple.com/accessibility/resources/, .

[3] Apple, Inc. The ios environment. https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/TheiOSEnvironment/TheiOSEnvironment.html, .

[4] O. Begemann. Remote View Controllers in iOS 6. http://oleb.net/blog/2012/10/remote-view-controllers-in-ios-6/.

[5] K. J. Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, volume 17, pages 18–25, 2012.

[7] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[8] K. Cook. [patch] security: Yama lsm. http://lwn.net/Articles/393012/.

[9] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Information Security*, pages 346–360. Springer, 2011.

[10] L. Davidson. Windows 7 uac whitelist: Proof-of-concept source code. http://www.pretentiousname.com/misc/W7E_Source/win7_uac_poc_details.html.

[11] J. Edwards. There's a huge password security quirk in ios 7 that lets siri control your iphone. http://www.businessinsider.com/password-security-flaw-in-ios-7-lets-siri-control-your-iphone-2013-9.

[12] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, pages 1–6, 2010.

[13] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[14] Go Launcher Dev Team. Go launcher ex notification. https://play.google.com/store/apps/details?id=com.gau.golauncherex.notification.

[15] Google Inc. Section 508 Compliance (VPAT). https://www.google.com/sites/accessibility.html.

[16] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[17] J. Han, E. Owusu, L. Nguyen, A. Perrig, and J. Zhang. Accomplice: Location inference using accelerometers on smartphones. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*, pages 1–9, Jan 2012.

[18] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *IEEE Symposium on Security and Privacy (Oakland '12)*, 2012.

[19] B. Lau, Y. Jang, C. Song, T. Wang, P. H. Chung, and P. Royal. Mactans: Injecting malware into iOS devices via malicious chargers. In *Proceedings of Black Hat USA*, 2013.

[20] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[21] C. McLawhorn. Recent development: Leveling the accessibility playing field: Section 508 of the rehabilitation act. *NORTH CAROLINA JOURNAL OF LAW & TECHNOLOGY*, 3(1):63–100, 2001.

[22] Microsoft. Windows integrity mechanism design. http://msdn.microsoft.com/en-us/library/bb625963.aspx, .

[23] Microsoft. Windows vista integrity mechanism technical reference. http://msdn.microsoft.com/en-us/library/bb625964.aspx, .

[24] Microsoft Corporation. Microsoft and section 508. http://www.microsoft.com/enable/microsoft/section508.aspx, .

[25] Microsoft Corporation. User account control. http://windows.microsoft.com/en-us/windows7/products/features/user-account-control, .

[26] S. Motiee, K. Hawkey, and K. Beznosov. Do windows users follow the principle of least privilege?: investigating user account control practices. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, SOUPS '10, New York, NY, USA, 2010. ACM.

[27] Motorola Inc. Moto X Features: Touchless Control. http://www.motorola.com/us/Moto-X-Features-Touchless-Control/motox-features-2-touchless.html.

[28] G. Ou. Vista Speech Command exposes remote exploit. http://www.zdnet.com/blog/ou/vista-speech-command-exposes-remote-exploit/416.

[29] PoPs. Pops ringtons & notifications. https://play.google.com/store/apps/details?id=com.pops.app.

[30] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[31] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, 2011.

[32] The United States Government. Section 508 Of The Rehabilitation Act. http://www.section508.gov/Section-508-Of-The-Rehabilitation-Act.

[33] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In *The 22nd USENIX Security Symposium (SECURITY)*, 2013.

[34] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on Android security. In *ACM conference on Computer and communications security (CCS '13)*, 2013.

[35] N. Xu, F. Zhang, Y. Luo, W. Jia, D. Xuan, and J. Teng. Stealthy video capturer: A new video-based spyware in 3g smartphones. In *Proceedings of the Second ACM Conference on Wireless Network Security*, WiSec '09, New York, NY, USA, 2009. ACM.

[36] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*, 2013.

APPENDIX

A Examples

```
// On real touch event
public boolean onTouchEvent(MotionEvent event) {
  switch (event.getAction()) {
    case MotionEvent.ACTION_UP:
    {
      // ...
      // performClick() is called to handle real click event
      performClick();
      // ...
    }
  }
}

// On a11y request for click
boolean performAccessibilityActionInternal(int action,
                                    Bundle arguments) {
  // ...
  switch (action) {
    case AccessibilityNodeInfo.ACTION_CLICK:
    {
      if (isClickable()) {
        // the same performClick() is invoked to handle a11y request
        performClick();
        return true;
      }
    } break;
  }
  // ...
}
```

Example 1: Code that handles real input (*above*), and code that handles a11y input (*below*) for click, in `View.java` of Android. The same function `performClick()` is used to handle both requests.

```
static void gtk_entry_copy_clipboard (GtkEntry *entry) {
  GtkEntryPrivate *priv = entry->priv;
  // ...
  // ### security check for password box ###
  if (!priv->visible)
  {
    // do not copy text to clipboard
    gtk_widget_error_bell (GTK_WIDGET (entry));
    return;
  }
  // ...
}
```

Example 2: Code that handles copy of text (pressing Ctrl-C) in GTK. Inside the function, GTK checks the security flag `priv->visible` to decide whether or not to provide selected text to the clipboard. If GtkEntry is set as password box (flag is true), then text will not be provided.

```
// A11y code snippet
void atk_editable_text_copy_text (Editable e, int start, int end) {
  AtkEditableText *text;
  // ...
  *(iface->copy_text) (text, start_pos, );
  // calls gtk_entry_accessible_copy_text()
}

static void gtk_entry_accessible_copy_text(AtkEditableText *t,
                                           int start, int end) {
  GtkEditable *e;
  // ...
  gchar *str = gtk_editable_get_chars (e, start, end);
  // ...
}
// A11y code end, calls functions in Gtk UI

// Gtk code snippet
gchar* gtk_editable_get_chars (GtkEditable *e,
                               int start, int end) {
  return (editable)->get_chars (e, start, end);
  // calls gtk_entry_get_chars()
}

// Final function that returns text content
gchar* gtk_entry_get_chars (GtkEntry *e, int start, int end) {
  gchar *text;
  text = gtk_entry_buffer_get_text (get_buffer (entry));
  // ### no security checks at all on getting text ###
  return g_strndup (text + start_index, end_index - start_index);
  // return text without checking priv->visible
}
```

Example 3: Code that handles an accessibility request (ATK) for copying text. In ATK, it calls a function of a module in GTK that supports accessibility. The module then calls a function that directly interacts with the UI widget (GTK functions). However, the module GtkEntryAccessible calls a different function `gtk_editable_get_chars()`, which misses required security checks of the password box.