AudioJacked - Theft of Money and Information Through a Phone's Audio Port

Alex Gould, `COMP116`

Ming Chow

Video

The video accompanying this paper can be found at:

`https://drive.google.com/file/d/0B_s2ARkgeNzwUE12OGVCeFpwdzg/view?usp=sharing`.

Acknowledgments

Immense thanks to William Woodruff at the University of Maryland College Park, who first discovered this erratic behavior due to a pair of faulty headphones, and was constantly available to run new tests with them and supply new ideas. This project would not exist without you, in every sense of the word.

Many thanks to Maretta Morovitz, who provided invaluable editing assistance (and helped me locate breadboards when I thought there were none). Many thanks for the help, and thanks for a great term!

And a huge thanks to Max Bernstien and Chris Gregg for lending me a 3.5mm audio cable during an emergency!

Abstract

In this paper, I demonstrate how software interrupts can be sent through a standard headphone port found on most smartphones that direct the phone's operating system to send personal information about the recipient to an attacker, as well as add charges to a user's phone bill that are paid directly to the attacker's account.

To the Community

I chose this topic due to my love of low-level machine coding, but I discovered the bug in particular due to a happy accident, when a frayed headphone cable caused erratic behaviors in a friend's iPhone. The breakthrough seemed so clever that I decided to make it the focus of this project. (I also have a love affair with '90s computer systems, so something like this seemed right for me.)

Why should you care? For the same reason that you don't protect your house with a state-of-the-art security system and then leave the door unlocked. There's constant awareness of techniques like "juice jacking" through USB chargers or malicious apps from the cloud, but next to no awareness about other mechanics of the phone, which can lead people to believe that these parts of the device are safe. Case in point, you've probably given far more thought before inserting an unknown flash drive than before plugging an unknown AUX cord into your phone.

It's vital that awareness be given to all parts of a system, even parts that don't seem like they can hurt you.

Introduction

Sometimes it's difficult to remember that the computers we carry around every day evolved from simple cell phones, but when the smartphone was first invented, interfacing the operating system with the underlying phone system was of paramount importance, to the point where the phone components have almost become part of the kernel itself, a given on any mobile platform that's usually not payed much mind. As smartphones have evolved, we think of them more and more as computers, and the underlying "phone" components are forgotten, as more and more effort goes towards fixing security holes in the complex programs running on the device.

However, this means that the phone components have not been given nearly as much security attention as they should. I aim to demonstrate how simple use of the phone components of a smartphone system can be used to cause a security hole the OS above is almost unaware of. The attack vector is a part of the phone ubiquitously used, but almost never thought about; the 3.5mm headphone jack has existed on mobile phones relatively unchanged since the 1990's, and is regarded as completely benign by most users. However, a critical lapse in the way it interacts with the phone's OS leaves many users vulnerable.

Prior Research

There has been a recent flurry of research into audio port manipulation due to the proliferation of Mobile Point of Sale systems like the Square reader, which plug into an audio port and transmit credit card data to a secure app as a sound wave. A paper[1] submitted to BlackHat in 2015 exposed a flaw that allowed malicious merchants to record these waves and decrypt them into the numbers themselves. While decryption was made impossible via an updated[2] reader with a built in encryption chip, the problem still remains that there are almost no ways to control where the signal from the audio port goes once it's read by the device. It's this key vulnerability that we will exploit.

Procedure

In the absence of any unifying standards at the outset of smartphones, it's no surprise that the two major operating systems take a very different approach to handling events over headphones.

iOS

Apple, as to be expected, wields far tighter control of what the headphone jack is and isn't allowed to do. This isn't documented anywhere, either, so I had to bust out the meters and some unofficial specs[4] and do some hands-on research.

Only one interrupt seems to be explicitly available, and it's accessed through a particularly unfriendly medium. For readers unaware, the concentric rings on the headphone jack correspond to 4 pins inside the device, with each ring touching a single pin. Here's the pin layout[4] for Apple devices:

Pin #	Pin Name	Use
1	Tip	Left Speaker Out
2	Ring	Right Speaker Out
3	Ring	Ground/Common Power
4	Sleeve	Microphone In

There's only one change in voltage the OS seems to respond to: the shorting together of the ground and microphone pins, which happens when the large button is pressed on the headphones. The length of time is measured on an internal system clock, and the appropriate action is taken, usually activating Siri or playing music.

As iOS devices support only one headphone action, this action is deeply ingrained in the OS. No applications have access to it, and no parameters can be passed like they could be with a function call, only a blunt hardware trigger with a length parameter. Anything more fine-tuned than that is handled by Siri.

Penetrating a walled garden is tiring, and the likelihood that we can get to the texting app is low, since it was probably never given a trigger. Instead, let's look at a system we *can* play around with.

Android

Android had a much different origin story than iOS, as it was designed to be open to developers. As a result, many low-level OS functions are available, deep in the sub levels of the Android API, including a table in the API for recognizing hardware button presses from the kernel[5]. This part of the documentation isn't very clear, so we can theorize some sort of abstract `Intent` class or other abstraction mechanism hides this from the applications running on the device. But the fact remains that, when a button is pressed, a constant is made available to all programs running on the device, and all of these constants can be loaded into any app from the `KeyEvent` class.

Now, it's all well and good knowing that we can translate `0x00000005` to `KEYCODE_PLACE_CALL`, and get button codes for testing, but that doesn't solve our original problem, which is how to trigger these events from an audio signal without human intervention.

Once again, we're going to have to stray a bit from the official documentation here and do a little experimenting, but Android provides some excellent documentation[6] to help us get started. Android uses the same jack as iOS does, so the table above is still accurate, but Android is far more lenient about the buttons required, only mandating a simple play-pause button with optional volume controls. But what's more interesting is the electrical specifications, which give the following precise values for each function:

Supplementary Function	Impedance
Play/Pause/Hook/etc.	0Ω
Vol+	240Ω
Vol-	470Ω
Nexus 5+ Only (Reserved by OS)	135Ω

It seems quite possible that the internals of an Android-compatible headset are simply resistors laid in parallel, (the spec[6] lists this way, among others), but the real breakthrough comes when later in the document, these events are directly mapped to keycodes. When the Android kernel receives a resistance, say of 240Ω, it automatically broadcasts the `KEY_VOLUMEUP` constant to any apps listening, with no checks to see if the user pressed the button or not. So this begs the question, what apps are listening?

The list of keycodes the spec[6] gives us is underwhelming to say the least; simply changing volume and activating keys like `KEYCODE_HEADSETHOOK`, which appear to trigger things like the phone app when you need to hang up, are hardly useful exploit routes. We're after something bigger, though. We want to be able to send a text message, or in the event that's impossible, dial a number. We know the latter to be possible, partly because there's a keycode for it, and partly because my friend saw it happen when a faulty audio cable dialed a number in his new Android phone's address book multiple times. So, how do we access those hidden keycodes?

After going over the incident again, I realized that we might have stumbled onto an edge case. The faulty cable belonged to a pair of Apple earbuds, while the phone was an Android device. Could it be possible that the unanticipated resistance of the frayed cable happened to be a value that triggered an undocumented keycode, perhaps the one to make a call?

I ran my tests using a Samsung Galaxy S5, and basically just plugged a male-to-male 3.5mm audio cable from my phone to a computer, applied 1000Ω resistance on the microphone pin (spec standard) and started varying the resistance. All the official actions worked, and I was able to find some extra ones.

Resistance	Action
650Ω	Open address book
700Ω	Place a phone call
810Ω	Quit all runnng apps

It should be fairly obvious which one we're interested in.

(The 650Ω action may have been a kernel bug, as the spec shows that to be well within the range for "Vol-", which runs from 360-680Ω.)

It's my guess that these values may vary from Android device to Android device, but I didn't have time to test that to be sure. It's also possible that these exploits could be due to a hardware error. I sincerely hope this isn't, so if any reader can replicate this on another device, I'd love to see it!

My email is `alxg833@gmail.com`. I'd seriously love to see this gain ground.

Payload

So now that we've gained the ability to perform actions on an Android device using a simple resistance change in the headphone jack (I'm considering automating it via an Arduino or a Pi), it's time to figure out the best thing to do with this capability. For the sake of simplicity and practicality, I'll assume we're limited to placing instantaneous phone calls.

- **Propagation of Worms** - Should an attacker need to quickly build a network connected to an infected device, calls can be placed to any number in the phone's address book. Given numbers, encoded as changes in resistance if the phone model permits, could be used to forward these calls to a waiting number owned by the attacker.
- **Malicious Voicemails** - Sending a victim's phone number to a waiting attacker could allow for customized phishing voicemails, based on the location of the malicious device, or possibly even the numbers in the victim's address book.
- **Extension Abuse** - Precious little exists in terms of managing extensions sent along with the phone number when dialed. An attacker's server could

be configured to expect data sent along as extensions, possibly taken from the victim's device's OS itself, assuming more keycodes are available than the ones found in this paper.

If we add texting to the mix, there's the potential for the theft of money. Donations by text to charity are quite common and operate very simply. A donation amount is sent to a server via SMS, which then contacts the phone company and has that amount added to your phone bill. Usually there are safeguards, such as confirmations and notifications of donation, but an attacker could feasibly implement such a system without such safeguards. As a result, a simple signal from an audio cable could trigger a text message instructing the victim's telephone provider to pay an attacker.

Applications

The ability to deduct money from another person's account by simply plugging in a device has huge implications for security. Even if the software interrupts are localized to a single operating system, this puts countless devices at risk even if a few doctored devices are used in highly trafficked areas.

Obviously there are many devices that use the 3.5mm jack, for audio or otherwise, giving us multiple possible attack vectors.

- **Headphones/Earbuds** - Not suspicious, and have a very high likelihood of connecting to multiple devices. However, being small, a signal generator likely wouldn't fit inside without arousing suspicion.
- **Large Speakers** - Probably the most likely success, large speakers have ample room for any sort of equipment we need to add, and are incredibly likely to be plugged into strangers' phones. However, their travel range is likely much less far than headphones, and they're more expensive to obtain.
- **Credit Card Readers** - Readers like Square and the like have a key advantage in that they only are used when money enters the account of the user, making them less likely to notice a small amount of money exiting their account at the same time. However, these readers are nearly impossible to obtain by a non-licensed merchant, and are small and fragile to the point where 2 are needed to produce 1 fake[1].
- **Car Audio Systems** - Usage patterns of a car would likely not hit a high enough amount of targets to justify the immense barriers to getting hardware or software undetected into the car of a victim.
- **Hobbyist Systems** - Video through audio jack[3] and other such novel uses of the port would likely be very easy to compromise with malicious code. Unfortunately, these uses aren't widely known, and the demographic they are used by is likely to have far more secure hardware than the users of any other attack vector here.

Defenses

A plan of defense is difficult, but significantly helped by the fact that devices must be physically compromised in order to become malicious. As a result, users should be kept safe so long as they use their own trusted devices.

Most of the danger comes from instinctively plugging a device in while at a party or some other group activity. In these cases, not much can be done other than to encourage an attitude of constant vigilance. Should you be in a situation where plugging the device in is inevitable, the attack can also be thwarted by simply enabling Airplane Mode, however this can be inconvenient.

With regards to manufacturers, I would recommend implementing rigid checks on what interrupts are and aren't allowed to do, possibly adding a setting disabling use of the microphone pin on the device, much how read-only buffers are used with suspicious flash drives. This could be done through a normal operating system update, with no hardware modification necessary. With this setting enabled, data would be able to travel to the speaker, but none can be received.

Conclusion

As we can see from this paper, it is frighteningly easy to gain unintended access to a system purely by manipulation of its older components.

Our modern Internet is built on the backs of telephone systems and old computer hardware. Losing sight of security holes in these foundations can open us up to all sorts of trouble down the road. If an exploit as simple as this can bypass an entire secure operating system, there could be thousands of these bugs lying in wait in the foundations of vital systems. Shellshock was one, resting in the code of one of the most widely used shells in the world. Heartbleed was another, hiding in ubiquitous server software. I hope that this paper gives security professionals the motivation to start at the bottom up, and not rely on the infallibility of technical stacks that could very well be vulnerable.

References

[1] *Mobile Point of Scam: Attacking the Square Reader* - Alexandrea Mellen, John Moore, Artem Losev, Boston University, 2015

[2] *Square Updates its Credit Card Reader to Include Hardware Encryption* - http://www.theverge.com/2012/3/28/2909699/square-dongle-hardware-encryption

[3] *Android's Headphone Jack is Versatile, Yet Underused*

http://www.tested.com/tech/android/522-androids-headphone-jack-is-versatile-yet-underused/

[4] *iPhone Headphone Pin Mapping*

http://pinouts.ru/HeadsetsHeadphones/iphone_headphone_pinout.shtml

[5] *KeyEvent - Android API*

http://developer.android.com/reference/android/view/KeyEvent.html#KEYCODE_CALL

[6] *Wired Audio Headset Specification (v1.1)*

https://source.android.com/devices/accessories/headset/specification.html