Security Analysis of Smartphone Point-of-Sale Systems

WesLee Frisby *Benjamin Moench* *Benjamin Recht* *Thomas Ristenpart*

University of Wisconsin–Madison

{wfrisby,bsmoench}@wisc.edu, {recht,rist}@cs.wisc.edu

Abstract

We experimentally investigate the security of several smartphone point-of-sale (POS) systems that consist of a software application combined with an audio-jack magnetic stripe reader (AMSR). The latter is a small hardware dongle that reads magnetic stripes on payment cards, (sometimes) encrypts the sensitive card data, and transmits the result to the application. Our main technical result is a complete break of a feature-rich AMSR with encryption support. We show how an arbitrary application running on the phone can permanently disable the AMSR, extract the cryptographic keys it uses to protect cardholder data, or gain the privileged access needed to upload new firmware to it.

1 Introduction

The ubiquity of commodity smartphones has prompted companies to leverage them as a platform for replacing dedicated hardware computing devices. In this work, we consider the security implications of this trend for the case of smartphone-based point of sale (POS) devices.

A POS device is responsible for collecting, transmitting, and (sometimes) storing payment credentials in order to facilitate the sale of some good or service. While newer mechanisms for POS systems exist (e.g., chip-and-pin [7], near-field communication [9], and radio-frequency identification [10]), the predominant mechanism in North America [28] remains plastic cards with credentials encoded onto a magnetic-stripe. POS devices use a magnetic-stripe reader (MSR) to conveniently read the encoded data — typically a credit or debit card account number and some supporting account details.

While there are a plethora of POS solutions, the canonical contemporary system is an all-in-one standalone device with an MSR, number pad, small receipt printer, and network connection. (Traditionally the telephone system, though increasingly via the Internet.) These are reasonably expensive (e.g., hundreds of US dollars), and have significant security features.

Recent years have seen growth in alternate POS form factors. One such uses an inexpensive hardware dongle MSR plugged into the audio jack of a smartphone; we refer to this component as an audio-jack magnetic stripe reader (AMSR). A payment application, colloquially called an "app", is installed on the phone via an app store associated with the phone, e.g. the Apple App Store [6] or the Google Play Android market [8]. Collecting payment credentials proceeds by swiping a card through the AMSR, having the app receive data from the AMSR via the audio jack, and, finally, communicating this data over the Internet to a payment processing service.

In this paper, we relay our experience performing a security audit of a collection of smartphone POS systems. This collection includes four AMSRs and a larger set of apps (some AMSRs work with multiple apps). The four AMSR devices range in complexity and security features, from an analogue-only device up to microcontrollers that encrypt payment card data before transmission to the smartphone app. Our main technical result is a complete break of one of the latter AMSR systems, which implements an extensive firmware API accessible from the phone. We show how an arbitrary app running on the phone can: (1) disable the AMSR (brick it); (2) turn off encryption of card data; (3) extract secret keys used to encrypt cardholder data; and (4) gain the necessary credentials needed to upload new firmware to the AMSR. These all stem from a handful of basic software vulnerabilities in the firmware.

2 Background

We start with a brief overview of payment transactions, using as an example credit card transactions. A credit card number (CCN) is a 12-20 decimal string. Most frequently CCNs are 16 digits. The first 6 digits are the issuer identification number (IIN), which specifies the issuing bank. The remaining digits specify the primary account number (PAN). The last 4 digits of the PAN are treated as public values. Plastic credit cards encode data on three tracks on the magnetic stripe. Track one data includes the full CCN, the card holders name, expiration date, service code and verification identifiers. Track 2 contains the full CCN again, along with the expiration date, service code and the card verification value (CVV1) which should not be confused with the CVV2 which is embossed on the back of credit cards. Track 3 contains the full CCN number along with additional security codes and identifiers to assist the backend processor.

A card-in-hand transaction commences by gathering the CCN from the card. This either occurs via manual entry (typing the card into a keypad on a POS device) or via swiping of the card through an MSR. In most transactions, the CCN is transmitted immediately to an acquiring bank, with which the merchant has an account. The acquiring bank makes an approval decision, but may consult further upstream entities (e.g., the issuing bank). When such online approvals cannot be made, the CCN and other data may be stored until such communication can be made. Approval decisions will sometimes require further information such as the CVV2.

Security of payment transaction handling is dictated by an industry standard, the payment card industry data security standard (PCI-DSS) [2]. Relevant to this work, the standard requires that: track 2 data never be stored and that PANs must only be stored in encrypted form with proper key management practices.

A study [16] in 2008 estimated, that a CCN (without CVV2) was worth \$0.10 to \$25 USD on the black market. Adding the CVV2 number raises this estimate from \$0.12 to \$50 USD. As such, there is direct fiscal incentive to steal them, and numerous incidents have been reported upon in which criminals setup various kinds of card skimmers [31, 30, 22]. We use the term skimmer to refer to any mechanism setup at the POS that captures card data during a transaction.

3 Smartphone POS Systems

We focus on smartphone POS systems that combine an audio-jack magnetic stripe reader (AMSR) dongle with a software payment application, colloquially an "app", that runs on the smartphone. We focus on the Android platform, but are not aware of any protections on other platforms (e.g., the Apple iPhone) that would render our attacks impossible.

Payment Apps. Payment apps are smartphone applications which have been designed specifically to facilitate plastic card transactions. Card data can be entered manually via a software keyboard interface or through an AMSR which will be discussed shortly. Android payment applications are installed via the Internet using Google's app Play Store. The payment app must authenticate itself to an acquirer. This is done using merchant credentials, e.g., a PIN or password setup. These credentials are either entered each time the app is started (or unsuspended) or they are stored on the phone. While these would not necessarily be the primary target of an attacker, they are sensitive because they allow logging into the acquirer using the merchants account.

We note that the protocol between the phone and the acquirer is another potential weak link for security. For example, we note that rampant logic vulnerabilities have been found in similar contexts such as single sign on [26] and web cashier services [25]. However, we did not analyze these protocols.

AMSRs. An audio-jack magnetic stripe reader (AMSR) is a hardware dongle that assists payment applications with reading magnetic card data. The audio jack provides a two-way communication channel. Unlike other interface options, like the docking port of iPhones, AMSRs are an attractive design point due to the fact that audio-jacks are standardized across smartphones and provide sufficient power that an AMSR need not have an internal power source.

We investigated a range of AMSR design points. Passive AMSRs do not accept messages from the app, and instead only send data to it in response to a card swipe. Active AMSRs implement an API to which the app can make requests. Consequently, the latter form of AMSR can provide more versatile functionality, in particular allowing reconfiguring the AMSR without special equipment.

A key differentiator for AMSRs is whether they encrypt card data using embedded keys. A canonical choice for key management is the derived unique key per transaction (DUKPT) standard [11]. We discuss relevant aspects of DUKPT in Section 6.

4 Smartphone POS Threat Models

With the transition from standalone devices to smartphone-based POS systems comes a shift in security threats faced. We therefore provide an extensive discussion of the threat landscape in the context of AMSR systems, breaking them down in (roughly) order of adversarial access.

Network Adversaries. Wireless communications (WiFi, cellular data networks) are used by the POS app in order to transmit payment information, merchant credentials, and other sensitive data. A network adversary is one that can intercept or even modify communications to/from the app. In our analysis of the applications, we found that all applications were protected from a network adversary via use of TLS to encrypt card data.

Malicious Apps. While managed app stores, including Google's Play Store, monitor and attempt to remove malicious software, users are nevertheless often tricked into installing malware apps that make it past censors [20, 19]. Thus phones running POS systems could have malware apps installed. Here the permissions model of Android becomes relevant. At installation time, apps request a variety of permissions. For example, the RECORD_AUDIO permission enables reading from the audio-jack while the INTERNET permission enables Internet access for the app. We assume below that the malicious app has arranged for needed permissions (e.g., by tricking the user into giving them).

An attack vector here is spying on the communication channel between the AMSR and legitimate app. Here security has (seemingly) serindipitously benefited from the fact that Android allows only one app to read from the audio-jack at a time. Parallel access to the audio-jack —the typical case on PCs— would enable malicious apps to listen in on communications between the AMSR and the legitimate app. In the case of AMSRs without encryption this would already be sufficient to perform software skimming.

Another vector for attack is globally readable storage locations, such as the SDcard. A poorly written payment app might write sensitive data to such a location, that could then be read by a malicious app.

Android allows users to install and use custom software keyboard applications, which can then be set as the default to be used by other apps. If a payment app uses a default keyboard, and it proves to be malicious, then manually entered data will be subject to theft. Looking ahead, the payment applications we studied used their own built-in software keyboard for input of customer card data, but default keyboards for usernames, passwords, and session login pins.

A final vector for attack is side channels over shared hardware components. Recent smart phones include, for example, accelerometers, which have been reported to enable a malicious app to infer the keys pressed on the screen with 78% accuracy [13]. This could reveal manually entered card information.

Fake POS Apps. An embellishment on the prior threat is to trick users into installing a fake version of a legitimate app. This is facilitated by the fact that an attacker can download the legitimate app, modify it to contain malicious functionality, e.g., a card skimmer, and republish it on the app market under a new name. This would allow the fake app to have the look and feel of the legitimate one. Typical phishing techniques, such as making the name of the new app close to that of the legitimate one, may confuse users. Fake apps have already been found in app stores [12, 32].

This would give the app immediate access to any manually entered card data as well as swiped card data should the AMSR not support encryption. A complicating issue for the attacker arises should it attempt to ensure that transactions are successfully handled, lest the merchant become suspicious due to failed transactions. While we have not investigated existing payment protocols in detail, we suspect that fake POS apps would not have too much difficulty with this. For example, the ability of a processor to remotely attest to the veracity of the smartphone app is limited.

Malicious OS. Should the OS of the smartphone be compromised, the attacker would have access to all manually entered card data, card data from AMSRs that do not offer encryption, and possibly other sensitive information such as merchant credentials. Unfortunately, software vulnerabilities enabling root exploits have been discovered in the Android operating system [18, 21]. One vector for rooting the phone is via a privilege escalation attack by a malicious app, and indeed malware apps that include such attacks have been found in the app store [15].

Another vector by which OS compromise can be achieved is via physical access to the phone. A number of physical security vulnerabilities have been reported upon [29]. A specific threat is that a malicious recovery image can be flashed onto the phone and be executed in a manner that preserves user data. This holds true for many Android phones today (for example, our Samsung Galaxy S2), as we verified experimentally requiring only a USB cable.

Manufacturers are beginning to plug this vulnerability: flashing a new recovery image will require unlocking the bootloader, and this will erase existing data and apps. This, at least, provides some forward security, and increases detectability of compromise by users.

Malicious firmware. All but the simplest AMSR hardware devices are embedded systems running firmware that could be compromised. Vulnerabilities in the firmware code might be exploitable by a malicious app, leading to take control over the firmware, access to any cryptographic keys managed by the firmware, and the ability to access card data. Attackers in physical possession of the device could abuse AMSRs that do not lock their memory from external writes in order to install malicious firmware. This might erase old data, including cryptographic keys, and so the AMSR's continued functionality might depend on the attacker being able to install appropriate keys. In either case, the attacker would need a way in which to exfiltrate gathered data from the AMSR. This could be facilitated by cooperating with a malicious app or OS, or by setting up some kind of covert channel (c.f., [27]).

Malicious hardware. Hardware skimmers are a tried-and-true mechanism for gathering card data [31, 30, 22]. This works because magnetic cards do not store data in encrypted form. A motivated attacker could add malicious hardware to individual AMSR devices, or even the smartphone itself. The form factor of either could be a barrier to attackers, since it increases the difficulty of developing unnoticeable hardware skimmers. In the case of AMSRs with encryption, the malicious hardware would have to be inserted in a place that can access the bus between the microcontroller handling encryption and the magnetic card analog to digital converter.

Hardware skimmers could be inserted anywhere in the supply chain before it reaches the merchant or at some point when the device is left unattended.

Given that AMSRs are inexpensive, criminals may seek to mount replacement attacks, in which they swap out one AMSR for a malicious one. Because many AMSRs are commodity components with programmable firmware, the attacker could attempt to pre-program the AMSR firmware with both legitimate and malicious functionality. For AMSRs with encryption, completing transactions by such a trojan AMSR would require the attacker having access to legitimate cryptographic keys.

As with malicious firmware, a hardware skimmer would need some way to exfiltrate data. Cooperation with a malicious app or OS would make this work. An attacker might also insert a small wireless transmitter in order to transmit data to a nearby attacker.

Insider attacks. A significant fraction of credit card fraud arises due to insider attacks, namely those who are authorized users of the POS system. For example, in the United States, where it is customary for restaurant staff to handle card processing out of the view of the card owner, a common ploy is for staff to write down card details or double-charge the customer.

Such insider attacks might be made easier in the face of smartphone-based POS. AMSRs that do not support encryption could be used in conjunction with an easy-to-install app for skimming card data.

5 Analysis of POS Systems

We obtained four types of AMSR devices and a corpus of compatible POS apps. A summary appears in Figure 1. We will start our analysis with the apps and then discuss the security of the individual AMSRs.

AMSR	Passive	Encryption	Analyzed apps
Square v1	Yes	No	Square 2.2–2.2.5
Square v2	Yes	Yes	Square 2.2.5–2.2.7
Roamdata	Yes	Yes	Intuit GoPayment 2.5.0, PhoneSwipe 1.4, PayAnywhere 1.5
UniMag II	No	Yes	Intuit GoPayment 2.5.0,

Figure 1: Summary of systems studied.

5.1 App Analysis

We begun our investigations with an analysis of POS apps. As discussed in prior sections, the app is a security-critical component of smartphone POS systems. In terms of card data confidentiality, the app (and OS) must remain uncompromised in the case of AMSRs that do not support encryption. We begin with a bit of background, and then discuss our analysis of POS apps. Our investigations here are not exhaustive, but rather serve to form a baseline understanding of the security posture of these critical components of smartphone POS systems.

Background. Android applications run within Google's Dalvik virtual machine and all are given unique application user IDs (UIDs). Permissions are then handled by standard Linux access-control lists

(ACLs). Applications can execute native code outside the Dalvik VM, but permissions are handled in the same manner. Each application is provided with a private directory where information can be stored that cannot be read by other applications. System permissions are configured when the application is installed, in which case the user is presented with a list of permissions that the application has requested. It is up to the user to grant (install) or deny (not install) the application. Additional permission may be requested during an application update, but not dynamically while the application is running.

Software accessibility and analysis. Gaining access to smartphone POS apps and analyzing them is straightforward. The apps are freely available from app stores, and downloading them does not require a merchant account. A merchant account may be required to fully use the application but many applications contain a "demo" feature to test the application menus and swipe cards but do not actually allow someone to complete a transaction. The considerable number of development tools available for Android can then be brought to bear immediately for reverse engineering efforts.

We performed three areas of analysis related to the handling by apps of sensitive data such as customer and merchant credentials.

Card data storage and transmission. We used a combination of manual analysis and taint tracking by way of TaintDroid [14]. The latter enables real-time tracking of marked data items via identification of sensitive data sources and dangerous sinks (e.g., network calls). A custom set of sources and sinks were added to the TaintDroid framework for our purposes and the payment applications were then modified to hook in the TaintDroid add source functions at the locations card data enters the application. At the sinks TaintDroid monitored messages to detect potential leaks. One question we sought to answer was: Might apps accidentally leak card data by transmitting it unencrypted over the network or storing it unencrypted in locations on the phone that malicious apps might access it?

Fortunately, all apps analyzed used HTTPS for transmitting card data to processors. This is a testament to the benefit of Android's API support for TLS connections, in the form of the HttpsURLConnection API call.

In terms of storage in vulnerable places, we discovered that the Intuit GoPayment app stores RoamData AMSR messages on the SDcard. This is world readable, meaning that a malicious app could access it. Fortunately, the RoamData AMSR encrypts card data so this does not represent an immediate vulnerability. Nevertheless the app would do better to store such swipes in private memory. Some apps were also found to store encrypted card data within the application's private storage, which seems less damaging, but still might endanger forward security of card holder data should other exploitable vulnerabilities exist.

Handling of merchant credentials. Several of the apps used software-based encryption mechanisms in an attempt to protect merchant credentials such as login names and passwords. The PhoneSwipe, PayAnywhere, and Verifone apps all use the same, non-standard and insecure encryption scheme before storing data such as merchant login PINs, usernames, passwords, and non-sensitive portions of credit card numbers. The code for their encryption mechanism appears to have been copied from a contribution to the Android Snippets website by one Ferenc Hechler in 2009 [17].

This code uses the Andriod/Java crypto API [5]. First an AES key is generated using the Java crypto SHA-1 PRNG seeded with a password. Then an AES mode of operation is used to encrypt the data. The mode of operation is left unspecified. In all three apps, the password used was hardcoded in the app executable: "swipe" was used by two of the apps and "V3RIfon3" for the third. We verified that for the same app running on two different phones, the same AES key were generated.

Several apps used bcrypt [1] to hash merchant credential passwords for comparison with future logins. Like any password hashing scheme, an attacker that obtains these hashes, e.g. via compromising a stolen phone, would be able to mount offline brute-force attacks.

We note that some payment apps never stored any sensitive merchant credentials and instead used online logins every time the app booted up. For example, Square uses an HTTP session cookie.

Memory remanance. The payment apps consistently use the Java String class for storing card data. Use of the String class with sensitive data gives rise to various well-known security risks [4], in particular because the String class is immutable and consumer credentials cannot be zeroed out by the app when no longer needed. We verified the impact of this by taking memory snapshots after entering card data, waiting several minutes, and then killing the app with the Android debugger. Card data frequently remained in memory.

5.2 AMSR Analysis

We were able to obtain four different AMSRs at varying quantities by ordering them online. Each of the AMSRs were opened up to understand their hardware layout, identify commodity parts as a first step before further security analysis. Square V1 and RoamData reads only track 1 while Square V2 and the UniMag II reads tracks 1 and 2. We give more details about each AMSR in turn.

Square v1. The first generation Square AMSR is an analogue, passive device. It sends the included MSR's readings of a swipe directly, in analogue form, through the audio jack to the Square app. The data is decoded in software.

The lack of encryption on the AMSR renders Square vulnerable to attacks by a malicious OS, since the latter can likewise listen to the microphone audio channel. If it were not for the fact that Android disables parallel access to audio jack reads, a malicious app would likewise have been able to spy on transactions. The security implications of the lack of encryption by Square's AMSR was observed previously [3], though in the context of concern over criminals using AMSRs to record card details.

The Square v1 is a small device (2.5cm x 1.2cm), with a sealed plastic encasing. This design would make adding an unobservable hardware skimmer challenging.

Square v2. The Square V2 is a passive AMSR consisting of an analog magnetic strip reader, a battery, a TI MSP430G2412 microcontroller and an additional IC to amplify the audio output signal. We note that the JTAG fuse had been blown, which prevents hardware attacks via JTAG. Interestingly, the TI MSP430G2412 does not include basic bootstrap functionality, meaning that one cannot install new firmware onto the device. This prevents adversaries from easily loading malicious firmware onto a device.

The second generation AMSR provides on-device encryption, alleviating the security concerns surrounding the first generation device. This was the only AMSR to include a battery, and while this might seem to be a potential vector for denial of service (DoS) attacks, as far as we can tell the passive nature of the device ensures that significant battery drain only occurs during swipes.

The Square v2 is sealed with plastic encasing and is only slightly larger than the Square v1 at (2.5cm x 1.4cm). Inserting a unobserving hardware skimmer would still be challenging.

ROAM Data. The ROAM Data is a passive AMSR consisting of an analogue magnetic stripe reader, an analogue-to-digital converter (MagChip E424078J), and a PIC microcontroller (PIC16LF19360) that encrypts magnetic card data. PIC microcontrollers are known to protect code reads by using a code protect (CP) bit set to one during programming. Earlier versions of the PIC microcontroller allowed selective portions of memory [23] to be replaced even when the the CP bit is set to one. This would allow a hardware attacker to load a small program into the firmware's memory that could dump the rest of the firmware code, the firmware memory (including cryptographic keys), and setup a firmware skimmer.

Fortunately, the PIC microcontroller used in the ROAM Data AMSR protects against read access to the firmware. When the CP bit is set, all firmware must be erased before the flash may be reprogrammed. This does not disable reprogramming, and so an attacker could still load new firmware. But in doing so the chip will erase its memory contents so existing sensitive data is lost.

The device has no anti-tamper protection and has sufficient space that a well-designed, small hardware skimmer could be hidden inside the AMSR.

UniMag II. The IDTech UniMag II is an active AMSR with an analog magnetic strip reader, an IC to convert the analog magnetic card data to digital signals, and a TI MSP430FR5728 microcontroller to encrypt the card data. The UniMag II AMSRs we analyzed were the version distributed by Intuit as part of their Intuit GoPayment product. The JTAG fuse on the microcontroller was blown, disallowing this vector for hardware attacks. The UniMag II card reader is a commodity component used within several mobile payment products. The AMSR is therefore designed to be highly configurable. The TI chip includes a special bootloader mode which is accessible from a smartphone via the audio jack. This would support firmware updates in the field. Using the bootloader mode requires authentication, as discussed more in the next section.

Like the ROAM Data, the lack of anti-tamper protection and the form factor of the AMSR means that a hardware skimmer could be hidden within. Also like the ROAM Data, this AMSR encrypts card swipes using DUKPT. In theory this should prevent a malicious OS or app from compromising user data.

We will show in the next section that this is not the case. The firmware implements a rich API that provides smartphone-based software attackers with a large surface area to explore. Manual inspection of the binary of one of the apps that uses the UniMag II AMSR revealed a portion of the API. Further information was found within some public documentation

for a similar device on the manufacturer's website. (Detailed documentation of this AMSR was not publicly available.) Several example commands are included in Figure 2. In the next section we will report on successful exploitation of these API commands in order to completely compromise the AMSR.

6 Vulnerabilities in the UniMag II

We did not have access to the firmware used by the UniMag II. To discover vulnerabilities, then, we built a bare-bones smartphone app that leveraged the IDTech SDK (contained within the Intuit GoPayment application) to enable issuing commands of our choosing to the AMSR.

Lack of command authentication. For all commands but the one used to enter bootloader mode, the AMSR did not perform authentication of the source of the commands. On Android an arbitrary app can play audio, which is the same as sending a message to the audio jack. One app can write to the audio jack at a time on Android (other phones are different: the iPhone allows multiple apps to access the audio jack in parallel). The lack of authentication is therefore already a serious vulnerability, as any malicious app can send commands to the device whenever the legitimate POS application is not sending commands.

This already enables some attacks, such as a DoS against an unsuspecting device. For example, one seemingly benign command built into the API, but which is not used by the payment apps we investigated, can select which of the 3DES or AES block ciphers to use during encryption. Since the app expects 3DES, if the encryption mode is changed to AES, a merchant would not be able to process future card swipes.

Using well-formed API commands in a malicious manner, however, does not enable a full break of the system, meaning the ability to skim card data or, worse, extract the cryptographic keys used to encrypt card data. We therefore fuzz-tested the API to search for software vulnerabilities in the firmware's implementation. To start with, we built a mutation-based fuzzing protocol to automatically search for inputs that would trip up bugs in the software. This proved to be too slow to be of immediate help: the round-trip response time for each command sent to the AMSR was around 2 seconds. In order to exhaustively search a four byte command at this rate would have taken 272 years.

To expedite testing, we targeted commands that had variable lengths. Some of the commands we explored are summarized in Figure 2.

SetPrePAN command. We began with a command documented in the public manual and used in the app's SDK, the SetPrePAN command. When a swipe occurs, the AMSR sends both an encryption of the track 1 and 2 data and also the track 1 and 2 data in the clear, but with sensitive portions masked out with (replaced by) asterisks.

Recall that the first six digits of the PAN, the IIN, identifies where the card was issued and the card brand. The documentation states that the number of the first PAN digits returned in the clear could be any value less than or equal to six. The SetPrePAN command sets the value. The last four digits of the PAN are also sent in the clear, and the rest are masked out. The first row in the table of Figure 3 shows the returned value (modulo the ciphertext) after a swipe using a test card and a PrePAN value of six.

The command has two arguments: a one byte length value and a one byte value indicating the number of characters to leave unmasked. Curious to determine if the firmware was interpreting the length byte as a signed or unsigned value, we set the byte to 0x80 which would be -128 signed or 128 unsigned. The AMSR responded with an ACK saying that the command succeeded. When we swiped our testing card we retrieved the data shown in Figure 3. All of Track 2 was unmasked. Compare this with the above swipe, where the correct number of digits were unmasked.

Subsequent swipes caused the AMSR to reboot and send out the initial powered-on message. The AMSR would, however, still respond to commands. A possible explanation for this behavior is that the PrePAN masking function uses the SetPrePAN length parameter as an offset into the track data buffer. With the length parameter set to 0x80, it is outside the bounds of that buffer, and forces the PrePAN function to write "*" characters over some other portion of memory, in turn causing the unpredictable behavior.

This signed integer vulnerability here allows a limited attack against card data confidentiality and also a DoS attack that renders the AMSR unusable.

SetPreamble. With the initial success of writing into the memory of the AMSR, we searched for other commands with length parameters that we might modify. Unfortunately, no command beside SetPrePan seemed to allow us to write into an arbitrary memory location. The ReviewSettings command proved helpful. It returned a sequence of byte strings that are, in fact, a list of settings from which one can

Command	Description	Vulnerability	Implications
Review device serial number	Returns the device's serial number	n/a	Leaks serial number
Review KSN	Returns the devices current key serial number	n/a	Leaks KSN
Review Settings	The review settings command returned the settings for a number of commands that could be controlled by application.	–	Reveals undocumented commands
SetPrePAN	Controls the number of the beginning account number digits that are not masked, up to a max of 6 digits.	Signed integer logic vulnerability	PAN sent in the clear ; DoS
SetPreamble	An undocumented command that sets the preamble header appended to each message sent by the AMSR to the phone. Used by the SDK to synchronize decoding of the waveform.	No bounds checking	Device key recovery
Enter bootloader mode	Command uses a challenge-response mechanism to enter into bootload mode and update the AMSR's firmware.	n/a	Firmware may be replaced

Figure 2: Example IDTech SDK commands, inferred software vulnerabilities in their implementation, and implications.

SetPrePAN Command (**length**)	Result

02 53 49 01 **06** 03	`Track 1`
	`601056******5410^CHIPOTLE/VL^2501**********?*;`
	`Track 2`
	`601056******5410=2501**********?*`

02 53 49 01 **80** 03	`Track 1`
	`6010568266865410^CHIPOTLE/VL^2501**********?*;`
	`Track 2`
	`6010568266865410=250100040000600578 87?8`

Figure 3: Unencrypted data returned in response to a swipe for different SetPrePAN values.

infer sufficient information to reconstruct the commands used to change the settings. So, for example, portions of the SetPrePAN command were returned as an entry. All of these returned strings matched documented commands except one, for which the byte string, in hex, was

`D1 10 0F 55 55` $\cdots$ `55 66`

and where there are 14 repetitions of the byte value 0x55. The byte 0x10 (as in other API commands) specifies the length of the argument. The argument, in turn, starts with a length given as a byte, the 0x0F byte above, followed by a string of that length. With further experimentation, we realized this is a command we will call SetPreamble, because the string specified in it (0x55$\cdots$0x66) above is the preamble for all subsequent messages from the AMSR to the phone.

For an initial test we set the length bytes to 0xFF and 0xFE and sent in 255 bytes of machine code that, if executed with proper alignment, would execute a jump to the lowest address of the code. This would test whether we could perform a trivial code injection attack.

Upon submitting this command, however, the device never responded. The device, furthermore, no longer responded to any control commands from the phone. That is, it was bricked. We believe that we overwrote crucial memory in the device preventing any further communication with the device. This therefore gave a more drastic DoS attack than that offered by the SetPrePAN (which disabled swipes but not, seemingly, the firmware's other functionality). We reiterate that any app can write to the audio port and so brick the device.

On a fresh device, we explored further this bounds checking vulnerability. We issued SetPreamble but with length bytes 0x11 and 0x10, but only a following string of length 0x0F. When we re-executed the Review Settings command an extra byte was re-

turned from the memory in the card reader, and the same extra byte was present in the preamble. This suggested that the extra byte was leftover in a buffer from a previously submitted command.

We iterated this approach to find the maximum lengths we could indicate that would not overflow the buffer and cause the device to stop functioning, while minimizing the length of the actual preamble. This then maximized the gap between indicated and actual lengths. This lead us to the following Set-Preamble command

```
D1 3C 3B 55 55 55 55 55 55 55 55 66
```

Executing Review Settings after this command exposed most of a previously executed command and, strangely, the entire response for that command. It is unclear why this command's response was found immediately after it in the buffer; these bytes were not overwritten when new responses were sent by the device. On another fresh device, this previous command turned out to be the Key Load command as indicated by the response code found in the buffer. This command is used to provision the device with the initial key used with the derived unique key per transaction (DUKPT) protocol. This indicates that the command buffer being used here is on the non-volatile flash memory, since otherwise it would have been lost after powering off.

DUKPT and Key Recovery. The DUKPT standard specifies a way in which to use a block cipher as a pseudorandom number generator in order to safely derive new encryption keys for each transaction. It is used frequently in the banking sector. This AMSR supported use of DUKPT with either AES or 3DES, the latter was used by our devices. Keying option two of 3DES is used [24], meaning two 56-bit keys K_1 and K_2 are used to encipher a 64-bit plaintext M via $\mathsf{DES}(K_1, \mathsf{DES}^{-1}(K_2, \mathsf{DES}(K_1, M))$. Decryption of cyphertext C is simply the reverse: $\mathsf{DES}^{-1}(K_1, \mathsf{DES}(K_2, \mathsf{DES}^{-1}(K_1, C))$.

The key loading command loads a 128-bit key that can be used for either AES or 3DES. The key is encoded as ASCII encodings of hex digits, so each byte encodes four bits of key material. In the case for 3DES, the least significant bit of each byte is used only within the DUKPT key derivations, and the remaining 112 bits are used as the keys K_1 and K_2. In the response to the Load Key command is a key check value: the first three bytes of ciphertext resulting from evaluating 3DES using the just-loaded key and the all zeros message.

The memory leaked by the combination of Set-Preamble and Review Settings commands reveals 15.5 bytes of the 16 byte initial DUKPT key along with the three byte key check value. The remaining four bits of the key are not revealed because the first byte of key material is overwritten by the Set-Preamble string argument. (Each key byte is hex encoded, so losing one byte of the encoding loses four bits of the key.) The key check value provides a plaintext-ciphertext pair. Other API commands conveniently provided the key serial number and device serial number; the message sent in response to a swipe also contains these values. At this stage we have everything needed to brute force the remaining four bits of the initial DUKPT key, and from there derive any past or future key used by the AMSR.

To confirm this, we generated new keys from the initial DUKPT key we retrieved from the device. We then swiped our card, matched the key serial number sent in the swipe to one of the generated keys, and decrypted the first track with 3DES in CBC mode using the first 8 bytes of the encrypted data as the IV. The second track was encrypted in the same manner with a new 8 byte IV from the data. The decrypted card data matched our test card:

```
8266865410^CHIPOTLE/VL^25010004000060057887
266865410=250100040000600578 87?8
```

Entering bootloader mode. As a final security issue, we investigated the possibility of loading new firmware onto the device. Recall that the Enter Bootloader command is the only API command that requires authentication from the phone. In the authentication request message, a key serial number is indicated; this matched the one just extracted from the device. Thus, the next derived DUKPT key also enables administrative access to update the firmware. We were able to upload new firmware to the device in blocks; after each block uploaded the device responded affirmatively with the ACK response. However, the firmware did not run properly. The device requires it to be in a specific format that is not published in the specifications. With further effort and knowledge (i.e., access to a copy of the current firmware of the AMSR), we suspect that an attacker would be able to craft a runnable malicious firmware.

Summary. In summary, we found several glaring vulnerabilities in this widely used AMSR:

1. API commands are not authenticated, so arbitrary applications can send commands (the audio port is world writable on Android).

2. The implementation of the only documented API command with a length value as a param-

eter suffered from a signed integer vulnerability allowing one to turn off masking of the PAN and also disable the device from being used for future transactions.

3. The implementation of an undocumented API command contained a bounds checking vulnerability that enables an arbitrary application on the device to either brick the device, or extract information sufficient to decrypt all past and future card data encrypted by the device.

4. An arbitrary application on the phone can, using the extracted key material, enter privileged mode on the device. This may enable malicious firmware to be installed on the AMSR.

7 Discussion: the Verifone vx670

As a point of comparison, we also performed a cursory investigation of a Verifone vx670 all-in-one standalone POS device. The contrast in terms of security engineering could not be starker. The vx670 has several layers of tamper-response countermeasures. Tamper-response can render more expensive the types of hardware attacks seen in the wild [31, 30, 22]. The AMSRs we investigated have no specific hardware attack countermeasures: instead, they appear to rely upon the small form factor to prevent use of hardware skimmers. The vx670's payment software is difficult to obtain from a device (we were able to extract a portion of it via a difficult-to-access open JTAG port). This slows down, but does not prevent, analysis of the software for vulnerabilities. In comparison, the apps of smartphone POS devices were trivial to download and reverse engineer. The vx670 uses public-key cryptography; all payment applications we looked at used either no cryptography, poor software cryptography within apps, or symmetric encryption on the AMSR to protect card data and other credentials. The only AMSR that implemented a rich firmware API fell over easily and repeatedly in the face of a malicious app running on the smartphone.

Our experiences provide anecdotal evidence of a marked decrease in security precautions with the newer smartphone POS solutions. Our cursory inspection certainly does not rule out vulnerabilities in the vx670 (let alone other standalone POS devices), but does show that new smartphone POS systems represent lower hanging fruit for attackers.

8 Conclusion

The ubiquity of smartphones has lead to widespread deployment of smartphone-based POS systems. This represents a significant architectural shift from prior commodity standalone POS devices, and one that seems to have, in notable cases, left security engineering struggling to keep up.

Our analysis suggests some general issues underlying the (in)security of smartphone POS systems. First, the audio port is being used in unenvisioned, security-critical ways. Some of the POS systems appear to (at least implicitly) consider it to be a secure channel, even though there is no fine-grained access control offered by the smartphone OS. Second, passive AMSRs (that include encryption) proved, at least in our investigations, a significantly less vulnerable design point compared to active AMSRs. Future work might clarify whether there is a need for active AMSRs in this context at all, or if there are deployable architectures that will encourage improved security engineering for active AMSRs. Third, the proprietary nature of the payment industry stands in the way of proper security auditing. For example, the vulnerabilities we found would likely be easily caught via a source-code review.

Finally we note that the issues uncovered here are unlikely to be unique to the payment domain. The move to use mobile phones as a platform for hosting special-purpose embedded devices will arise in other settings, and there may be overarching security engineering principles that will apply equally well to POS systems and beyond.

Disclosures and vendor response. The most severe vulnerabilities (discussed in Section 6) we uncovered were reported to the companies involved (Intuit and IDTech), and with approximately 90 days of lead time before public disclosure would occur. Both companies responded quickly; worked together to develop a firmware patch; allowed us to verify the patch prevented all the attacks we had uncovered; and have scheduled an upgrade to occur before public release of this paper. We note that this was only possible because of the active nature of the AMSR. We thank the employees of the companies for their professionalism and for their successful efforts to improve their customers' security.

References

[1] bcrypt, 2002. `http://bcrypt.sourceforge.net/`.

[2] PCI Data Security Standard v 2.0, 2010. https://www.pcisecuritystandards.org/security_standards/documents.php.

[3] An Open Letter to the Industry and Consumers, 2011. www.sq-skim.com.

[4] Java cryptography architecture reference guide, 2011. http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html#PBEEx.

[5] Android javax.crypto API, 2012. http://developer.android.com/reference/javax/crypto/package-summary.html.

[6] Apple App Store, 2012. http://www.apple.com/itunes.

[7] Chip and PIN, 2012. Homepage: http://www.chipandpin.co.uk/.

[8] Google Play, 2012. https://play.google.com/store.

[9] NFC Forum, 2012. http://www.nfc-forum.org/.

[10] RFID, 2012. Homepage: http://www.rfid.org/.

[11] ANSI X9.24-1992. Financial Services Retail Key Management, 1992.

[12] T. Armstrong. Stealing apps, installing ads, 2011. http://www.securelist.com/en/blog/208193251/Stealing_apps_installing_ads.

[13] L. Cai and H. Chen. TouchLogger: inferring keystrokes on touch screen from smartphone motion. In *Proceedings of the 6th USENIX conference on Hot topics in security (HotSec'11). USENIX Association, Berkeley, CA, USA*, pages 9–9, 2011.

[14] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6. USENIX Association, 2010.

[15] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[16] M. Fossi. Symantec report on the underground economy, 2008. Symantec Corporation.

[17] F. Hechler. Encrypt/decrypt strings, 2009. http://www.androidsnippets.com/encryptdecrypt-strings.

[18] X. Jiang. GingerMaster: First Android Malware Utilizing a Root Exploit on Android 2.3 (Gingerbread), 2011. http://www.csc.ncsu.edu/faculty/jiang/GingerMaster.

[19] X. Jiang. Security Alert: New Android SMS Trojan – YZHCSMS – Found in Official Android Market and Alternative Markets, 2011. http://www.csc.ncsu.edu/faculty/jiang/YZHCSMS/.

[20] X. Jiang. Security Alert: New Stealthy Android Spyware – Plankton – Found in Official Android Market, 2011. http://www.csc.ncsu.edu/faculty/jiang/Plankton/.

[21] X. Jiang. Security Alert: New RootSmart Android Malware Utilizes the GingerBreak Root Exploit, 2012. http://www.csc.ncsu.edu/faculty/jiang/RootSmart.

[22] Kevin McCallum. Losses mount from skimming scam at Lucky store in Petaluma, 2011. http://www.pressdemocrat.com/article/20111205/ARTICLES/111209763/1350?Title=Losses-mount-from-skimming-scam-at-Lucky-store-in-Petaluma.

[23] M. Meriac and H. Plötz. Analyzing a modern cryptographic RFID system, 2010. Presentation at the 27th Chaos Computer Congress.

[24] National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. Oct. 1999. supersedes FIPS 46-2.

[25] R. Wang and S. Chen and X. Wang and S. Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 2011.

[26] Rui Wang and Shuo Chen and XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[27] G. Shah, A. Molina, and M. Blaze. Keyboards and covert channels. In *Proceedings of the 15th conference on USENIX Security Symposium*, volume 15, 2006.

[28] Smart Card Alliance. Card Payments Roadmap in the United States: How Will EMV Impact the Future Payments Infrastructure?, 2011. `http://www.smartcardalliance.org/resources/pdf/Payments_Roadmap_in_the_US_020111.pdf`.

[29] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pages 10–10. USENIX Association, 2011.

[30] Waterloo Region Record. Police warn against newest scam, 2009. `http://www.sparkfun.com/tutorial/news/SparkFun-PINScam.pdf`.

[31] J. Winters. Credit card skimmers discovered inside local gasoline pumps, 2012. `http://www.times-herald.com/Local/Credit-card-skimmers-discovered-in-gasoline-pumps-2119062`.

[32] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of 2nd ACM Conference on Data and Application Security and Privacy (CODASPY 2012)*, 2012.