

Why Banker Bob (still) Can't Get TLS Right: A Security Analysis of TLS in Leading UK Banking Apps

Tom Chothia, Flavio D. Garcia, Chris Heppel, and Chris McMahon Stone

School of Computer Science,
University of Birmingham,
United Kingdom

Abstract. This paper presents a security review of the mobile apps provided by the UK's leading banks; we focus on the connections the apps make, and the way in which TLS is used. We apply existing TLS testing methods to the apps which only find errors in legacy apps. We then go on to look at extensions of these methods and find five of the apps have serious vulnerabilities. In particular, we find an app that pins a TLS root CA certificate, but do not verify the hostname. In this case, the use of certificate pinning means that all existing test methods would miss detecting the hostname verification flaw. We also find one app that doesn't check the certificate hostname, but bypasses proxy settings, resulting in failed detection by pentesting tools. We find that three apps load adverts over insecure connections, which could be exploited for in-app phishing attacks. Some of the apps used the users' PIN as authentication, for which PCI guidelines require extra security, so these apps use an additional cryptographic protocol; we study the underlying protocol of one banking app in detail and show that it provides little additional protection, meaning that an active man-in-the-middle attacker can retrieve the user's credentials, login to the bank and perform every operation the legitimate user could.

1 Introduction

The use of TLS in smartphone apps has proved challenging for developers to get right. Common mistakes involve accepting self signed certificates, not checking the hostname, accepting weak cipher suites and allowing SSL stripping due to sending HTTPS links over insecure connections [4,9,12]. These issues are all trivial to fix and easy to detect so we would expect that leading international banks would not make such mistakes.

Banking organisations have previously threatened legal action against security researchers [6,18] and the UK courts have granted temporary injunctions against researchers on the grounds that they could not show that they followed proper procedure in their analysis [5]. Therefore, when carrying out our work it would be useful to be able to follow all of the terms and conditions of use of the apps, failure to do so could lead to legal pressure not to publish the results of the analysis. In particular, the terms and conditions of the banking apps and app store, forbid reverse engineering the code, and accessing app stores outside of your geographical area. We note that our analysis method does not require this and allows analysis without breaking the terms and conditions of the app.

A correctly configured TLS client should check (among other things) that the server it communicates with uses a certificate for the hostname the client is expecting, and that this certificate is signed by someone the client trusts. Additionally, it is good practice for the client to check that the certificate is signed by a single, prearranged Certificate Authority (CA), rather than an arbitrary CA from the list of those trusted by the client's OS. This additional check is known as certificate pinning. There are a number of options as to which certificate in the chain is being pinned to: pinning the server certificate provides the highest security level but it also provides less flexibility, e.g., when this key expires or needs to be revoked. Alternatively, it is possible to pin the intermediate or even the root CA. The higher in the chain you pin, the more flexibility you get at the cost of security.

In this paper we report on the analysis of Android and iOS apps from 15 of the leading retail banks based in the UK (see Table 1). We test all of the apps for the most common TLS flaws, i.e. whether they accept self-signed certificates; correctly check the hostname; permit protocol downgrades or weak cipher suites; allow SSL stripping (by sending secure links over insecure connections); Using available tools and previous defined methods we found no vulnerabilities in any of current versions of the apps.

However, we note that as one of our test devices was an old Apple tablet that did not support the most recent version of iOS. We found that when apps were downloaded on these older devices, an older legacy version of the app was provided. In fact, two of the legacy apps accepted self-signed certificates, and using this an attacker could steal the initial setup code used to authenticate the app, which is generated during the registration process. It seems likely that this was a flaw in the application that had been previously detected and fixed, but fixing the legacy version of the app had been overlooked.

While checking the up-to-date apps, we found that 7 of the 15 banks we tested used certificate pinning, as we might expect from high security apps. We note that the method of checking that apps are verifying server hostnames used by others [9] is unreliable when certificate pinning is being used. This is because by proxying TLS connections with a trusted certificate for an unrelated hostname, one cannot distinguish whether the app rejects the connection because the hostname is invalid, or because a chain of trust cannot be established due to pinning being in use.

While pinning to the server public key would be secure, to test for apps that pinned to higher up the certificate chain, we obtained a certificates from corresponding root CAs but for our own hostnames. We found two apps that accepted such certificates Co-op and Natwest bank, meaning that these apps could be MITMed and were not secure. While pinning the certificates is good security practice, it seems possible that in the case of Coops app, pinning the certificate hid the more serious problem of having no hostname verification. Natwest on the other hand was not pinning, but implemented proxy bypassing measures. This additional security measure also hid the problem of no hostname verification.

The banks involved all have rigorous security testing regimes, and during the disclosure process they told us that they had also hired leading outside penetration companies to test the apps; however the certificate pinning would have meant that many of the standard penetration tests on apps (such as trying to MITM with a valid certificate for the same hostname) would have failed. While high security apps should













App Name	Vulnerabilities			
	Accepts any certificate (self-signed or any hostname)	Pinning or proxy bypass without hostname validation	In-app phishing	Broken second layer protocol
Natwest	Legacy 		X	X
Lloyds	X	X	X	X
RBS	Legacy 	X	X	X
Barclays	X	X	X	X
Co-Op	X		X	 
HSBC	X	X	X	X
Nationwide	X	X	X	X
Santander UK	X	X	 	X
TSB	X	X	X	X
Halifax	X	X	X	X
Metro	X	X	X	X
Tesco	X	X	X	X
Clydesdale	X	X	X	X
First Trust	X	X	 	X
Allied Irish (GB)	X	X	 	X

Table 1: Apps in our test set and vulnerabilities discovered.

certainly continue to pin the certificates, care must be taken to ensure that this does not mean that other vulnerabilities are missed.

When examining the apps we also found that some apps which passed all of the TLS checks, requested some of their data over unencrypted connections. While not a vulnerability in itself, a few of the apps were requesting images with links to be displayed to the user. We found this vulnerability in apps from Allied Irish Bank and Santander UK, among others. We additionally found unencrypted update checks which could be used to make the app tell the user that they needed to upgrade and redirect them to a site of the attacker’s choosing, e.g. a phishing website.

When examining apps with failed TLS, we discovered that some apps use an additional layer of security. Requirement 1 of the PCI PIN security regulations [15] states that *PINs must never appear in the clear outside of an SCD*. If this is applied to mobile banking application PINs, it would make a second layer protocol compulsory. Web servers that decrypt TLS traffic from these apps are unlikely to be Secure Cryptographic Devices (SCD). Therefore there is a need to implement a second layer secure protocol so that PIN data can be forwarded to the banks’ SCDs for processing. Further to this, new attacks have frequently been discovered against TLS so an additional layer of security is a sensible precaution. We found that some of the apps did use an additional cryptographic protocol on top of TLS, however we show that this protocol is flawed, allowing an adversary to obtain the information needed to log into the victim’s online banking. We go on to propose an alternative protocol that would keep mobile banking secure even when the TLS protection failed and we formally verify this protocol.

Summary of our contribution:

- Carrying out a manual analysis of the 15 leading UK banking apps and finding that they are not vulnerable to TLS flaws previously reported in the literature.
- Discovering a new vulnerability that can arise when certificate pinning is used incorrectly. We have manually tested for this vulnerability and found it in one of the apps.
- Identifying “in app phishing” as an issue for apps which establish secure TLS connections but request resources, such as images and links, via an insecure connection. We found this vulnerability in three iOS banking apps and their Android counterparts.
- Identifying the existence of vulnerable legacy versions of apps as being an issue for the Apple App Store, and we have found two vulnerable apps from UK banks that were still being made available to customers.
- Studying the cryptographic protocol used by one of the banks as a second layer of security. We found that this protocol is flawed and that an attacker can obtain the credentials needed to take control of a users’ Internet banking.
- We propose an improved protocol which addresses this problem and formally verified it using ProVerif.

A summary of the issues found is given in Table 1. For two Android banking apps, we are able to mount a man-in-the-middle attack that is able to retrieve the user’s credentials (username and PIN/password), login to the bank and do every operation the legitimate user can i.e., see the balance and past transactions, make and modify bank transfers, etc. For three more iOS and Android banking apps, we could inject our own content and links into the apps, and use this to phish log-in details, and for two legacy iOS apps we found that we could MITM the TLS connection and so eavesdrop on the initial setup process of the app. These attacks are very practical. All an attacker needs to do is to create a “free WiFi” hotspot at a popular coffee shop and run a script that waits for a user to use their banking app, then MITMs the connection and collects the security tokens and user PIN numbers, which the attackers could then use at a later date. Through a lengthy disclosure process we have informed all of the banks involved and all but one have now fixed the problem.

Structure of the paper

In the next section we discuss related work and then we provide some background, briefly describing the TLS protocol and certificate pinning. Section 3 describes the apps we tested and the approach we took to our security assessment. Section 4 presents a novel vulnerability that arises if certificate pinning is used incorrectly. In Section 5 we perform an in depth analysis of an additional protocol used by one of the apps, and show that it provides little extra security. Section 6 looks at banking apps which are vulnerable because they load some of their content over non-TLS connections, and we give examples of in app phishing attacks against banking apps from Santander and Allied Irish Bank. We present our improved secure banking protocol in Section 7 and we conclude in Section 8.

2 Background and Related Work

2.1 The TLS Protocol

Transport Layer Security (TLS) is a cryptographic protocol designed to provide confidentiality, integrity and server authentication (with optional client authentication) to application layer network traffic. To establish a secure connection between a client and a server, the server must present the client with its public key. This key is usually encapsulated in an X.509 certificate which also contains other information required to provide assurance of the authenticity of the key. A certificate includes a CommonName field and SubjectAltName set, which tie the key to specified hostname(s); a cryptographic signature provided by a CA, usually an RSA encrypted SHA-256 value; and a valid from date and expiry date. Additional meta-data is also present (see RFC 5280).

In order to be sure that a public key belongs to a given server, client applications trust a predefined set of root certificates belonging to trusted certificate authorities. These root CA certificates are carefully selected by the developers of the OS e.g. Android or iOS, and come pre-installed in the OS. This trust store is then updated as required by pushing updates to the OS, but can also be modified by the user.

When attempting to set up a TLS connection, the server will provide a chain of certificates so that the user can build up a chain of trust to one of the trusted root CAs. Such chains typically have a length of three: the server or leaf certificate; which is signed by the issuer or intermediate CA which is itself signed by a trusted root CA.

To validate a TLS certificate, the client makes the following default checks:

- Validate that each certificate in the chain is signed by the previous one, starting with the leaf certificate.
- The final certificate in the chain is that of a trusted CA.
- The requested hostname matches the certificate’s CommonName value or is contained within the SubjectAltName set.
- The current date is within the certificate’s valid date range.

Further verifications should also be carried out such as checking the revocation status of the certificates. However, these are often omitted.

2.2 Certificate Pinning

The trust model that TLS adopts is arguably its biggest weakness. If a single Certificate Authority is compromised, then valid certificates for any hostname can be produced by signing them with the compromised CA root certificate. This would enable an attacker to MITM any TLS connection.

Additionally, an individual user’s trust store could be compromised. There is the potential for an attacker to trick a user into adding a custom root certificate to their trust store, for example by suggesting it is a requirement to install a free app. Similarly, but more alarmingly, [19] showed how malicious apps on rooted Android handsets can modify the trust store totally unbeknown to the user. Previous work has also exposed the how bloated trust stores are and identified many root certificates that are unused [16]. Removing these would reduce the attack surface associated with CA compromise, however the risk still remains.

In order to mitigate these risks, Evans et al. proposed a technique named certificate pinning [8]. Applicable in situations where the hostname of the server is known in advance, like connections made by a mobile application, the certificate or public key that the app expects to see can be fixed. This aims at restricting the trust from all valid certificate chains originating from root CAs in the trust store, to a specific public key certificate, or ones derived from a particular fixed certificate. In general there are two ways this can be implemented:

- **Leaf certificate or Public Key** - Pin the servers specific public key certificate which is usually achieved by hard coding it's fingerprint (typically a SHA-256 value). Alternatively, just the server's public key can be pinned. The downside of this type of pinning is reduced flexibility, as the certificate may expire or the use of a static key may violate key rotation policies. If the certificate changes, users will be forced to update the application to continue use.
- **Intermediary or Root certificate** - A specific root CA or intermediary certificate can be pinned. The server can then re-new its leaf certificate whenever needed as long as it is signed by the pinned root or intermediary certificate.

2.3 Related Work

Fahl et al. [9] carried out a large scale analysis of Android apps in 2012 and found widespread misuse of TLS, in particular they found that many apps accepted self-signed certificates, did not check the hostname they connected too and did not encrypt some connections at all. Their testing methods would not find the class of vulnerabilities we are dealing with here, as the use of certificate pinning on its own would have been enough to pass their automated MITM tests. We note that they did not find any flaws in banking apps.

Georgiev et al. [10] carried out a similar analysis for application and library code and find similar issues, some of the same authors go on to develop a testing tool [4] which randomly mutates certificates to look for errors, again this approach would miss the vulnerabilities we present here, as the test would lack the certificate required by the pinning.

Reaves et al. [17] reverse engineered banking apps from developing countries and found a wide range of security issues. We note that while this work did look at banking apps, the banks involved were not large banks that had invested heavily in security, and all of the weaknesses could have been found by a competent penetration testing company. Their analysis involved decompiling the app code to look for vulnerabilities. This approach might find the code which allowed any hostname however it requires considerable time and effort to carry out such an analysis and so does not scale. Such an analysis also breaks the terms and conditions of the app, which as we mention above is something our analysis method does not do.

Oltrogge et al. [14] performed an extensive study on the applicability of certificate pinning in mobile apps. A classification method to establish whether an app would benefit from the use of pinning was applied to over 600,000 Android apps. Developer feedback was collected from a number of respondents and found that only a quarter of them grasped the concept of pinning and yet still found it too complex to use. The

pinning vulnerability presented in this paper is therefore perhaps directly a result of the complexity of pinning implementations.

Google has developed a testing tool called “nogotofail”¹ and there are a range of similar online checking tools which carry out tests on TLS configurations² but again, none of these can detect the lack of hostname verification if certificate pinning is used.

3 Testing Apps

Test set For test cases, this paper focuses on mobile apps from large, leading retail banks. These types of apps have all the necessary ingredients for a good case study:

- They use TLS.
- They are security critical applications, with well motivated attackers, and large user bases.
- They often use techniques to provide additional security such as certificate pinning and two-factor authentication.
- Unlike many other apps, they have been subject to thorough penetration testing, so avoid basic vulnerabilities.

Additionally, customers of high-street banks are increasingly making use of mobile apps to manage their finances. A report by the British Bankers Association [1] found that in 2015 there were 40,000 downloads and 11M logins to UK banking apps per day. These usage statistics and the sensitive nature of the data that is managed by these apps, highlights the need to carry out a thorough review of their security.

We chose the top 15 consumer banks that are based in Great Britain or Northern Ireland. These are all listed in table 1.

Approach The first stage of our investigation was to look for well-known TLS vulnerabilities and misuses that have been brought up in past literature. These included the type of problems that should be detected by penetration tests carried out during the development of the apps. We looked for:

- Sensitive data sent over insecure channels, using BurpSuite.³
- Basic invalid certificate verification, using Mallodroid⁴ and BurpSuite. Including:
 - Accepting self-signed certificates.
 - Not validating that requested hostname matches CommonName value or contained in SubjectAltName set.
- Secure HTTPS links sent over non-TLS connections. Using SSLStrip,⁵ we proxied traffic looking for HTTPS links that we could downgrade to HTTP.
- TLS version downgrade vulnerabilities. Many TLS client implementations do not solely rely on the standard negotiating mechanism. Some will make reconnection attempts using a downgraded version of TLS if initial handshakes fail. We tested for this by proxying TLS handshakes

¹ <https://github.com/google/nogotofail>

² see e.g. <https://geekflare.com/ssl-test-certificate/>

³ <https://portswigger.net/burp/>

⁴ <https://github.com/sfahl/malldroid>

⁵ <https://github.com/moxie0/sslstrip>

We found that none of the up-to-date apps in our test set exhibited any of these vulnerabilities. This led us to explore alternate ways that TLS could be broken or mis-configured.

Vulnerable Legacy iOS Apps Most of the banking apps we tested required you to have one of the latest iOS versions. However, as one of our testing devices was an old iPad running iOS v5, we found some apps that offered to install of the latest version of the app compatible with your OS. At first glance this sounds reasonable but when considering security critical applications like banking, this deserves more care.

Concretely, we found that the banking apps from NatWest and the Royal Bank of Scotland for iOS v5 would accept self-signed certificates. We established this by using the BurpSuite (without modifying anything on the iOS device) and verified that the apps would accept BurpSuite’s certificate and establish the connection as usual, but having BurpSuite as man-in-the-middle. These vulnerabilities have been patched and are no longer present in the latest version of the apps. However, users who have older iOS devices are still exposed.

Legacy apps should either be suitably maintained and patched or removed from the app store in order to avoid compromising user’s security and privacy. Experimenting with the apps, we found that both apps first asked for registration codes, which would be delivered to the user out of band. Once entered into the app these codes were sent over the insecure TLS connection, meaning that they could be captured by an attacker. Without accounts at either of these banks we were unable to experiment further.

Disclosure: We informed the banks concerned in January 2016, and shortly after both apps were updated, to no longer accept self-signed certificates.

4 Certificate Pinning Without Hostname Verification

In addition to checking that the server’s certificate has a verifiable chain of trust, it is essential to check that the requested hostname matches the CommonName field or is contained in the SubjectAltName set of the X.509 certificate. This however is an unnecessary step if the leaf certificate or the servers public key is pinned in the application, since the valid hostname is implicit. On the other hand, if the application is pinning to a root or intermediary certificate then hostname verification is still required.

Past work in 2012 [9] has demonstrated that incorrect hostname verification is a mistake that was regularly made by app developers. However, previous analysis of TLS usage in mobile apps has failed to consider the possibility of incorrect hostname verification when certificate pinning is in use.

To detect this form of invalid verification in mobile apps, we manually analysed apps using the following process:

1. Establish if certificate pinning is being used
 - (a) We add a custom self-signed root certificate, generated by Burp, to the phone’s CA trust store. The phone is also configured to use our machine running Burp as a proxy.
 - (b) We start the apps, one by one, and select the log in option to trigger a TLS connection attempt.

- (c) We observe the network traffic and if the app accepts the server certificate signed by the Burp root CA in the handshake, then we know the app is implementing chain of trust verification back to a trust store root certificate. If not, then the app must be pinning a particular certificate or public key.
2. Determine certificate chain in use
 - (a) From observations made in the previous test, we analyse the certificate chain provided by the server that the app is communicating with.
 - (b) Given this information, we can then obtain a certificate for our own domain from the same intermediary or root CA that the server's certificate is signed by.
 - (c) We then install the newly obtained certificate onto our own TLS server.
 3. Check if hostname verification is correctly implemented
 - (a) The app is then restarted and TLS connection attempts are triggered again.
 - (b) The TLS traffic is re-directed to our own TLS server and the handshake is analysed to determine if the hostname is being verified correctly. If so, then it should reject the TLS connection attempt, and if not, we should observe a fully successful TLS handshake with encrypted application data being sent by the app.

For apps that are found to pin a certificate but not check the hostname, an attacker could go to the CA used by the servers certificate and obtain a valid certificate in their own name, as we demonstrate in our testing method. The attacker's certificate can then be used to Man-in-the-Middle TLS traffic and hence break the security of the app.

Results: We followed the process described in section 4 for the iOS and Android apps from the UK banks in our test set. We found apps from two banks in our test set, Co-op and Natwest, passed our initial round of tests, and accepted certificates signed by the same expected root CA but for different hostnames. Co-op pinned to a Comodo root certificate, but didn't check the hostname. Natwest on the other hand, initially appeared to be pinning, but upon further analysis we found it was actually avoiding the phone's proxy settings. Hence, it would also accept a certificate signed by any of the phone's trusted root CAs.

While pinning the certificates is a good security measure, it seems possible that in the case of Co-op, pinning the certificate hid the more serious problem of having no hostname verification. Natwest's flaw was also hidden from the standard tests we carried out due to bypassing the phone's proxy settings, another reasonable security measure to take.

We note that these apps have a collective user-base of up to 5 million people, illustrating the seriousness and potential for this vulnerability to be exploited by an attacker. Additionally, this problem did not occur in any of the iOS apps that we tested which suggests that the iOS API makes it harder for this mistake to be made.

Attack Scenario: This attack could be carried out by an attacker that sets up their own Wi-Fi hotspot, it could also be carried out by any attacker on the route the data takes between the victim and the bank, or an attacker on the same network as the

victim, through use of techniques such as ARP or DNS poisoning. Other attacks (e.g. [11]) have shown that it is sometimes possible to access a home network, and when combined with these, an attack could be performed against anyone in the vicinity.

Disclosure: Both banks have been given over a year to ship fixes for their apps before we published the issues. We initially contacted Co-op on a Friday, some of their engineers met with us on the following Monday and they shipped a fixed updated app by the end of the week. One month later they removed all support for the vulnerable app and therefore this can no longer be used. A similar process was carried out with Natwest. We note that Co-op said that they had previously hired two penetration testing companies to test their apps, both of which had missed this vulnerability, so they had no reason to believe that their apps were insecure before we contacted them.

5 In-depth analysis of a Second Layer Banking Protocol

Breaking the TLS connection of a banking app does not necessarily mean that an attacker can gain access to a victim's account. Given the number of vulnerabilities that have been found in the TLS protocol [2,7,13], it would make sense for the apps to implement additional cryptographic protection.

The Co-op app we examined pinned to the Comodo root certificate (instead of the specific bank's certificate or the high security EV intermediate certificate). Since Comodo offers free certificates signed by this root certificate, authenticated only by e-mail, we were able to obtain a certificate for a domain we owned. Equipped with this certificate we were able to get the app to establish a TLS connection with us and take a look at the app's traffic which runs over TLS. We found the app first requested a one-time registration code, sent to the user out of band, after which it used an additional cryptographic protocol when the user logs on.

Traffic was sent between the app and the server using HTTP post requests containing JSON encoded data. Much of this included human readable tags. The first message sent by the app, and response from the server checked if the app needed to be updated. Next, the app sent a HTTP post request for an `RSAPublicKey`. The app then requested the user to enter their banking pin code, and the following two exchanged messages are shown in the appendix.

The fourth message includes a JSON encoded modulus and exponent, so it seems logical to

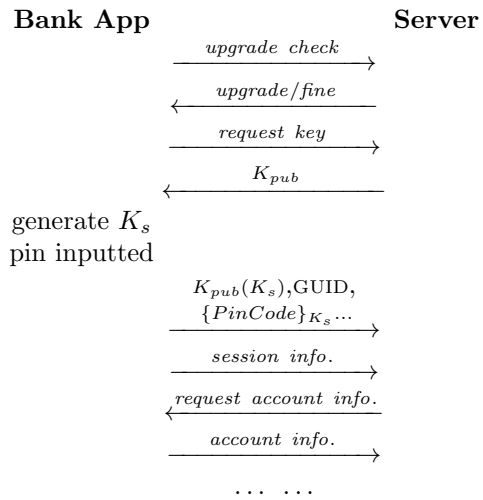


Fig. 1: A 2nd Level Banking Protocol

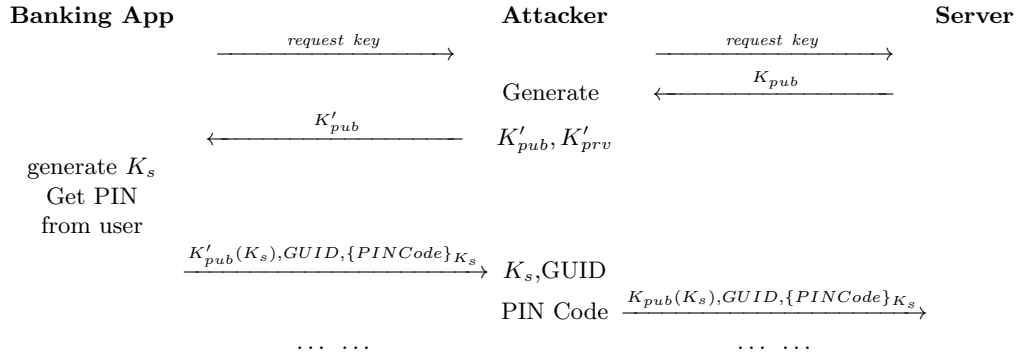


Fig. 2: A Simple MITM Attack on the banking protocol

conclude this is an RSA key. The 5th message contains a number of fields, of particular interest is the `passcode` field, which appeared to be encrypted, and the `guid` which appeared to contain the global unique identifier for the app. The `twk` field is exactly the right length for a single block of RSA cipher text.

To investigate this further we tried proxying the traffic, and sending our own RSA public key to the app. Doing so we found that we could then decrypt the `twk` field with our private key. This decrypted to 112-bits of correctly padded data. This is the correct length for two DES keys, and triple DES is a popular encryption algorithm in the banking community, therefore we tried decrypting the passcode field with these two keys using triple DES and found our banking PIN code, which was entered to access the account. Further investigation found that only the GUID and a correctly encrypted PIN code were needed to access the account, and no other cryptographic verification was required after these messages (although confirmation via SMS message was required to add new payees).

We summarise the protocol in Figure 1. As shown by our ability to learn the PIN code this second level protocol is flawed, and therefore does not provide any additional protection to the communication channel. Figure 2 describes a simple man-in-the-middle attack against this protocol, in which an attacker can substitute their own key for the banks. By running this attack, adversaries can learn both the GUID and the user’s PIN number, which is all they need to login to the user’s account. If the attacker wants to hide their actions they can continue relaying traffic between the app and the bank, and store the GUID and the PIN for use at some later date and some other location.

We additionally found that the bank did not use a fresh RSA key for each session, in fact for the 2 month period we studied this app, the key was always the same. This means that an attacker can simply record the traffic from the app and replay the `twk` and the `passcode` fields, to gain access to the victims bank account, at any time, without ever actually learning the PIN code.

We note that obfuscating the messages would not have improved security, given the distinct sizes of the cryptography involved, it would have been relatively easy to

reverse engineer this protocol without the human readable tags. What would have improved security would have been a correct protocol, which was not vulnerable to MITM and replay attacks, as we propose in Section 7.

Disclosure: We described the problems with this protocol to the Co-op at the same time as we disclosed the pinning but no hostname vulnerabilities, and we suggested a secure alternative. The Co-op decided to only fix the TLS vulnerability and leave the underlying protocol as it was. We note that the use of this underlying protocol does not represent a vulnerability, but also does nothing to protect the communications link.

6 In-App Phishing Attacks

While examining the banking apps from our test set, we noticed the First Trust Bank, Santander and Allied Irish Bank apps mixed TLS and non-TLS traffic, requesting some resources over a secure TLS connection and others over an unprotected connection.

Mixing TLS and non-TLS traffic, does not necessarily make an app vulnerable, so we examine the app in more detail and found that it was loading adverts for the banks own products and links to information that may be helpful to its users. In particular, the app loads an image to be displayed to the user and a link to go to, if the user clicked on the image.

Figure 3a gives an example of this. The “Help Centre” box in the middle of the screen on the left was loaded dynamically, and clicking on this takes the user to the screen on the right. At other times adverts for mortgages and savings accounts were loaded.

Downloading images and links over unprotected connections allows an attacker, on the same network as the victim, to replace them with an image and link of their own choice. This could be used to perform a phishing attack. While phishing attacks against banking credentials are common, being able to carry out the phishing attack inside the actual banking app means that the user is far more likely to trust the link.

We give an example of this in Figure 3b. On the left is a screenshot of the real banking app, which we connected to a wi-fi hotspot we controlled. As no protection is used we were able to replace the “Help centre” image with a “Mobile Banking Log In” image (the real log on option can only be accessed via the “Banking” icon). We additionally sent our own link to the app that sends the user to the page shown on the right of Figure 3b: a page with a spoofed URL which steals the users credentials.

In the case of Santander UK, the image was displayed on the bottom portion of the screen, e.g. the mortgage panel in Figure 3c. Clicking on this takes the user to a Santander website. We show a possible in-app phishing attack against this in Figure 3d. The real option to view the users accounts is under the menu at the top right of the screen, so a large “My Accounts” button with the same logo as the real one will likely attract many users. Additionally, we found that the information asked for on our phishing page, on the right of Figure 3d, is all that is needed to install the Santander UK app on another phone. Again, as the app does not use TLS on the site it links to, our site’s URL can be spoofed to look like a Santander page to the user.

The Allied Irish Bank app turned out to be very similar to the First Trust Bank, and we found out that the First Trust Bank is in fact a trading name for Allied Irish.

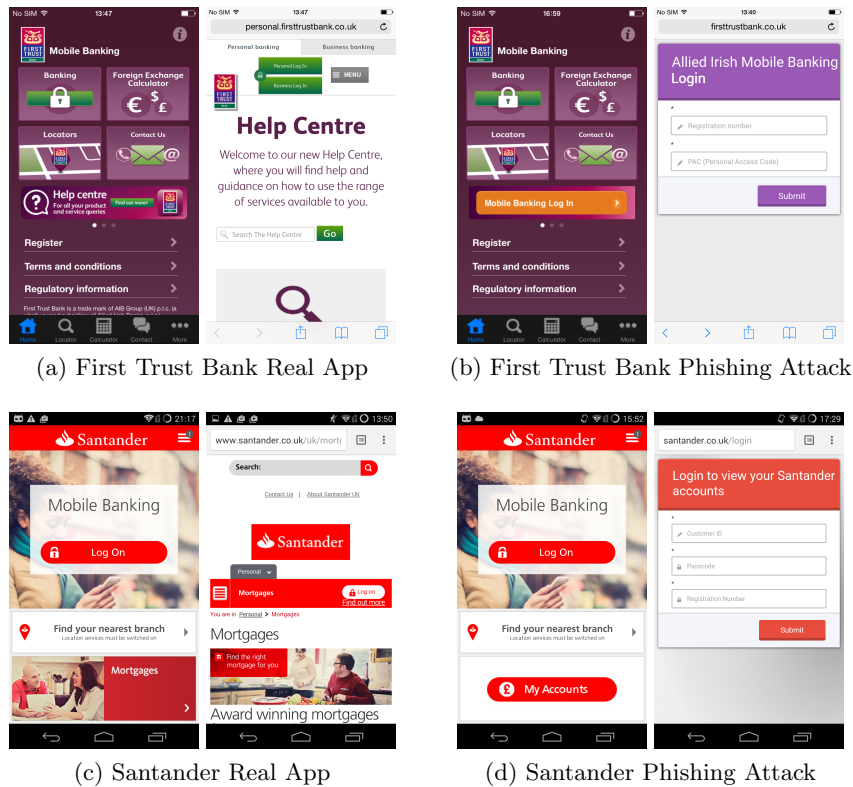


Fig. 3: In-App Phishing Attacks

We additionally found that Allied Irish Bank performs an un-encrypted version check, which, in the case the app needs updating, the server replies to with a link to the Android Marketplace. This can be hijacked to redirect the app's user to any website of the attackers choice.

Attack Scenario: To perform this attack, the attacker must have the same access as required for the previous attack. However, for this attack, they do not need a certificate of their own; as there is no protection on the image and link sent to the app an attacker can substitute their own image and link. Additionally, the attacker can spoof the address of the page to make it seem like it belongs to the bank (none of the apps we found directed users to pages protected with TLS). We emphasise that the attacker does not need to interfere with the victims phone or the banking app in anyway, this attack is carried out purely externally.

Some of the apps we looked at did not have a direct log in button on the opening page of the app. So placing such a button on the first page, with the same look as the other buttons on the app, would attract a lot of users. For systems that didn't require two factor authentication, the users could then be asked for their online banking

credentials, or the information needed to install the banking app on another device, so giving the attacker full access to the victims online banking.

Disclosure: These vulnerabilities were disclosed to the banks involved in January 2016 and they were fixed the following month. Shortly following this Allied Irish and First Trust Bank updated their app to a completely new code base, which did not have these errors.

7 A Secure Protocol for Mobile Banking

We have recently seen many attacks against TLS and we are likely to see more in the future. These attacks will compromise the security of TLS used in apps (indeed, when we first approach one of the banks to say we had discovered a vulnerability, their first reaction was to ask if it was the POODLE attack, as they were aware that this had been a problem and had only just fixed it). Additionally, PCI standards on PIN security state that PINs must be encrypted at all times [15], so it should not be sent over TLS in the clear if the host receiving it is just going to send them on to another host to be verified.

In our view this means that TLS should not be the only protection and a lightweight, second level protocol to protect the PIN number and messages should also be used. However, as we have shown in Section 5, designing such a protocol appears to be error prone. In this section we work constructively, proposing a protocol which is suitable for this purpose and that can be used, freely, by any system designer. The protocol is a quite straightforward adaptation of an RSA encrypted key transport protocol as depicted in Figure 4.

This protocol starts off as the protocol in Section 5, with an update check. After that the user requests and the server generates a secure random 128-bit nonce. The banking app has the public key of the bank K_{pub} hard-coded, and this can be updated automatically by pushing an app update. This makes use of a second channel (the app update process) to split the public key distribution from the PIN transfer. During authentication, the app will generate a session key K_s and a MAC key K_m uniformly at random, we would recommend AES 128-bit, but 3-DES may be required for compatibility with backend banking systems. When users input their

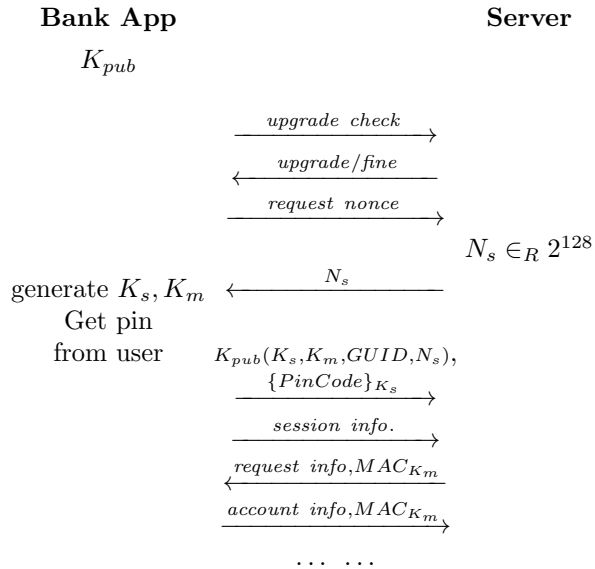


Fig. 4: An Improved Secure Banking Protocol

PIN codes, the app will send an encryption of the session key, a MAC key, app GUID and a nonce from the server encrypted with the banks public key: $K_{pub}(K_s, K_m, GUID, N_s)$, along with an encryption of the *pincode* with the session key K_s (plus potentially any relevant session information as seen in Section 5). From this point on the app and the server exchange information in the usual way but authenticating every message with a CBC-MAC using K_m . We pad the PIN and the messages for the CBC-MAC with PKCS5.

Unlike the original protocol, our version includes the GUID in the block that is encrypted with the banks public key. We note that the session key is 128/112 bits therefore, when encrypted in the original protocol it will be padded out to 2048 bits. So in our suggested version of the protocol the GUID and server nonce replaces some of this padding, therefore we require no additional encryption and the message becomes shorter.

We note that this protocol does not keep the session data secret, or prevent message replays in the same session. This should be done by the application layer and (most of the time) by TLS. What it does aim to do is to provide protection for the customer that stops an attacker using their account in the case that the TLS protections fail (which is unfortunately common e.g. [2,9,13]), in particular it will keep the data needed for an attacker to log onto the victims account secret, it will prevent replay attacks across different sessions, and it will prevent an attacker from altering messages. Furthermore, it provides this additional protection using a protocol that is slightly shorter and largely compatible with the original broken version.

Additionally, in the case that TLS fails, an attacker is prevented from distributing to the victim a version of the app containing their own public key. In Android, app updates are digitally signed by the app developer and the OS will check that the update is signed with the same key as the current version of the app.⁶ In iOS, the situation is even more robust. App updates are signed by the developer and are then uploaded to their iTunes Connect account. Apple then verify the update themselves and then re-sign it with their key before distributing it to customers.⁷ Upon receiving app updates, iOS devices will check it is signed by Apple. In both cases, as long as the underlying signature scheme is not broken, an attacker cannot change the public key that is used in our proposed protocol.

Formal Analysis of Our Protocol To offer assurance that our protocol is correct we analyse it using the formal protocol analysis tool ProVerif [3]. This uses a simple language for modelling protocols and can automatically prove a wide range of security properties, including secrecy, absence of offline guessing attacks, and correspondence assertions. Because ProVerif uses an automated theorem proving method it can prove that these properties hold against an active Delov-Yao attacker for an unbounded number of runs of the protocol and protocol participants, however it may sometimes fail to terminate, or report false attacks.

The main part of our model is presented in the appendix of the online version of this paper. It includes an **App** process to model the smart phone app, and a **Server** process that defines a bank process and the entire system is defined on the last

⁶ <https://developer.android.com/studio/publish/app-signing.html>

⁷ <https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingCertificates/MaintainingCertificates.html>

line using the `process` key word. The system models an unbounded number of banks that may be using the protocol, each bank has an unbounded number of customers, some of which may be using the same PIN, but each of which has a unique GUID, for each customer we run an arbitrary number of `App` processes representing the user running the app and `Server` processes representing the bank handling this particular user.

We use the ProVerif tool to first check that this protocol keeps the session key and the PIN code secret, which ProVerif confirms it does. We next check the authentication properties of this protocol, in particular that every time a server finishes with a particular set of values (a `serverFinish(ks,km,guid,pin,bankkey)` event is reached) the app has also run the protocol using the same values, i.e., a `appInit(ks,km,guid,pin,bankkey)` event was reached with the same values as with the `serverFinish` event. We also check in the other direction confirming that the `appFinish` is always matched by a corresponding `serverInit` event with matching values. ProVerif confirms that these correspondences always hold indicating that an attacker cannot impersonate one side of the protocol or interfere with any of the values used, without being detected. Finally, as the PIN code may be a low entropy secret we check if an attacker can perform an offline guessing attack to find it, using ProVerif's `weaksecret` test and a second model that uses a fixed PIN code. ProVerif confirms that no such guessing attack exists.

8 Conclusion

In this paper we have carried out an extensive security analysis of TLS implementations in the UK's major banking mobile applications. We have discovered a new vulnerability in one of these banking apps that arises due to a misuse of certificate pinning. To avoid breaking the terms and conditions of high-security apps, such as banking, we have shown to detect this vulnerability dynamically and without reverse engineering the app. We also found one app that fails to verify the hostname but hides this flaw from standard tests by bypassing proxy settings. These two cases are examples of how additional security protections have obscured the serious vulnerability of no hostname verification. We have demonstrated how these vulnerabilities could be exploited by an attacker to break the security of the app and eavesdrop sensitive information. We also found apps that are vulnerable to phishing attacks, legacy apps that accept self-signed certificates and a broken application layer banking protocol.

A Co-op app traffic

```

HTTP/1.1 200 OK
Server: webserv
X-response-id: -907159463
Pragma: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Cache-control: no-cache no-store
Content-type: application/json; charset=UTF-8
Content-Length: 311

{"modulus": "9C8C54XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 4082C1", "exponent": "3", "keyVersion": "81XXXXXX
C1"}

POST /mrs/3/security/session HTTP/1.1
user-agent: Mozilla/5.0 (Linux; Android 4.4.2; XT1021 Build/KXC20.82-14)
AppleWebKit /537.36 (KHTML, like Gecko) Version/4.0 Chrome/30.0.0.0 Mobile
Safari/537.36
client-version: 4.4.2
X-Request-Id: -907159453
Accept: application/json
Content-Type: application/json; charset=UTF-8
Cookie: JSESSIONID=9f7b4441e12b265fde33d1f0e73b; Path=/mrs; Secure
Cookie: JROUTE=9qIW.9qIW; Path=/mrs; Secure
Content-Length: 577
Connection: Keep-Alive

{"msisdn": "44XXXXXXXXXX", "clientType": "|2.2.7| |motorola|XT1021|Android|4.4.2|
|", "passcode": "93XXXXXXXXXX07", "twk": "8170b6XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXabf54c",
"pinLength": "6", "guid": "d2xxxxxxxxxxxxxxxx83", "applicationName": "Coop",
"deviceID": "44XXXXXXXXXX", "issuerName": "COOP", "authenticationType":
:"passcode", "clientVersion": "2.2.7", "rootDetection": "NOT_DETECTED"}

```

Fig. 5: Messages sent between the app and bank (with some information blanked out)

References

1. B. B. Association. The way we bank now, 2016.
2. B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security and Privacy*, 2015.
3. B. Blanchet, B. Smyth, and V. Cheval. Proverif 1.88: Automatic cryptographic protocol verifier, user manual and tutorial, 2013.
4. C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 114–129. IEEE, 2014.
5. R. Carolina and K. Paterson. Megamos crypto, responsible disclosure, and the chilling effect of volkswagen aktiengesellschaft vs garcia, et al. 2013.
6. O. Choudary. The smart card detective: a hand-held emv interceptor. 2010.
7. J. de Ruiter and E. Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., Aug. 2015. USENIX Association.
8. C. Evans and C. Palmer. Certificate pinning extension for hsts. 2011.
9. S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android SSL (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12*, 2012.
10. M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12*, pages 38–49, New York, NY, USA, 2012. ACM.
11. E. N. Lorente, C. Meijer, and R. Verdult. Scrutinizing WPA2 password generating algorithms in wireless routers. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., Aug. 2015. USENIX Association.
12. M. Marlinspike. New tricks for defeating SSL in practice. In *Black Hat Europe*, 2009.
13. B. Möller, T. Duong, and K. Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback, 2014.
14. M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl. To pin or not to pin—helping app developers bullet proof their tls connections. In *USENIX Security*, pages 239–254, 2015.
15. P. C. I. PCI. Pin security requirements. 2014.
16. H. Perl, S. Fahl, and M. Smith. You wont be needing these any more: On removing unused certificates from trust stores. In *International Conference on Financial Cryptography and Data Security*, pages 307–315. Springer, 2014.
17. B. Reaves, N. Scaife, A. Bates, P. Traynor, and K. R. Butler. Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.
18. I. Sample. Bankers fail to censor thesis exposing loophole in bank card security. The Guardian, 2010. <https://www.theguardian.com/science/2010/dec/30/bankers-thesis-bank-card-security>.
19. N. Vallina-Rodriguez, J. Amann, C. Kreibich, N. Weaver, and V. Paxson. A tangled mass: The android root certificate stores. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 141–148. ACM, 2014.