Spinner: Semi-Automatic Detection of Pinning without Hostname Verification

Chris McMahon Stone
University of Birmingham
Birmingham, UK
c.mcmahon-stone@cs.bham.ac.uk

Tom Chothia
University of Birmingham
Birmingham, UK
t.p.chothia@cs.bham.ac.uk

Flavio D. Garcia
University of Birmingham
Birmingham, UK
f.garcia@cs.bham.ac.uk

ABSTRACT

Certificate verification is a crucial stage in the establishment of a TLS connection. A common security flaw in TLS implementations is the lack of certificate hostname verification but, in general, this is easy to detect. In security-sensitive applications, the usage of certificate pinning is on the rise. This paper shows that certificate pinning can (and often does) hide the lack of proper hostname verification, enabling MITM attacks. Dynamic (black-box) detection of this vulnerability would typically require the tester to own a high security certificate from the same issuer (and often same intermediate CA) as the one used by the app. We present Spinner, a new tool for black-box testing for this vulnerability at scale that does not require purchasing any certificates. By redirecting traffic to websites which use the relevant certificates and then analysing the (encrypted) network traffic we are able to determine whether the hostname check is correctly done, even in the presence of certificate pinning. We use Spinner to analyse 400 security-sensitive Android and iPhone apps. We found that 9 apps had this flaw, including two of the largest banks in the world: Bank of America and HSBC. We also found that TunnelBear, one of the most popular VPN apps was also vulnerable. These apps have a joint user base of tens of millions of users.

CCS CONCEPTS

• **Security and privacy** → **Mobile and wireless security**; *Web protocol security*; *Network security*;

1 INTRODUCTION

TLS is a tricky protocol to get right: both misconfiguration vulnerabilities (e.g. [10, 16]) and attacks on the protocol are common (e.g. [2, 4, 7, 17]). A TLS server will authenticate itself to a client by presenting it with a certificate containing its public key. The client will then, providing it is correctly configured, verify this certificate by making a number of checks. The most vital of which are that it is correctly signed by a trusted Certificate Authority (CA) and that the hostname (contained either in the Common Name or Subject Alternative Name fields) matches that which is expected.

Like web browsers, mobile platforms such as Android and iOS rely on a trust store containing a large number of CA root certificates. If a single CA acted maliciously or were compromised, which has happened before (see e.g. DigiNotar in 2011 [15]), valid certificates for any domain could be generated allowing an attacker to Man-in-the-Middle all apps trusting that CA certificate.

Perl et al. [19] argue that a large number of certificates could be removed from trust stores to reduce the risk of such an event. However, applications can mitigate the problem to a much greater extent by implementing an additional security measure known as Certificate Pinning. Here, developers can choose to only accept certificates signed by a single *pinned* CA root certificate. Alternatively, but at the cost of reduced flexibility, a leaf certificate can be pinned. Although developers have been aware of the technique for some time, Oltrogge et al. [18] found that adoption has been slow, citing misunderstanding and complex implementations as the root causes. Despite this, Chothia et al. [6] discovered that a large proportion of the high security UK banking apps they tested were implementing pinning. Focusing their analysis on these apps, they considered the possibility that apps which pinned to a CA root certificate, correctly checked that server certificates were signed by this root CA, but failed to verify the hostname.

Automated tools do exist to test a variety of TLS flaws. Lack of certificate signature verification can be tested for by serving the client a self-signed certificate, lack of hostname verification by serving a valid certificate for a different hostname, and lack of certificate pinning can be checked for by adding a custom CA to the device's trust store. These tests have been shown to be effective at finding vulnerabilities in apps [10] and poor TLS certificate validation [5]. However, none of these tools can detect the possibility that an app will pin to the root or intermediate certificate used but fail to validate the hostname.

In their analysis of banking apps, [6] formulated a manual method to detect this issue. For each app that used pinning, a certificate for a domain under the tester's control was purchased from the same CA signing the domains that the app was legitimately trying to connect to. These were then installed on a TLS testing proxy one-by-one when testing each app.

In this paper we argue that conducting large-scale testing in this manner is difficult and expensive. Many apps pin to a high security intermediate or root certificates and these are only available after careful ID checks and payment of a large fee (below we estimate the cost of purchasing all certificates needed to test all Android and iPhone apps at over $100K a year). Without access to a certificate as described above, the use of pinning makes the underlying problem of no hostname verification much harder to detect.

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

We address these issues by designing and implementing a zero-cost, automated tool, named Spinner, to facilitate testing of this vulnerability at scale.

1.1 Related Work

There are a number of past studies that have looked at certificate verification in non-browser software. Fahl et al. [10] developed a tool that statically analyses Android code and detects non-standard implementations. They then go on to manually test for Android apps that accept self-signed certificates or do not verify hostnames. Carrying out this process with over 10,000 apps, they find a surprisingly large portion of them contain these serious flaws and hence expose themselves to Man-in-the-Middle attacks.

Georgiev et al. [13] independently carried out a similar study. In addition to Android, they also considered other platforms and applications such as instant messenger clients, merchant payment SDKs and cloud client APIs. Brubaker et al. additionally developed a tool to fuzz certificates [5]. By generating certificates with mutated fields, and then presenting them to clients during a TLS handshake, they also find many critical bugs in the verification code in a number of implementations.

Sounthiraraj et al. [23] improved on the analysis techniques proposed in [10]. They develop a tool named SMV-Hunter, which, in addition to statically analysing source code to detect non-standard implementations, carries out dynamic tests by automatically running apps and emulating intelligent user input to trigger TLS connection attempts. Basic TLS certificate mis-verification flaws are then detected automatically. We note however, that this tool also lacks the capability to detect hostname verification flaws when certificate pinning is being used.

Concentrating on the hostname verification process, Sivakorn et al. [21] find that the set of acceptable hostnames specified by the patterns in the CommonName (CN) and SubjectAltName values form a regular language. They use automata learning algorithms to infer the corresponding models, and then use this to find discrepancies in particular hostname verification implementations. Running these tests on popular TLS libraries, they find 8 unique violations of RFC specifications, several of which render them vulnerable to man-in-the-middle attacks.

Previous research has also focused on applications implementing certificate pinning. Oltrogge et al. [18] scanned a large portion (around 600,000) of the Play Store to find out how many were employing this additional security measure. Using various metrics, they determine and recommend that significantly more apps would benefit from certificate pinning. They also interview a number of developers and find that general understanding on the topic is good, but the complexity of implementation methods makes it too difficult to use. Implementations with serious flaws discovered in [6] backup this finding.

Previous work on tools has not considered the case of missing hostname verification when certificate pinning was being implemented. Chothia et al. [6] manually analysed the way in which TLS was being used in apps UK's largest banks. They found a number of misuses, including two apps from major banks that pinned a root CA certificate but did not validate hostnames. They tested for this by purchasing a certificate from the same CA being used by

App name	No. of Installs	Platform
Bank of America Health	100k - 500k	Android
TunnelBear VPN	1m - 5m	Android
Meezan Bank	10k - 50k	Android
Smile Bank	10k - 50k	Android
HSBC	5m - 10m	iOS
HSBC Business	10k - 50k	iOS
HSBC Identity	10k - 50k	iOS
HSBCnet	10k - 50k	iOS
HSBC Private	10k - 50k	iOS

Figure 1: Apps that implement certificate pinning but fail to verify hostname.

each app, but for a domain they owned. The work presented in this paper automates this testing technique and removes the need to purchase certificates.

1.2 Our contribution

This paper presents a black-box method to detect apps (or devices in general) that, when using TLS, pin to a root or intermediate certificate but do not check the hostname of the host they connect to. Instead of trying to get certificates from all possible certificate authorities, which we argue is infeasible, we build a tool that makes use of the Censys Internet scanning search engine. Given the certificate for a target domain, the tool queries for certificate chains for alternate hosts that only differ in the leaf certificate. The tool then redirects the traffic from the app under test to a website which has a certificate signed by the same CA certificate, but of course a different hostname (Common Name). If the connection fails during the establishment phase then we know the app detected the wrong hostname. Whereas, if the connection is established and encrypted application data is transferred by the client before the connection fails then we know the app has accepted the hostname and is vulnerable. The key insight here is that although we cannot decrypt the traffic, the information we need is provided by analysing exactly when and how the TLS communication fails. While the tool itself is fully automated, we refer to the testing framework as semi-automated, as the user still needs to install and run the app by hand (this is something that could also be automated using an app emulator).

Using this tool, we then carry out a test of 400 iOS and Android high security applications including banking, stock trading, cryptocurrency and VPN apps. We find 9 new apps that pin to root or intermediate CA certificates but fail to verify the hostname, rendering them all vulnerable to Man-in-the-Middle attacks. Figure 1 summarises our findings. We note that the total user base of these apps is tens of millions of users, which highlights the severity of the issue. We then reverse engineered each of the vulnerable apps in order to get a better understanding on how this weakness is introduced by the app developers and how to prevent it.

Spinner will be released under the Creative Commons Licence, as we believe that it would be a valuable addition to every pen tester's toolbox.

Structure of the paper

The next section provides some background, briefly describing certificate pinning and some elements of the TLS protocol. Section 3 discusses the problem of pinning without hostname verification and in Section 4 we describe our test framework in detail. We discuss the results of our analysis of security sensitive apps from the Android and iPhone app stores in Section 5. In Section 6 we discuss the impact of the vulnerabilities we find and in Section 7 look at the apps in detail to find out why hostname verification failed and if this weakness would be detectable by static checking. We conclude in Section 8.

2 BACKGROUND

2.1 Transport Layer Security (TLS)

In order to establish a secure TLS channel, part of the initial handshake involves the communication of a server's X.509 certificate. The purpose of this certificate is to bind the server's identity to a given public key. It is the job of the client, to verify this certificate so that it can be sure the public-private key pair belongs to the server it intends to be communicating with. Once this verification process is complete, the public key encapsulated in the certificate is used to negotiate a session key.

In a typical TLS set-up, this trust is enabled through the maintenance of a set of by-default trusted root CA certificates. Servers wishing to use TLS, must obtain their own cryptographically signed certificate from one of these CAs. A client wishing to set up a TLS connection will then validate the server's certificate by carrying out a number of checks: Recursively checking the signatures of each certificate in a chain to a trusted anchor, verify the certificate was issued to the expected host, ensuring the certificate has not been revoked or has expired.

Unfortunately, Certificate Authorities are not immune from compromise, which if were to happen (and has done before, see DigiNotar [11]), would enable the perpetrator to generate valid certificates for domains of their choosing. These forged certificates could then be use to Man-in-the-Middle TLS connections from any client trusting the compromised CA. Furthermore, a number of CAs have been caught mis-issuing certificates. In early 2017, Google announced plans to gradually phase out trust of Symantec certificates in the Chrome browser, citing concerns that the organization had mis-issued thousands of certificates [22]. Developers are further motivated to avoid reliance on a devices trust store due to the potential for users to be phished into inserting malicious certificates into to trust store.

2.2 Certificate Pinning

In many mobile applications, unlike generic browsers, the client knows the identity of the server(s) in advance. The consequence of this is that dependence on CAs can be entirely removed, or reduced down to a single CA. Conceptually, certificate pinning achieves this by allowing a developer to implement their own trust store. In practice, certain elements of a certificate chain are fixed or hard-coded into the application. Exactly which certificate in a chain is pinned, depends on the developer's requirements:

- **Leaf Certificate** - The most obvious element to pin is the end-entity or leaf certificate. No CAs are relied upon and attack surface is minimal. Flexibility is reduced however; if key rotation policy mandates that the certificate is changed, then users will be required to update their apps to continue use, similarly when the certificate expires.
- **Intermediate Certificate** - Certificate chains will typically be of length 3 or more. If the certificate is neither a root, nor a leaf, then it is called intermediate. Pinning to this type of certificate enables the end-entity to easily renew it's leaf certificate (as long as it is signed by the pinned intermediate).
- **Root Certificate** - Pinning to a root certificate is the most flexible approach as length of validity is usually ~20 years. However, it is also the least secure.

Developers must also deliberate over whether the entire certificate or just the public key is pinned. Pinning the certificate is the most common implementation, but suffers from issues regarding certificate expiration. On the other hand, public key pinning is less widely used, but allows for key continuity even when the certificate has expired.

2.3 Server Name Indication

In standard TLS, it is a requirement that each host, with its corresponding certificate, has its own IP address. The implication of this is that virtual hosting, where multiple domains, each with their own certificate, share a single IP address, is not possible. To overcome this limitation, the Server Name Indication (SNI) extension was proposed in RFC6066 [9]. The solution works by enabling the client to specify the host it wishes to communicate with in the ClientHello message. The server can then inspect this value and serve to the client the appropriate certificate for the requested hostname.

We use the SNI to facilitate transparent TLS proxying; as some mobile applications bypass manual proxy settings, we require a methodology that transparently intercepts and redirects network traffic. Unfortunately, DNS spoofing on its own does not suffice. Many clients will make numerous DNS lookups before making any TLS connections. Therefore we use SNI to distinguish between TLS connections, so that they can then be proxied to the appropriate host.

3 PINNING WITHOUT HOSTNAME VERIFICATION

TLS misconfiguration vulnerabilities are clearly common, however none of the existing frameworks will detect that a client pins a root or intermediate certificate, but fails to check the hostname in the leaf. Obtaining a complete set of test certificates is not feasible, many apps use intermediate certificates that are only available after thorough identity checks and payment of a large fee (see below). This might be an option for a well motivated attacker who knows they will get a large financial pay-off, but it is beyond the scope of speculative checking.

Another option involves reverse engineering the apps [20] but this is often a time involving process which requires an expert. We are after a solution that scales. Our solution is to make use of certificates that have already been issued to other websites such as www.facebook.com and www.ibm.com. We cannot obtain the

private keys for these certificates, and so we cannot use them to decrypt the traffic. However, we can direct traffic from the app to this other website and analyse the result. Explicitly, we can direct traffic from the app to a website using a TLS connection and a certificate signed by the same root and/or intermediate certificates, but with a different hostname. If the client rejects this connection we can be sure that it checks the hostname and found it incorrect, but if the app proceeds to send encrypted data, then we can tell that the app accepted the certificate with the incorrect hostname.

We can then go on to perform additional checks to see if the app actually accepts any certificate (i.e. it does not in fact pin the certificate) by serving the app a legitimate certificate with a different CA. We also check which element of the certificate chain the app is pinning to. We automate this process using the framework described in the next section.

We note that this framework is aimed at only detecting apps that pin the certificate, but do not check the hostname, we are not, for instance, testing for apps which accept self-signed certificates for the correct hostname, or apps that make connections to hard-coded IP addresses, as detection frameworks for such vulnerabilities is the subject of much other work [3, 5, 23].

Purchasing all certificates

An alternative to the method presented here would be to adopt the methodology followed by Chothia et al. [6]. Here, a pentesting company would have to purchase one of every possible certificate, for their own domain, and then use these to test the app. However, we do not think such a method would be practical. Looking at just one certificate authority: Symantec uses 14 intermediate certificates, which collectively link to 4 root certificates[1]. The particular certificate you are allocated depends on a combination of your product requirements (e.g. do you need Extended Validation/Wildcards) and your choice of cryptography (SHA-1 or SHA-2, ECC, RSA, DSA). At an average price of \$1,200 per certificate, to obtain all 14 Symantec certificates alone would cost \$16,800 per year. There are 146 root certificates on the Android platform [19] and roughly 4 intermediate certificates per root certificate. Symantec seem to be one of the most expensive certificate authorities on the market, but even assuming that the average certificate authority charges only a quarter of the price of Symantec, to maintain a collection of all certificates required for testing would cost well over \$100k. Of course, limited coverage would be possible for a smaller price, but it would still cost a very substantial amount of money to effectively check apps using purchased certificates.

Moreover, many of the certificates used by the apps are Extended Validation certificates [12]. These certificates are only available to registered businesses (certificate authorities are explicitly forbidden from issuing EV certificates to unincorporated businesses). The guide lines for EV certificates require a number of legal and identity checks that must be carried out by hand before a EV certificate can be issued, and certificate authorities are audited against these checks. Therefore, it is possible that a certificate authority would refuse to issue multiple EV certificates to a single company to avoid difficulty in the audit process, and so it might be necessary

to register a number of different companies in order to acquire a complete set of certificates.

4 A FRAMEWORK FOR SEMI-AUTOMATIC DETECTION

The key elements of our framework are a method for looking up the domain of websites that use a given certificate chain, a custom built DNS server that will let us whitelist some URLs and redirect others to an IP of our choice, and a TLS server that will redirect TLS messages and look for encrypted application data. See Figure 2 for a high level overview of the framework.

To find domains running TLS servers with specified certificate chains, we make use of the Censys[2] search engine, which hosts information on internet hosts by carrying out daily scans of the IPv4 address space [8]. By analysing the certificate chain in use by a domain that the app attempts to communicate with, the tool extracts the Common Name value embedded in the issuing certificate (either an intermediate or root certificate). This value is then used to construct a *Certificate* API query to Censys, which when executed, provides the tool with a list of domains corresponding to our requirements. The tool then selects the first reachable domain and communicates this to the TLS server accordingly.

To accommodate for the possibility that Censys is not accessible, we additionally provide a static database containing a mapping of issuer certificates to a set of hostnames in possession of a corresponding certificate. This was built by scanning the certificate chains in use by the Alexa top 1 million websites. In total, our database contains 33,601 unique issuing certificates, which easily satisfied the requirement for all the apps we tested.

The DNS component of the tool is used to redirect the messages from the apps being tested to the TLS proxy, and the TLS server will redirect the apps traffic to a site selected from either Censys or the database, and inspect the handshake see if the app accepts or rejects the connection.

Experimenting with apps we found five distinct ways in which TLS connections could fail, these are depicted in Figure 3. In the case when a server is using a certificate for the wrong hostname, the client may send a TLS alert message as soon as the server sends the certificate and then close the connection (Figure 3.a). This is the first point in the protocol at which the client can detect the bad certificate. In contrast to this, we found that some Apple apps would omit the transmission of an alert message and instead just stop communicating. We also found examples of Android apps that continued the TLS protocol until after the two ChangeCipherSuite messages were sent, and then sent an alert message and closed the connection (Figure 3.c). A variation of this is depicted in (Figure 3.d).

Interestingly, in cases (c) and (d), we further observed a slightly different breakdown in the connection. That is, in the time just before a client sent an alert message and then closed the connection, the server would send some encrypted application data. This bears the important consequence that the framework must only detect a successful connection if a client sends encrypted application data.

---

[1]https://knowledge.symantec.com/support/ssl-certificates-support/index?page=content&actp=CROSSLINK&id=INFO4033
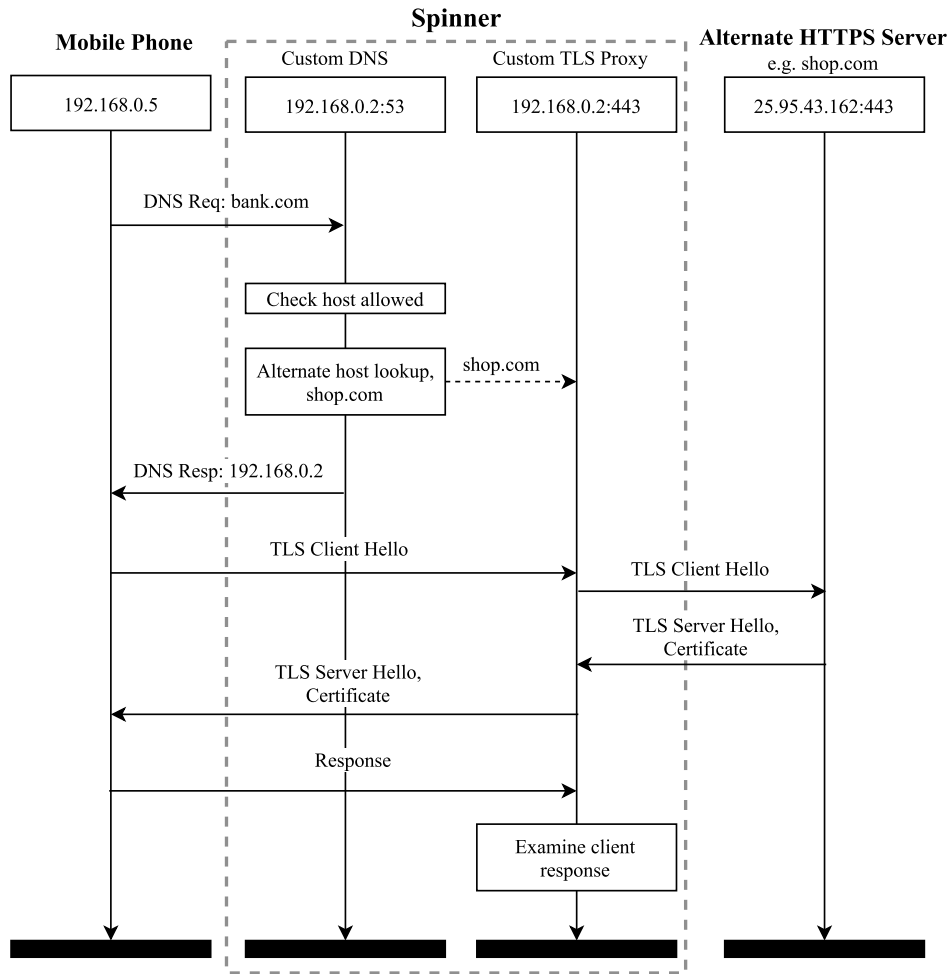
[2]https://censys.io/

Figure 2: Overview of Spinner

To summarise, our framework TLS server detects either an alert message or a time out as a fail connection and detects any connection where the client sends encrypted application data as a successful connection.

Using our TLS server, DNS server and our database, our framework then tests apps using the following three stage process:

(1) **DNS Look Up Stage**

(a) We start our framework running, which listens on ports 53 and 443 for DNS and TLS requests respectively. We load all the apps under test onto a phone and set the phone to use our framework as its DNS server. Note that this caters for apps which bypass HTTP proxy settings. Our tool will additonally monitor the network to flag any apps that bypass DNS settings (this is very uncommon).

(b) We start the apps, one by one, and select the login option to trigger a TLS connection attempt.

(c) The app sends DNS lookup request(s) to our framework for the domain(s) it wishes to connect too.

(d) Our framework looks up the real IP address of the requested domain(s). It then requests the certificate from the server the app wishes to connect to. Using this, it then queries Censys (or the static database) to find the URL for another site which uses the same certificate chain as the site the app wished to talk to. We note that some apps query for IP addresses for multiple domains at once, before any connections are made. Therefore, a mapping of requested hostnames to redirect domains is maintained.

(e) The IP address of the machine running our framework is then returned to the app, as the answer to its DNS request.

(2) **TLS Stage**

Once the app knows to direct its traffic to our tool's IP address, the following process is executed. Each connection to each domain is tested in isolation. To acheive this, all other connections are whitelisted and proxied to their legitimate servers.
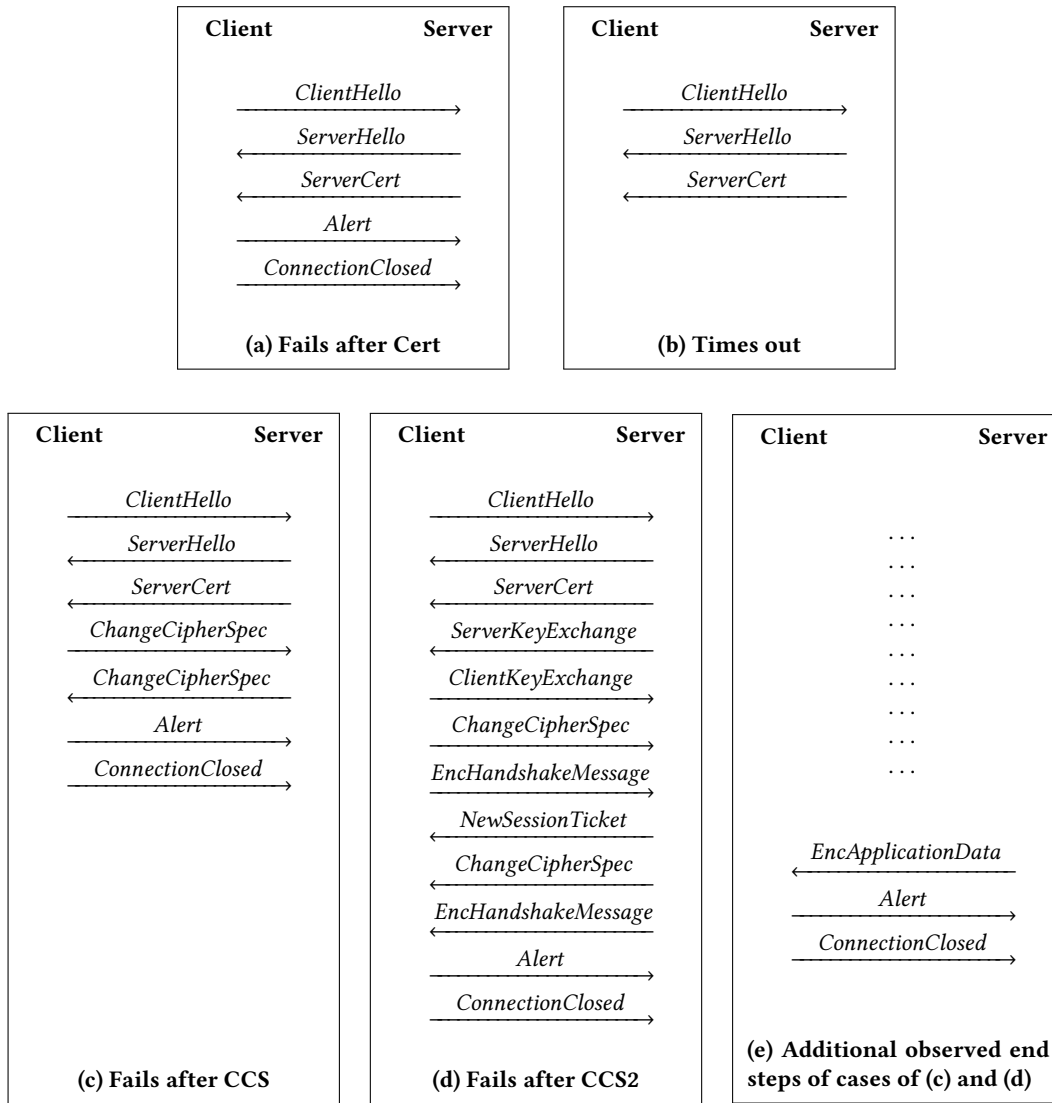
Figure 3: Ways in which a TLS connection may fail

(a) Our TLS proxy inspects the SNI contained in the Client Hello message of incoming TLS connections. This value is used to distinguish connections, and is also used as a lookup to chose which domain to redirect to.

(b) The Client Hello and all subsequent TLS handshake messages are forwarded to and from the server it looked up (which will be using the same certificate chain, but with a different hostname).

(c) If the app sends a TLS alert fail message or times out before encrypted application data was sent then we know the app refused to connect to the server with the wrong hostname.

(d) If the app does send encrypted application data then we know the TLS handshake was successful and the app will accept a certificate with the wrong hostname.

(e) If no more connections to test, finish. Otherwise, white list the current domain being tested, and repeat test with next domain.

(3) **Vulnerability Identification Stage**

For each connection that the app makes, which is found to succeed when proxied to the alternate domain, the following process is carried out to pinpoint the exact vulnerability.

(a) To establish whether the app accepts self-signed certificates and doesn't check the hostname (i.e. does not check the server's certificate is signed by a trusted CA). We redirect the TLS connection to https://self-signed.badssl.com[3].

---

[3]BadSSL.com is a site hosted by Google for basic testing of clients against incorrect TLS config.

The test fails if our tool detects a successful TLS connection.

(b) If the previous test fails, we know that the app either doesn't check the hostname and uses the devices trust store for signature verification, or, the app pins to a particular certificate in the trust chain but also fails to verify the hostname. To identify whether the former is the case, we redirect the traffic to a domain that uses an entirely different certificate chain.
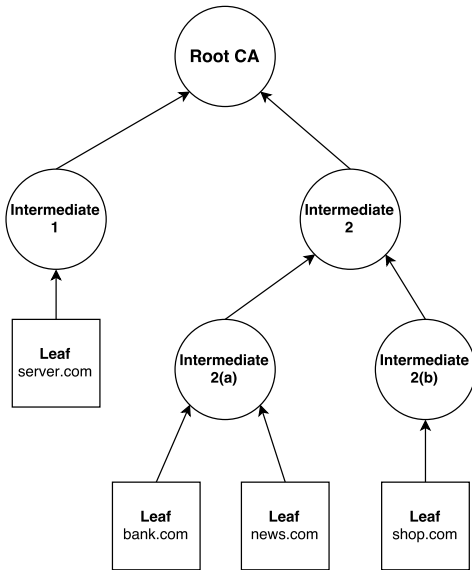
Figure 4: Typical hierarchical PKI, with four possible certificate chains

(c) If the app is found to pin and not verify the hostname, the tool executes additional tests to determine exactly which certificate in the chain is being pinned. This done by iteratively querying Censys for domains that have certificate chains that vary in intermediates, but share the same root. For example, consider Figure 4, which illustrates four different certificate chains originating from the same root CA. When testing an app that connects to bank.com, the first round of tests proxies the traffic to news.com, which we find is accepted. According to Figure 4, traffic would then be proxied to server.com to establish if the Root CA is pinned. If the test passes, we know the app must pin to either Intermediate 2 or Intermediate 2(a). To narrow down further, traffic would then be proxied to shop.com. If successful, then the app pins to Intermediate 2, if not then to Intermediate 2(a).

5 RESULTS

To try out our framework we tested 250 Android and 150 iPhone applications. We choose to look at categories of apps that are often considered to be high security. This included banking, trading, cryptocurrency and VPN apps. Such apps should be less prone to the simple kinds of attack that other frameworks can detect, such

as the acceptance of self-signed certificates. Furthermore, it seems more likely that these apps will pin their certificates (a precondition for this particular attack).

For each app category we considered, the top 20 apps by download numbers were installed and tested. The remaining apps were chosen by browsing through the appropriate categories in each of the app stores (Google Play and iOS). We downloaded apps from the US, UK, Belgium and Indian app stores. However, a significant portion of these are apps that are available globally.

We ran our framework, and manually opened each app in turn. Of the 400 apps, 24 failed our first round of test (i.e., transferred application data to the site we redirected them too). Of these, 6 were found to accept self-signed certificates; these were all banking apps from developing countries, some of which have been reported before [20].

Our framework then ran further tests on the remaining apps to discern whether they were implementing certificate pinning. This process is described in section 3(b) of the vulnerability identification stage. Of the 18 apps that only accepted certificates signed by a trusted CA, 9 were found to not pin and hence we deduced they would accept any valid certificate for any hostname.

The remaining 9 apps were all found to be pinning the intermediate or root certificate but not checking the hostname. For each of these apps, our framework then went on to determine the exact certificate being pinned. A summary of the apps that failed our tests, along with the certificates they pinned are shown in Figure 5. If we regard the HSBC family of apps as a single code base, then the distribution of apps that pin to intermediate certificates in contrast to those that pin root certificates, is approximately the same.

For apps that are found to pin the certificate but not check the hostname, an attacker could then go to the trusted third party used by the certificate and obtain a valid certificate in their own name. This certificate can then be used by the attacker to trick a victim's app into thinking it's communicating with the server it expects, and hence decrypt and/or modify any sensitive traffic.

More Prevalent Than Self-Signed Certificate Acceptance

Past investigations of certificate verification flaws in mobile applications, such as Fahl et al [10], found that the acceptance of any certificate (including those that are self-signed) to be the most prevalent of this category of vulnerabilities. In 2012, Fahl et. al analysed 13,500 apps from the Google Play store. They found that 790 accepted any certificate, and a further 284 accepted any CA signed certificate for any hostname. In contrast, from our scan of 400 apps, we find that although the total proportion of apps that contain TLS flaws has reduced, the distribution of the types of flaws that vulnerable apps do have has changed significantly. We find that in fact lack of hostname verification to now be the most common flaw, and the variation of this that arises to due the use of certificate pinning to be the second most common. Figure 6 shows all the other apps that were flagged as vulnerable by our tool, but were found not to pin to any certificate.

---

[4]This domain is used when setting up the app in the US, however is different for other countries.

App name	Platform	Certificate pinned	Certificate Type	Cost to exploit
TunnelBear VPN	Android	COMODO RSA Certification Authority	Intermediate	Free
Bank of America Health	Android	VeriSign Class 3 Public Primary Certification Authority - G5	Root	$366
Meezan Bank	Android	VeriSign Class 3 Public Primary Certification Authority - G5	Root	$366
Smile - the internet bank	Android	AddTrust External CA root	Root	Free
HSBC	iOS	Symantec Class 3 EV SSL CA - G3	Intermediate	$995
HSBC Business	iOS	Symantec Class 3 EV SSL CA - G3	Intermediate	$995
HSBC Identity	iOS	Symantec Class 3 EV SSL CA - G3	Intermediate	$995
HSBCnet	iOS	Symantec Class 3 EV SSL CA - G3	Intermediate	$995
HSBC Private	iOS	Symantec Class 3 EV SSL CA - G3	Intermediate	$995

Figure 5: Apps in our test set that pinned but lacked hostname verification

App name	Vulnerability	Platform
Emirates NBD	Self-signed	iOS
Kotak Bank	Self-signed	iOS
Al Rajhi Bank	Self-signed	iOS
Santander UK (biocatch)	No hostname check	iOS
CommBank Property	No hostname check	iOS
American Bank of Sydney	No hostname check	Android
Ulster Bank NI	No hostname check	Android
Ulster Bank RI	No hostname check	Android
BofAML Research Library	No hostname check	Android
First Financial Bank	No hostname check	Android
ACU Mobile	No hostname check	Android
Bitcoin.co.id	No hostname check	Android
Britline	Self-signed	Android
Opal Transfer	Self-signed	Android
Aman Bank	Self-signed	Android

Figure 6: Apps that additionally failed our tests.

App name	Domains connected to	Vulnerable?
TunnelBear	api.tunnelbear.com	✓
	stream.tunnelbear.com	✓
	s3.amazonaws.com	✓
BofA Health	benefitsolutions .bankofamerica.com	✓
Meezan Bank	mbanking.meezankbank.com	✓
Smile	public-05.p01cd18.monitise.eu	✓
HSBC	services.mobile.hsbc.com	✓
	mapp.us.hsbc.com [4]	✗
	security.us.hsbc.com [4]	✗
HSBC Business	www.hsbcnet.com	✓
	www.business.hsbc.co.uk	✗
	www.secure.hsbcnet.com	✗
HSBCnet	www.hsbcnet.com	✓
	www.secure.hsbcnet.com	✗
HSBC Private	services.mobile.hsbc.com	✓
	www.us.hsbcprivatebank.com	✗
HSBC Identity	www.business.hsbc.co.uk	✓
	www.hkg1vl0077.p2g .netd2.hsbc.com.hk	✗

Figure 7: Domains that are connected to

Pinning On The Rise

We took a random subset of 135 apps from our Android app data set. We then checked how many of these apps where using certificate pinning. This was done by adding our own root certificate to the phone's trust store, spoofing the DNS request and generating a certificate (on the fly) for the requested domain which is then used to MITM the connection. If the app accepts this then we conclude that it is not pinning.

Our experiments show that 38 out of those 135 apps implement certificate pinning, which is a much higher ratio than the results from Oltrogge et al. who found that 45 out of 639,283 apps used pinning in 2015 [18]. This increase could be attributed to a bias in our sample set, since it only includes high security apps; or it could be evidence of an increase in the use of pinning; or most likely, a combination of the two.

6 VULNERABILITY IMPACT

Having identified a number of apps that pin to intermediate or root certificates but miss the crucial step of hostname verification, our next steps were to evaluate the impact on security for each of these apps. An important consideration to be made is whether

sensitive information, such as log in credentials, is sent over these vulnerable connections. It may be the case that connections to the vulnerable domains only contain non-sensitive data such as adverts or analytics (this can be avoided to some extend by white listing common analytics domains such as crashlytics.com and webtrends.com, however is obviously not possible for all domains). We therefore, adopted the following methodology to determine the seriousness of the vulnerability.

- Run our framework on each app, making efforts to trigger as much functionality of the app as possible through manual interaction. This gives us a list of domains the app connects to and whether these connections are vulnerable to TLS Man-in-the-Middle.
- For each app with at least one vulnerable domain, we judge whether connections that involve the transmission of sensitive data, are made to the vulnerable domain. This is done

by checking whether the submission of random log-in credentials results in a connection attempt to that domain. If so, and the credentials are rejected, then we deem the app to be vulnerable to an attacker stealing these credentials.

Applying this methodology the all the apps that were detected by our tool (listed in figure 5), we find that TunnelBear VPN vulnerable connections to three domains, whilst Bank of America Health, Meezan Bank and Smile all made single connections to the vulnerable domains when submitting log in or registration details. Each app and the corresponding domains it made connections to is listed in figure 7.

Special Case: HSBC

The behaviour of HSBC's apps meant that its security impact analysis was more involved. On opening HSBC's main app, requests were made to the domain services.mobile.hsbc.com. We found that these connections were pinning to an intermediate certificate, but not checking the hostname. Despite this, as no interaction (other than opening the app) was made, we could not initially conclude that any sensitive data was being transmitted. Furthermore, when these connections were white-listed, and we were presented with the log in screen, a connection to mapp.us.hsbc.com was made. Additionally, any submitted log-in credentials were seemingly sent to another domain security.us.hsbc.com. Connections to these two latter domains were found to be secure i.e. passed our tool's tests. We did however find that they were not using certificate pinning, hence we were able to confirm the log-in details were sent to these non-vulnerable domains by installing a custom CA on the phone.

At this point one would deduce that users of the app are safe from having their credentials stolen by an attacker. However, it is possible that the app could be performing other sensitive operations, such as an update check. Moreover, the fact that pinning was being implemented solely for connections to this domain, suggested that in fact requests to this domain were of some significant importance.

We approached this problem by reverse engineering the iOS app. We searched for requests to the domain services.mobile.hsbc.com, and found that the app downloaded the resource /app/EntityList-1.5.17.xml. As the Android version was seemingly identical to the iOS version, including the same domains connected to (as well as pinning connections to the mobile.services.hsbc.com domain), we confirmed that this XML resource was also obtained, but named slightly differently: /app/EntityList-1.5.17-Android.xml.

Contained in this file are URLs for config files that we observed being downloaded after white listing connections to mobile.services.hsbc.com. We note that as the main HSBC app is available globally, the XML file contains different config URLs for each respective country where HSBC operates. A portion of this file is listed in the Appendix, Listing 3. Inspecting the contents of these config files, we can see that the app uses this for a number of things including: update checks, with included download links and corresponding checksum values (HSBC uses hot code push/replacement for some of its updates); miscellaneous app content like in app text, images and adverts; links to web resources that open in the mobile browser; and, critically, the domain to communicate with during the log-in process.

22 May 2017	Vulnerability discovered
23 May 2017	Initial disclosure by e-mail to HSBC
25 May 2017	Conference call discussing details of vulnerability
6 June 2017	We notice that an update to the "HSBC Business" app has added the vulnerability, we e-mail HSBC
3 August 2017	We send a follow up e-mail to check on progress and we contact the UK National Cyber Security Centre (NCSC) who also contact HSBC
4 August 2017	We receive an e-mail from HSBC saying they are working on the issue
31 August 2017	Follow up conference call with HSBC and NCSC. HSBC confirms the issue and say they are testing a fix
14 September 2017	Patch rolled out for main HSBC app

Figure 8: HSBC Disclosure Timeline

As a result, we conclude that by exploiting the pinning vulnerability for connections to services.mobile.hsbc.com, an attacker can intercept and modify the downloaded config file to force a victim to send authentication details to a domain under the attackers control.

Attack Scenario

To carry out this attack, the adversary needs to be in a position where they can man-in-the-middle the network traffic from a victim. A example of such a situation would be where the attacker is on the same WiFi network, such as an airport or coffee shop. Using ARP or DNS spoofing, the victims traffic can be redirected to the attacker. Alternatively, a victim could be tricked into connecting to 'fake' hotspot set up an attacker in what is known as an "evil twin" [1] attack. When the victim attempts to use their vulnerable app, the attacker can intercept the TLS handshake and provide the app with a certificate signed by the certificate that the app pins to. Figure 5 shows the cost attached to carrying out the attack for each app. Note that as hostname verification is not being done, only one of each certificate needs to be purchased. Therefore it would cost $995 to attack all of HSBCs vulnerable apps.

Disclosure

We have disclosed the vulnerabilities in the apps to all of the companies involved. The vulnerabilities in these apps have now all been fixed, or will be fixed before the publication of this paper. We received varying degrees of responsiveness from the organisations we disclosed to. Some patched their apps within a few days, whilst others took longer. The fastest responses were from Smile bank and from CommBank fixing their property app. The longest patch cycle was from HSBC, as shown in Figure 8. Fixing their app seemed to take longer due to the fact that the same app was available in 39

different countries' app stores, hence making testing the app much more complex. We would like to thank the UK's National Cyber Security Centre (NCSC) for helping us with the disclosure process.

7 WHY HOSTNAME VERIFICATION FAILS

To find why the vulnerable apps were weak, and to find out if simple static analysis methods could have found these vulnerabilities, we reverse engineered the vulnerable Android apps. We did this using the JADX tool[5] that can unpack Android APK files and decompile Smali to Java source code.

TunnelBear. On initial inspection the TunnelBear app seemed very secure. It did not disable hostname verification, and went further, defining a custom built hostname verifier to be used by all connections. The code for this verifier is shown in Figure 1. This test is applied automatically by the API, for all connections, and it seems to verify that the host connected to is one of `s3.amazonaws.com`, `amazonaws.com`, `tunnelbear.com` or `captive.apple.com`. However, on closer inspection we found that this is not actually how the verifier method should work.

The verifier method is called by the API with a string giving the hostname the user has requested a connection with, and an active SSL Session object. The verifier should then check that the given string matches the name of the host that the SSL Session connects to. The tunnelBear code does not do this: instead it simply checks the value of the string given to it and ignores the SSL Session object. If this app tries to connect to "tunnelbear.com", and we redirect it to "evil.com", the verifier will be called with str equal to "tunnelbear.com" and an SSL Session object that connects to "evil.com". The verifier will just check the string, find it does equal an expected string, ignore the session object and log the connection as trusted.

This represents an understandable confusion about how the API works. A quick manual inspection could easily miss this subtle error, and simple static analysis tools would also miss this: a verifier is present, and hostname checking is enabled. To tell that the hostname string and SSL Session object were not correctly compared would require complex information flow checking, and even without a direct comparison, code could still be secure, if it for instance checked the SSL Session object against a static string.

The Smile Banking App. Inspecting the Smile banking app we find that it does not specify a custom built hostname verifier. It does however set the flag `STRICT_HOSTNAME_VERIFIER` which requires that hostnames match exactly. Again, on initial inspection the app looks secure. We identified the code that opened the weak connection, and it is shown in Figure 2.

This code uses the `SSLSocketFactory` object from the Apache HttpClient Java Library to create a SSL connection. As Georgiev et al. [13] have previously pointed out, this object only checks the hostname when the protocol is set to be HTTPS or LDAP by, for instance, using the `HttpsURLConnection` object. In this code the hostname and port number are resolved into an Internet address on line 5, and the raw socket connects to this address. Therefore this code does not check the hostname, irrespective of any flags set.

The setting of the `STRICT_HOSTNAME_VERIFIER` flag in the Smile banking app suggests the authors of this app were aware of the issue of hostname verification, however they had misunderstood the subtleties of the API. After our disclosure the company fixed this error by explicitly adding a verifier object. This could also have been fixed by setting the protocol to HTTPS or checking the hostname manually. Because of the range of ways to use `SSLSocketFactory` safely, we currently do not know of any static checker that could detect these issues without a high false negative result.

Bank of America Health. This app follows the same pattern as the Smile app: it uses the Apache HttpClient Java Library `SSLSocket Factory` with a raw internet address. It does define a hostname verifier object, which would have correctly verified the Bank of America hostname, however because the protocol of the connection is not set to HTTPS and the verifier is not explicitly invoked, this code never runs.

Meezan Bank. On inspecting this code we found that it used the `HttpsURLConnection` object that would normally check the hostname. However, the `ALLOW_ALL_HOSTNAME_VERIFIER` flag had been explicitly set, which disables all hostname verification. This one vulnerability could have been found by static checking, because there is no safe way to use this flag, therefore its presence in the code indicates a weakness.

Discussion

Reverse engineering these vulnerable apps has revealed a number of interesting and subtle misuses of varying APIs that are available for implementing certificate pinning. These errors contrast significantly with those discovered in previous studies. Apps failing to verify certificates in the study by Fahl et. al [10] were found to be doing so in glaringly obvious ways, such as disabling certificate verification all together. This indicated a fundamental misunderstanding of the importance of server authentication in TLS.

Our results demonstrate that this is no longer the case, the vast majority of apps no longer have these basic errors. However, the added complication of certificate pinning, which appears to be on the rise, has spawned a new class of vulnerabilites. These are more subtle and hence have not been detected by existing detection techniques. Moreover, our findings are testament to the feedback received from developer interviews carried out in [18], which found that although general understanding of pinning is good, implementation complexity has made it difficult to roll out.

Clearly, the abundance of pinning implemenation options available to developers has played a role in causing these flaws to be made. Platform providers can make this less of an issue by providing standardised implementations with clear documentation. To this end, Google have introduced Network Security Configuration[6] in the Android 7.0 SDK. This provides a easy way to configure certificate verification, including the ability to specify certificate pins and associated hostnames in an XML file. If app developers make use of these standard implementations, instead of rolling out their own or using 3rd party libraries, these errors will be much less likely to occur.

---

[5]https://github.com/skylot/jadx

[6]https://developer.android.com/training/articles/security-config.html

```
1        public final boolean verify(String str, SSLSession sSLSession) {
2            if (str.contains("s3.amazonaws.com")) {
3                log("BearTrust", "Regular trust enabled");
4                return true;
5            } else if (str.contains("amazonaws.com") && !str.contains("s3.amazonaws.com")) {
6                log("BearTrust", "API Gateway enabled");
7                return true;
8            } else if (str.contains("captive.apple.com") || str.contains("tunnelbear.com")
9            || ("https://" + str + "/").equals("https://api.tunnelbear.com/") {
10               log("BearTrust", "BlueBear trust enabled");
11               log("BearTrust", "BlueBear enabled, trying IP");
12               return true;
13           } else {
14               logError("BearTrust", "Failed to verify hostname");
15               return false;
16           }
17       }
```

Listing 1: Reverse Engineered Code from the TunnelBear app

```
1        public Socket connectSocket(Socket socket, String hostname, int portNo,
2        InetAddress inetAddress, int i2, HttpParams httpParams) {
3            int connectionTimeout = HttpConnectionParams.getConnectionTimeout(httpParams);
4            int soTimeout = HttpConnectionParams.getSoTimeout(httpParams);
5            SocketAddress inetSocketAddress = new InetSocketAddress(hostname, portNo);
6            socket = socket != null ? (SSLSocket) socket : (SSLSocket) createSocket();
7            if (inetAddress != null) {
8                if (i2 < 0) { i2 = 0; }
9                socket.bind(new InetSocketAddress(inetAddress, i2));
10           }
11           socket.connect(inetSocketAddress, connectionTimeout);
12           socket.setSoTimeout(soTimeout);
13           return socket;
14       }
```

Listing 2: Reverse Engineered Code from the Smile Banking app

8 CONCLUSION

In this paper we have presented a new methodology for dynamically testing applications that use TLS but fail to verify the hostname. We have argued that previous tools being used by pen testers to detect this flaw are insufficient, as the use of certificate pinning hides the vulnerability. We implemented our proposed technique and then used our tool to test hundreds of high security apps, including banking, trading, VPN and cryptocurrency apps. Our tests find that apps from some of the world's largest banks contain the flaw, which if exploited, could enable an attacker to decrypt, view and modify traffic (including log in credentials) from the users of the app.

REFERENCES

[1] K. Bauer, H. Gonzales, and D. McCoy. Mitigating evil twin attacks in 802.11. In *Performance, computing and communications conference, 2008. IPCCC 2008. IEEE International*, pages 513–516. IEEE, 2008.

[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] K. Bhargavan, C. Fournet, A. D. Gordon, and G. O'Shea. An advisor for web services security policies. In *Proceedings of the 2005 workshop on Secure web services*, pages 1–9. ACM, 2005.

[4] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 98–113. IEEE, 2014.

[5] 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.

[6] T. Chothia, F. D. Garcia, C. Heppel, and C. McMahon Stone. Why Banker Bob (still) Can't Get TLS Right: A Security Analysis of TLS in Leading UK Banking Apps. In *Financial Cryptography and Data Security*. Springer, 2017.

[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] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A search engine backed by internet-wide scanning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 542–553. ACM, 2015.

[9] D. Eastlake et al. Transport layer security (TLS) extensions: Extension definitions. 2011.

[10] 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.

[11] D. Fisher. Final report on diginotar hack shows total compromise of ca servers. *Retrieved September*, 8:2013, 2012.

[12] C. Forum. Guidelines for the issuance and management of extended validation certificates, version 1.5.7, 2015.

[13] 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.

[14] T. Guardian. Bankers fail to censor thesis exposing loophole in bank card security, 30/12/10, 2010.

[15] G. Keizer. Hackers may have stolen over 200 ssl certificates. *Computerworld*, 2011.

[16] M. Moixe. New tricks for defeating ssl in practice. In *BlackHat Conference, USA*, 2009.

[17] B. Möller, T. Duong, and K. Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback, 2014.

[18] 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.

[19] H. Perl, S. Fahl, and M. Smith. You won't 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.

[20] 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.

[21] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and J. Suman. HVLearn: Automated Black-box Analysis of Hostname Verification in SSL/TLS Implementations. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017.

[22] R. Sleevi. Intent to deprecate and remove: Trust in existing symantec-issued certificates. *Chromium development forum*, 2017.

[23] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. SMV-Hunter: Large scale, automated detection of SSL/TLS Man-in-the-Middle vulnerabilities in Android apps. In *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. Citeseer, 2014.

APPENDIX

```xml
<?xml version="1.0" encoding="UTF-8"?>
<entityList>
<entity id="148" headerImg="default" hometype="app">
<supportedDevices>
<device name="iPhone"/>
<device name="iPad">
<configurl>https://www.hsbc.co.uk/content_static/
tablet/1/5/17/1/config.json</configurl>
</device>
</supportedDevices>
<supportedSSLPinning>
<enable>false</enable>
</supportedSSLPinning>
<localised locale="en" name="United_Kingdom" shortname="UK" default="true"/>
<configurl>https://www.hsbc.co.uk/content_static/mobile/1/5/17/1/config.json</configurl>
</entity>
<entity id="159" headerImg="default" hometype="app">
<supportedDevices>
<device name="iPhone"/>
<device name="iPad">
<configurl>https://www.hsbc.gr/1/PA_esf-ca-app-content/
content/mobile/GR/tablet/1/5/17/0/config.json</configurl>
</device>
</supportedDevices>
<supportedSSLPinning>
<enable>false</enable>
</supportedSSLPinning>
<localised locale="en" name="Greece" shortname="GR" default="true"/>
<localised locale="el" name="Greece" shortname="GR" default="false"/>
<configurl>https://www.hsbc.gr/1/PA_esf-ca-app-content/content/
mobile/GR/mobile/1/5/17/0/config.json</configurl>
</entity>
<entity id="144" headerImg="default" hometype="app">
<supportedDevices>
<device name="iPhone"/>
<device name="iPad">
<configurl>https://mapp.us.hsbc.com/1/PA_1_083Q9FJ08A002FBP5S00000000/
content/usshared/Mobile/tablet/HBUS_1-5-TabletApp_Jun2015_PROD.js</configurl>
</device>
</supportedDevices>
<supportedSSLPinning>
<enable>false</enable>
</supportedSSLPinning>
<localised locale="en" name="United_States" shortname="US" default="true"/>
<configurl>https://mapp.us.hsbc.com/1/PA_1_083Q9FJ08A002FBP5S00000000/content/
usshared/Mobile/Native/HBUS_1-5-PhoneApp_Jun2015_PROD.js</configurl>
</entity>
</entityList>
```

Listing 3: XML Config file downloaded over vulnerable HSBC connections