A-SIT

Zentrum für sichere Informationstechnologie - Austria

Prävalenz von HTTP in aktuellen Android-Anwendungen

A-SIT

Prävalenz von HTTP in aktuellen Android-Anwendungen

Autor:
Florian Draschbacher:
florian.draschbacher@iaik.tugraz.at

Datum: 25.01.2023

Abstract/Zusammenfassung:
Im Gegensatz zum Web-Ökosystem, wo sich nach anfänglichen Schwierigkeiten die Verschlüsselung von HTTP-Verbindungen mittels TLS mittlerweile weitgehend durchgesetzt hat, verwenden einige Android-Anwendungen noch immer unverschlüsselte HTTP-Verbindungen. Problematisch ist dies besonders in Anwendungen, die sensible Nutzerdaten (etwa Account-Daten) an Server schicken. Ein Man-In-The-Middle-Angreifer kann hier nahezu mühelos unbemerkt Daten abgreifen, etwa mithilfe von ARP-Spoofing oder Evil-Twin-Attacken.

Im Rahmen dieses Forschungsprojektes beleuchten wir die allgemeine Problematik, und beschreiben insbesondere die Situation von Apps für das mobile Betriebssystem Android. Hierzu wird ein Überblick über relevante wissenschaftliche Studien aus der Vergangenheit geboten, bevor wir Ergebnisse aus unseren eigenen Untersuchungen an einem Datensatz aktueller populärer Anwendungen vorstellen und diskutieren.

Inhalt

1. Einleitung

Als das World Wide Web zu Beginn der 1990er-Jahre konzipiert wurde, waren primär Anwendungsfälle angedacht, die den Konsum von Informationen zum Inhalt hatten. Der Datenfluss sollte also vor allem von einem zentralen Speicherpunkt (Server) an den Konsumenten (Client, oft ein Web Browser) am Endnutzer-Gerät laufen. Konzeptionell folgte der Aufbau damit dem von Bibliotheken, die Jahrhunderte lang die etablierteste Form eines Informationsspeichers gewesen waren. Schon nach kurzer Zeit zeigte sich allerdings, dass die neue Technologie sich auch bestens eignete, komplett neue Paradigmen zu verwirklichen, in denen die vom Server ausgelieferten Informationen dem jeweiligen Benutzer angepasst waren. Dafür wurden nun mitunter als Teil der Kommunikation auch personenbezogene Daten vom Nutzer an den Server geschickt. Bald war klar, dass manche Anwendungsfälle, etwa die Abwicklungen von Geldgeschäften, den Austausch von Informationen erforderten, die um keinen Preis in die Hände Dritter gelangen durften.

Allmählich entstand nun ein Bewusstsein dafür, dass das HTTP-Protokoll zwischen Browser und Server keinerlei Schutz gegen Dritte bot. Selbst wenn keine sensiblen Daten vom Nutzer an den Browser übermittelt werden, stellt schon alleine die Information über den Zugriff auf eine bestimmte Seite potentiell eine sensible Information dar, die im Sinne der Privatsphäre vor den Blicken von Beobachtern geschützt werden sollte. Angreifern war es darüber hinaus nicht nur trivial möglich, den Datenaustausch mitzulesen und so an Informationen des Nutzers zu kommen, sondern auch gezielt falsche Daten einzuschleusen.

Um diese erheblichen Sicherheitsprobleme zu beheben, wurde schließlich eine abgesicherte Version des HTTP-Protokolls zur Kommunikation zwischen Browser und Server entwickelt. Das neue Protokoll mit dem Namen HTTPS verfügt über eine zusätzliche Sicherheitsschicht, die Secure Socket Layer (SSL) bzw. später Transport Layer Security (TLS) getauft wurde. Auf Basis von Kryptografie bietet diese zusätzliche Schicht die Möglichkeit, die Identität eines Servers eindeutig festzustellen und die Kommunikation gegen Einsicht oder Manipulation durch Dritte abzusichern.

Um von den Sicherheitsgarantien des HTTPS-Protokolls profitieren zu können, mussten allerdings Server und Client entsprechend angepasst werden. Nur wenn sowohl Server als auch Client HTTPS unterstützen, und der Client den Verbindungsaufbau über dieses Protokoll startet, ist auch die weitere Kommunikation über die Verbindung tatsächlich geschützt.

Nachdem die Verfügbarkeit und Nutzung von HTTPS während der ersten Jahre nach dessen Entwicklung nur langsam anstiegen, gelang es schließlich, das notwendige Sicherheits-Bewusstsein bei einem erheblichen Teil von Browser-Nutzern zu schaffen. Dies gelang unter anderem durch die klare Hervorhebung von sicheren Verbindungen in der Benutzeroberfläche der gängigen Webbrowser, sowie durch gezielte Warnungen bei ungesicherten Verbindungen.

Während so allmählich das ursprünglich adressierte Problem in traditionellen Server-Browser-Konfigurationen weitgehend gelöst werden konnte, hatte sich durch die Einführung der Smartphones in der Zwischenzeit allerdings ein zusätzliches Problem ergeben. Diese hatten sich immer mehr als jederzeit verfügbare Möglichkeit zum Internet-Zugriff etabliert. Im Unterschied zu klassischen Anwendungsfällen diente aber häufig nicht mehr der Browser als Schnittstelle zwischen Nutzer und Server, sondern applikations-spezifische Client-Anwendungen, die eine nahtlosere Integration von Web-Diensten in mobile Betriebssysteme ermöglichten.

Weil diese Client-Anwendungen von den jeweiligen Dienste-Anbietern bzw. Website-Betreibern entwickelt und angeboten werden, ist es hier deutlich schwieriger, flächendeckend ein Upgrade auf HTTPS zu erwirken. Für Nutzer gibt es keine Möglichkeit, festzustellen, ob eine App mit ihrem Backend-Server über eine abgesicherte Verbindung kommuniziert oder nicht.

Dieser Bericht soll mittels Zusammenfassung wissenschaftlicher Veröffentlichungen und eigener Untersuchungen beleuchten, wie hoch unter aktuellen Android-Anwendungen der Anteil an Apps mit unsicheren Verbindungen ist.

2. Hintergrund

In diesem Abschnitt sollen jene Technologien erklärt werden, die für das weitere Verständnis der späteren Ausführungen notwendig sind.

2.1. HTTP

Das HyperText Transfer Protocol wurde von Tim Berners-Lee als Teil des World Wide Webs 1990 entwickelt und bildet heute eine der zentralen Säulen des modernen Internets. Es dient der Kommunikation zwischen einem Client, der in der Regel Daten anfordert, sowie einem Server, der Anfragen bearbeitet und beantwortet. Gegenstand der Übertragung sind meistens HTML-Inhalte, die von Webbrowsern angezeigt werden. Im Falle von applikations-spezifischen Client-Anwendungen läuft die Kommunikation häufig mit einem Web-Server, der über HTTP eine REST-Schnittstelle zur Verfügung stellt. Im weitesten Sinne handelt es sich dabei um eine Möglichkeit zum Zugriff auf eine Datenbank, die häufig mit einer Authentifizierung versehen ist, sodass das genaue Ausmaß des Datenzugriffs für jeden Nutzer individuell kontrolliert werden kann.

Aus technischer Sicht handelt es sich beim HTTP-Protokoll um ein Protokoll auf Anwendungs-Schicht im Protokoll-Stack. Unterhalb von HTTP befindet sich das TCP-Protokoll, das wiederum auf das IP-Protokoll aufbaut.

2.2. TLS/SSL

TLS ist ein kryptografisches Protokoll, das im HTTPS-Stack verwendet wird, um die Vertraulichkeit, Integrität und Authentizität einer Kommunikations-Verbindung zwischen einem Server und Client sicherzustellen. Vertraulichkeit bezeichnet hier, dass der Inhalt der Kommunikation von Dritten nicht entschlüsselt werden kann. Integrität bedeutet, dass eine Nachricht von Dritten am Weg zwischen den beiden Kommunikationsteilnehmern nicht manipuliert werden kann. Mit Authentizität ist im Falle von HTTPS meist gemeint, dass sich der Client über die Identität des Servers sicher sein kann. Es ist einem Angreifer also nicht möglich, die Identität des Servers anzunehmen und damit die Kommunikation, die für den legitimen Server gedacht ist, zu übernehmen.

2.3. HTTPS

Anders als der Name impliziert, handelt es sich bei HTTPS aus technischer Sicht nicht um eine eigenständige sichere Alternative zum HTTP-Protokoll. Vielmehr bezeichnet HTTPS die Verwendung des bestehenden HTTP-Protokolls über einer abgesicherten Schicht des Transport Layer Security Protokolls.

Um von den Sicherheitseigenschaften von HTTPS bzw. der TLS-Schicht profitieren zu können, muss nicht nur der Server das Protokoll unterstützen, sondern der Client auch eine entsprechende HTTPS-URL

verwenden, um den Kontakt zum Server aufzunehmen. Im Falle eines Webbrowsers als Client bedeutet das, dass der Nutzer eine entsprechende HTTPS-URL in die Adress-Leiste des Programms eintragen muss bzw. einem Link auf eine solche URL folgen muss.

Von der Vorstellung der ersten öffentlichen SSL-Spezifikation 1999 [1] dauerte es einige Jahre, bis HTTPS breite Verwendung fand. Noch 2017 lagen Schätzungen für die Verbreitung von HTTPS-Unterstützung bei Web-Servern bei rund 50% [2]. Messungen in populären Webbrowsern ergaben zur gleichen Zeit, dass auch etwa die Hälfte aller Serververbindungen über HTTPS liefen. Während sich die serverseitige Unterstützung für HTTPS seither massiv verbesserte, liefen immer noch viele Verbindungen über ungesicherteres HTTP, da noch immer viele alte Links auf HTTP-URLs verwiesen und keine Weiterleitung auf HTTPS implementiert war [3]. Um dieses Problem in den Griff zu bekommen, wurden verschiedene Lösungen vorgestellt, mit denen Webbrowser bei Vorliegen von HTTPS-Unterstützung des Servers clientseitig ein automatisches Upgrade von URLs vornahmen.

Handelt es sich beim Client-Programm um eine applikations-spezifische Client-Anwendung, dann muss die HTTPS-URL vom App-Entwickler entsprechend hinterlegt werden. Eine flächendeckende Lösung zum automatischen Upgrade ist aufgrund der Vielfalt der Client-Implementierungen hier im Allgemeinen nicht möglich.

3. Situation unter Android

3.1. Hintergrund

Android, ein quelloffenes Software-Projekt von Google, ist das meistverbreitete Betriebssystem für mobile Geräte wie Smartphones und Tablets. Aufgrund ihrer guten Transportierbarkeit werden diese Produkte besonders häufig für den mobilen Internetzugriff genutzt. In vielen Fällen findet der Zugriff auf Web-Server hier nicht über einen Browser statt, sondern über eine native Client-Anwendung, die besonders eng in den Rest des Betriebssystems integriert werden und so im Vergleich zu einer Web-Site ein besseres Nutzererlebnis bieten kann.

Um auf der Android-Plattform das Problem von ungesicherten HTTP-Verbindungen in den Griff zu bekommen, führte Google ab Android 6.0 nach und nach neue Sicherheitsfunktionen für App-Entwickler ein. Als ersten Schritt handelte es sich um das usesCleartextTraffic-Flag im Applikations-Manifest (hier teilen Entwickler dem System verschiedene Eigenschaften ihrer App mit). Wurde das Flag deaktiviert (und damit die Sicherheitsfunktion aktiviert), so verhinderte das System, dass die App in ihrer Server-Kommunikation HTTP verwendet. Aufgrund der mangelnden Granularität konnte diese Sicherheitsfunktion allerdings nur genutzt werden, wenn der Entwickler sicherstellen konnte, dass die App zu keiner Zeit auf einen Server zugreifen muss, der kein HTTPS unterstützt. Dies schloss etwa Apps aus, die für nicht sicherheits-relevante Server-Zugriffe bewusst HTTP einsetzten. Außerdem wurde das Flag per Default aktiviert, sodass es des Wissens und der Initiative des Entwicklers bedurfte, um die Sicherheitsfunktion zu aktivieren.

Mit Android 7.0 wurde daher eine XML-basierte Konfigurationsdatei namens Network Security Configuration (NSC) eingeführt, die Entwicklern erlaubt, unkompliziert applikations-weit eine genaue Policy für die Netzwerkverbindungen einer App zu konfigurieren. Damit ist es möglich, HTTP im Allgemeinen zu unterbinden, aber für ausgewählte Domänen zu erlauben.

Ab Android 9.0 werden alle HTTP-Verbindungen schon in der Default-Konfiguration unterbunden, sodass Entwickler zwangsläufig mit der Sicherheitsfunktion konfrontiert werden, wenn sie eine HTTP-Verbindung

verwenden möchten. Um die Rückwärtskompatibilität beizubehalten, gilt der neue Default allerdings nur für Apps, deren Entwickler im Applikations-Manifest Android 9 oder neuer als Target API vermerken. Um trotz dieser Einschränkung einen flächendeckenden Effekt zu erzielen, erzwingt Google für Apps im Play Store, dass eine aktuelle Target API anvisiert wird.

3.2. Forschung

Als Teil des immer breiteren wissenschaftlichen Interesses für Themen der mobilen Sicherheit entstanden in jüngerer Vergangenheit auch Studien, die sich der Analyse der Sicherheit von Client-Anwendungen widmen. Dabei zeigte sich, dass die Verbreitung von HTTPS hier nur sehr langsam voranschreitet. Ren et al. fanden 2018 anhand einer Analyse von Versionsverläufen von Android-Apps heraus, dass es für eine bestimmte Web-Domäne 5 Jahre ab serverseitiger Unterstützung für HTTPS dauerte, bis die Hälfte aller auf sie zugreifenden Apps dies mittels HTTPS taten [4]. Über alle Versionen hinweg setzten fast 70% aller Anwendungen für ihre Kern-Funktionalität HTTP-Verbindungen ein. Als Grund vermuten die Forscher, dass Entwickler etwa beim Aktualisieren der Bibliotheks-Abhängigkeiten ihrer Software-Projekte zu zögerlich sind. Piccolboni et al. [5] berichteten 2020, dass etwa ein Drittel von 1780 automatisiert untersuchten Android-Anwendungen HTTP-Verbindungen aufbauten.

Seit der Einführung des NSC-Systems wurden auch dessen Effekte untersucht. Possemato et al. [6] kamen dabei 2020 zu dem Schluss, dass trotz Einführung des NSC-Systems und Erzwingung von HTTPS mittels Target API auf Google Play noch immer 67% der untersuchten Anwendungen HTTP-Kommunikation erlaubten. Die Forscher fanden auch heraus, dass manche Anbieter für In-App-Werbung App-Entwickler gezielt dazu anleiten, HTTP global zu erlauben, weil Werbeanzeigen teils über ungesicherte Verbindungen ausgeliefert wurden. In einer ähnlichen Studie aus 2021 kamen auch Oltrogge et al. [7] zur Erkenntnis, dass knapp 90 % der Apps, die eine NSC definierten, dies taten, um die Sicherheit der Default-Konfiguration zur schwächen.

4. Untersuchung

Die aktuellsten Studien zur Prävalenz von HTTP-Verbindungen in Android-Anwendungen liegen schon mehrere Jahre zurück. Da das Forschungsgebiet Mobile Sicherheit besonders schnelllebig ist, ist davon auszugehen, dass die damals erhobenen Daten nicht mehr der aktuellen Wirklichkeit entsprechen. Aus diesem Grund wiederholen wir ähnliche Untersuchungen an einem Datensatz aus aktuellen populären Android-Anwendungen.

4.1. Datensatz

Die hier untersuchten Android-Anwendungen wurden 2022 vom Google Play Store gesammelt. Es handelt sich dabei um die in Österreich beliebtesten 100 Anwendungen aus 33 verschiedenen Kategorien. Mittels eines selbstentwickelten Hilfsprogramms wurden für jede Anwendung alle APK-Files für die x86-Plattform heruntergeladen. Weil einige Anwendungen mit dieser Plattform nicht kompatibel waren, wurden in Summe 2697 Anwendungen heruntergeladen.

4.2. Test-Setup und Methodik

Aufgrund des Umfangs unseres Datensatzes wurde der erste Schritt der Analyse automatisiert durchgeführt. Dazu konstruierten wir ein Testbed basierend auf dem Android-Emulator. Jede Anwendung

wurde am Emulator ausgeführt und durch zufälligen Benutzer-Input gesteuert. Der Emulator war hierzu so konfiguriert, dass jede Netzwerk-Verbindung über einen Proxy-Server lief. Dieser wurde auf Basis des quelloffenen mitmproxy-Programms[1] so aufgesetzt, dass alle unsicheren HTTP-Verbindungen in einem Log-File gespeichert wurden. Von den 2697 heruntergeladenen Anwendungen waren 1604 in unserem Testbed lauffähig.

Die Analyse wurde auf einem bare-metal-virtualisierten Ubuntu 20.04 LTS System mit 128 CPU-Kernen und 128 GB RAM ausgeführt, das auf einem Dual AMD EPYC 7502 CPU mit 1 TB RAM lief. Dieses Setup erlaubte es, 6 Instanzen unseres Analyse-Testbeds parallel auszuführen. Jede Instanz erhielt nicht nur ein eigenes virtuelles Android-Gerät im Emulator, sondern auch einen eigenen Proxy-Server. So konnte sichergestellt werden, dass der Netzwerkverkehr jeweils einer ausgeführten App zugeordnet werden konnte.

Auf Basis des Log-Files wurde schließlich eine zweite automatisierte Analyse zu den gefundenen Web-Servern durchgeführt. Hierzu wurde für jeden über HTTP kontaktierten Server festgestellt, ob auch eine HTTPS-Verbindung möglich gewesen wäre. Bei erfolgreichem Verbindungsaufbau wurde überprüft, ob der Server auch ein gültigen TLS-Zertifikat verwendet, das von einer Certificate Authority signiert ist.

Als letzten Schritt unserer Analyse untersuchten wir auch die Network Security Configurations der Apps in unserem Datensatz. Hierzu wurden zwei Python-Skripts entwickelt, die die entsprechende XML-Datei aus einem APK-Paket extrahieren und anschließend analysieren können.

4.3. Prävalenz von HTTP

Unsere Untersuchung zeigt, dass 163 Apps in unserem Datensatz (10 %) ungeschützte HTTP-Verbindungen zu 144 verschiedenen Hosts aufbauen. Alle Hosts, die von 3 oder mehr verschiedenen Apps über HTTP kontaktiert wurden, sind in **Tabelle 1** Am häufigsten über HTTP kontaktierte Hosts zusammengefasst.

Der am häufigsten kontaktierte Host ip-api.com dient der Geolokalisierung von Nutzern anhand ihrer IP-Adresse. Dass hier nicht HTTPS verwendet wird, ist aus Sicht des Datenschutzes unbedenklich. Auch bei Nutzung von HTTPS können die IP-Adressen der beiden Kommunikationsteilnehmer nicht geschützt werden, da sie schon für den Aufbau der TLS-Verbindung benötigt werden. Dennoch ist hier im Android-Kontext die Verwendung von HTTP problematisch, da sie den Entwickler zur Lockerung der Default-NSC zwingt. Ist der Entwickler hier nicht vorsichtig, aktiviert er unter Umständen den Cleartext-Verkehr für die gesamte Anwendung.

Host	Anzahl Apps
ip-api.com	14
cdn-api.admost.com	8
med-api.admost.com	7
link.mobihealthplus.com	7
player.vimeo.com	3
onex-28c2.kxcdn.com	3
4dwallpapers.com	3
examcontrol.com	3
superpowered.com	3

Tabelle 1 *Am häufigsten über HTTP kontaktierte Hosts*

[1] https://mitmproxy.org

Die beiden an der Nutzungs-Häufigkeit gemessen nächsten Hosts gehören zu AdMost, einem Anbieter für Werbe-Mediation. Dabei handelt es sich um einen Werbe-Vermittler, dessen Aufgabe es ist, den App-Anbietern jene Werbeanzeigen zu beschaffen, die im Moment am besten bezahlt werden. In der Dokumentation des AdMost-Android-SDKs[2] beschreibt der Hersteller im Detail, wie Entwickler eine unsichere Network Security Configuration einrichten müssen, um AdMost nutzen zu können. In der Branche ist es nicht unüblich, dass nutzerbezogene Daten an den Werbe-Anbieter übermittelt werden, um mit personalisierter Werbung den Profit zu maximieren. Dass hier eine ungesicherte Verbindung genutzt wird, ist bedenklich.

Vimeo ist ein Video-Portal, auf das Android-Anwendungen über einen eigens angebotenen Software Development Kit zugreifen können. Weshalb hier ein Zugriff über HTTP erfolgt, ist unklar. Es kann vermutet werden, dass hier App-Entwickler eine veraltete Version des SDKs verwenden. In der aktuellen Version konnten wir keinen Bedarf für HTTP (und entsprechende Änderungen an der NSC) feststellen.

Bei Superpowered handelt es sich um einen Anbieter für eine Software-Bibliothek zur Audioverarbeitung. Es kann vermutet werden, dass die Bibliothek entweder als Lizenz-Check oder zur Sammlung von Analysedaten Serverkontakt aufnimmt. Ob der Kontakt sensible Daten überträgt, wurde hier nicht untersucht. Da die Verbindung aus nativem Code heraus aufgebaut wird, wird die NSC ignoriert. Sie muss also vom App-Entwickler nicht deaktiviert werden.

Alle anderen häufig kontaktierten Hosts betreffen die Kern-Funktionalität der jeweiligen Apps und sind deren Entwicklern zuzurechnen. Im Ranking vertreten sind diese Hosts, weil mehrere Apps desselben Entwicklers auf sie zugreifen.

4.4. Verfügbarkeit von HTTPS

Von den 144 verschiedenen über HTTP kontaktierten Hosts konnten 128 (89 %) auch in unserem automatisierten Test-Programm erreicht werden. Auf 109 dieser Hosts (85 % der über HTTP erreichbaren Hosts) war auch ein HTTPS-Server erreichbar (offener Server-Socket auf HTTPS-Standard-Port 443). Von diesen verfügten 84 (77 %) auch über ein gültiges TLS-Zertifikat, so dass ohne Probleme eine Verbindung hergestellt werden konnte. Abbildung 1 illustriert die Aufteilung der Hosts nach dem Grad der HTTPS-Unterstützung.

[2] https://admost.github.io/amrandroid/

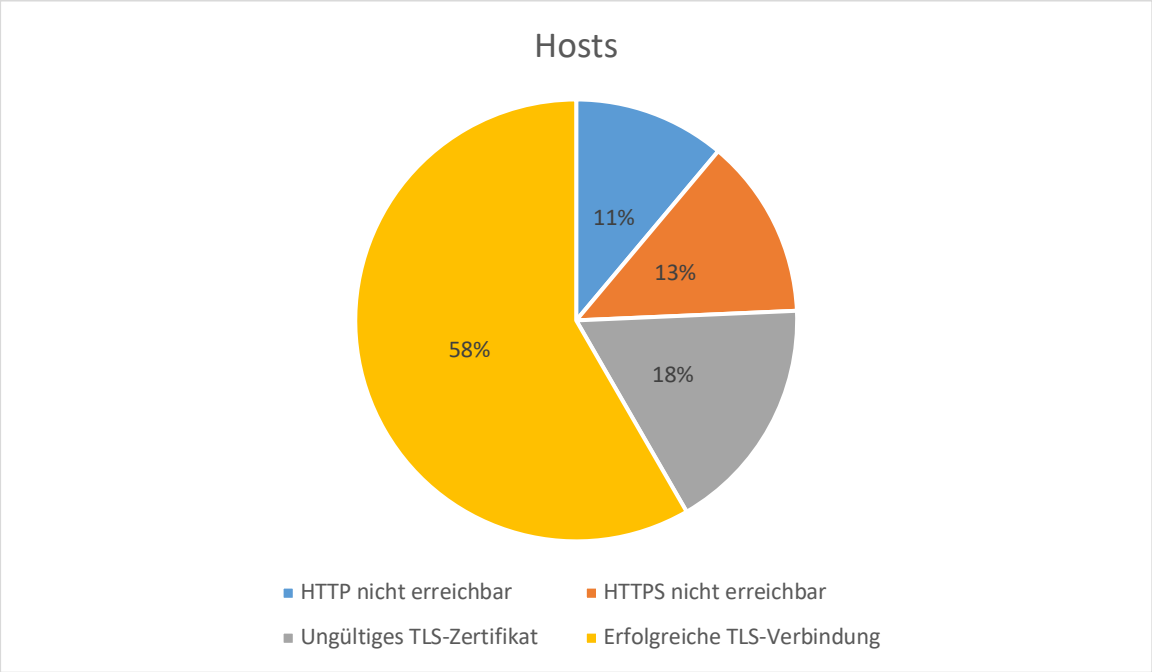

Abbildung 1 Über HTTP kontaktierte Hosts nach HTTPS-Kompatibilität

4.5. Untersuchung der Network Security Configurations

40,5 % der 2697 Anwendungen in unserem Datensatz nutzten eine eigene Network Security Configuration. Von den verbliebenen 1603 Apps spezifizierten 1534 (96 %) ein Target SDK, das die Nutzung von HTTP in der Standard-Konfiguration verbietet. 44,9 % der Apps mit NSC verwendeten eine Base Configuration, die applikations-weit unverschlüsselte HTTP-Kommunikation erlaubt. Auch domänen-spezifische Konfigurationen wurden primär dazu genutzt, HTTP-Kommunikation zu erlauben, nämlich in 88 % der Fälle. Abbildung 2 gibt diese Aufteilung grafisch wieder.

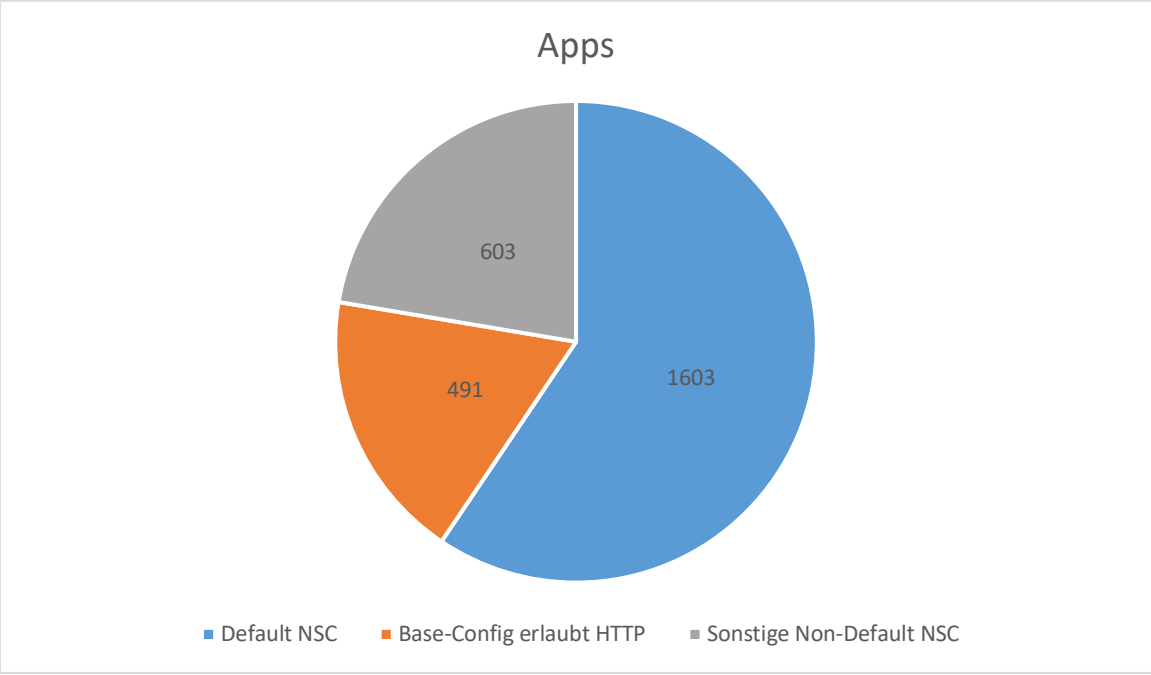

Abbildung 2 Clustering der häufigsten NSC-Eigenschaften

4.6. Interpretation der Erkenntnisse

Obwohl unsere Untersuchungen zeigen, dass sich die Sicherheit der Internet-Verbindungen von Android-Apps seit den letzten diesbezüglichen wissenschaftlichen Studien leicht verbessert hat, bleibt noch immer viel Verbesserungspotential. Positive Entwicklungen lassen sich besonders bei der Zahl der Apps feststellen, die HTTP-Verbindungen verwenden. Stellten Piccolboni et al. [5] hier noch 2020 fest, dass fast ein Drittel aller Apps betroffen waren, so konnten wir das Problem nur in rund 10 % der untersuchten Anwendungen finden. Die Erkenntnisse von Oltrogge et al. [7] in Bezug auf die Network Security Configuration konnten wir hingegen im Wesentlichen bestätigen. Deren Publikation dokumentiert den Fund von 57123 Apps, die global HTTP-Kommunikation erlauben, bei 96400 Apps mit Non-Default NSC. Das entspricht 59,2 %. Unsere Untersuchung an einem 2 Jahre jüngeren Datensatz ermittelte hier einen Anteil in ähnlicher Größenordnung von 44,9 %. Auch die Erkenntnis von Possemato et al. [6], wonach teilweise Werbe-Anbieter für unsichere NSCs verantwortlich sind, konnten wir in aktuellen Apps bestätigen.

5. Zusammenfassung

In diesem Bericht untersuchten wir die Prävalenz von HTTP in Android-Anwendungen. Es wurde ein Überblick über die Problematik von ungeschützten HTTP-Verbindungen, die Vorteile von TLS und die Schwierigkeiten des Deployments geboten. Zusätzlich wurde auch die Situation unter Android, insbesondere in Bezug auf HTTP-Verbindungen in Client-Anwendungen beleuchtet. Nach einer Zusammenfassung relevanter Literatur dokumentierten wir die Ergebnisse unserer eigenen Untersuchungen eines Datensatzes von aktuellen Android-Anwendungen. Wir konnten zeigen, dass zwar ein Positivtrend festzuhalten ist, aber dennoch viele Probleme, die in der Vergangenheit von der Wissenschaft aufgezeigt wurden, nach wie vor bestehen.

Referenzen

[1] T. Dierks, „The TLS Protocol Version 1.0," January 1999. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2246. [Zugriff am 25 January 2023].

[2] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel und P. Tabriz, „Measuring HTTPS Adoption on the Web," in *Proceedings of the 26th USENIX Security Symposium*, Vancouver, 2017.

[3] C. Kerschbaumer, J. Gaibler, A. Edelstein und T. Van der Merwe, „HTTPS-Only: Upgrading all connections to https in Web Browsers," in *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2021*, Virtual, 2021.

[4] J. Ren, M. Lindorfer und D. J. Dubois, „Bug Fixes, Improvements, ... and Privacy Leaks," in *Network and Distributed Systems Security (NDSS) Symposium 2018*, San Diego, 2018.

[5] L. Piccolboni, G. D. Guglielmo, L. P. Carloni und S. Sethumadhavan, „CRYLOGGER: Detecting Crypto Misuses Dynamically," in *IEEE Symposium on Security and Privacy*, San Francisco, 2021.

[6] A. Possemato und Y. Fratantonio, „Towards HTTPS Everywhere on Android: We Are Not There Yet," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[7] M. Oltrogge, N. Huaman, S. Amft, Y. Acar, M. Backes und S. Fahl, „Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications," in *Usenix Security Symposium 2021*, Vancouver, 2021.

[8] Y. Zhou, K. Patel, L. Wu, Z. Wang und X. Jiang, „Hybrid User-level Sandboxing of Third-party Android Apps," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015.