

# Chain of Fools

An Exploration of Certificate Chain Validation Mishaps

Olabode Anise,  
James Barclay,  
Nick Mooney



# Overview

- X.509 Certificates and Certification Path Validation
- Android SafetyNet Overview
- JWS Overview and SafetyNet JWS Usage
- pyOpenSSL and the X509Store
- More Bad Advice
- Misuse Resistance
- Quantifying the Use of SafetyNet
- Forging Android SafetyNet Attestations

Why Johnny can't  
*properly* validate  
certificate chains.



DUO LABS

# “verify certificate chain python”



verify certificate chain python

About 2,460,000 results (0.65 seconds)

[How to validate / verify an X509 Certificate chain of trust in ...](#)  
Jun 9, 2015 - While the response of Avi Das is valid for the trivial case of verifying a single trust anchor with a single leaf certificate, it places trust in the ...  
2 answers  
[Verify SSL/X509 certificate is signed by another certificate ...](#) Jul 19, 2016  
[Validate SSL certificates with Python](#) Nov 8, 2011  
[Retrieve and validate x509 certificate chain](#) Apr 1, 2013  
[Getting certificate chain with Python 3.3 SSL module](#) May 24, 2017  
More results from stackoverflow.com

[Verifying X509 Certificate Chain of Trust in Python - Avi Das](#)  
The openssl module on the terminal has a verify method that can be used to verify the certificate against a chain of trusted certificates, going all the way back to the root CA. The builtin ssl module has create\_default\_context(), which can build a certificate chain while creating a new SSLContext.

[Validate x509 Certificate in Python - 38911 Basic Bytes Free](#)  
Mar 15, 2016 - I need to validate a x509 certificate's chain of trust in python. TL;DR version is that you can use PyOpenSSL. The code below gives an example.

[Validate x509 certificate using pyOpenSSL - GitHub](#)  
Verify the certificate, returns None if it can validate the certificate ... It would be awesome if OpenSSL provided a way to verify untrusted chains as the openssl ... in this article



verify certificate chain python

About 2,460,000 results (0.65 seconds)

[How to validate / verify an X509 Certificate chain of trust in ...](#)

Jun 9, 2015 - While the response of Avi Das is valid for the trivial case of verifying a single trust anchor with a single leaf certificate, it places trust in the ...

[Verify SSL/X 509 certificate is signed by another certificate ...](#) Jul 19, 2016

[Validate SSL certificates with Python](#) Nov 8, 2011

[Retrieve and validate x509 certificate chain](#) Apr 1, 2013

[Getting certificate chain with Python 3.3 SSL module](#) May 24, 2017

More results from stackoverflow.com

[Verifying X509 Certificate Chain of Trust in Python - Avi Das](#)

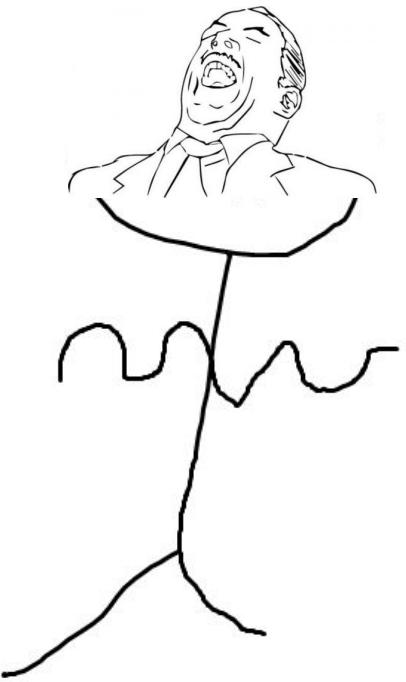
The openssl module on the terminal has a verify method that can be used to verify the certificate against a chain of trusted certificates, going all the way back to the root CA. The builtin ssl module has create\_default\_context(), which can build a certificate chain while creating a new SSLContext.

[Validate x509 Certificate in Python - 38911 Basic Bytes Free](#)

Mar 15, 2016 - I need to validate a x509 certificate's chain of trust in python. TL;DR version is that you can use PyOpenSSL. The code below gives an example.

[Validate x509 certificate using pyOpenSSL - GitHub](#)

Verify the certificate, returns None if it can validate the certificate ... It would be awesome if OpenSSL provided a way to verify untrusted chains as the openssl ... in this article



```
root_cert = load_certificate(FILETYPE_PEM,  
root_cert_pem)  
intermediate_cert =  
load_certificate(FILETYPE_PEM,  
intermediate_cert_pem)  
intermediate_server_cert =  
load_certificate(FILETYPE_PEM,  
intermediate_server_cert_pem)  
store = X509Store()  
store.add_cert(root_cert)  
store.add_cert(intermediate_cert)  
store_ctx = X509StoreContext(store,  
intermediate_server_cert)  
print(store_ctx.verify_certificate())
```

Unfortunately, there's a security bug in Johnny's code. Johnny might never know, though, because *everything keeps working.*

# X.509 Certificate Chains

A refresher

```
Certificate:  
  Data:  
    Version: 3 (0x2)  
    Serial Number:  
      04:00:00:00:00:01:15:4b:5a:c3:94  
    Signature Algorithm: sha1WithRSAEncryption  
    Issuer: C=BE, O=GlobalSign nv-sa, OU=Root  
CA, CN=GlobalSign Root CA  
    Validity  
      Not Before: Sep 1 12:00:00 1998 GMT  
      Not After : Jan 28 12:00:00 2028 GMT  
    Subject: C=BE, O=GlobalSign nv-sa, OU=Root  
CA, CN=GlobalSign Root CA  
    Subject Public Key Info:  
      Public Key Algorithm: rsaEncryption  
      Public-Key: (2048 bit)  
      Modulus: ...
```

# Components of an X.509 Certificate

- A public key
  - Fundamentally, a certificate is an *identity* associated with a key pair, where other parties can make claims about that identity
- Metadata such as subject name, SANs (valid domain names in the TLS context), organization info
- Issuer info (when not self-signed)



# The Chain of Trust

- Root CAs
  - Shipped with the operating system, sometimes the browser
  - Explicitly trusted
  - Used to sign other certificates, usually intermediate CA certificates
- Intermediate CAs
  - Not globally trusted, but part of a chain leading to a root CA
- Leaf certificates
  - The end of the chain
  - Identifying a particular key pair
    - Ex (SafetyNet): a key pair that is used to sign a SafetyNet attestation

# Validating a Certificate Chain

- The root CA must be self-signed and explicitly trusted
- The root CA must have signed the next intermediate in the chain, if one exists
- That intermediate must have signed the next...
- The last intermediate must have signed the client leaf

We also have to worry about:

- Making sure the leaf certificate legitimately describes the service
  - CN, SAN validation
- Making sure the intermediates are allowed to issue chains of  $n$  length
- Expiration and validity

# Validating a Certificate Chain

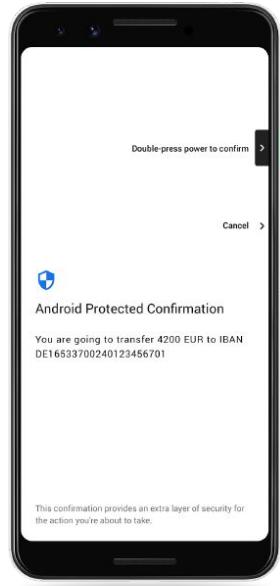
- (a) for all  $x$  in  $\{1, \dots, n-1\}$ , the subject of certificate  $x$  is the issuer of certificate  $x+1$ ;
- (b) certificate 1 is issued by the trust anchor;
- (c) certificate  $n$  is the certificate to be validated (i.e., the target certificate); and
- (d) for all  $x$  in  $\{1, \dots, n\}$ , the certificate was valid at the time in question.

When should  
**developers** have to  
worry about validating  
certificate chains?

Probably **never**.

# When might developers *actually* have to worry about validating certificate chains?

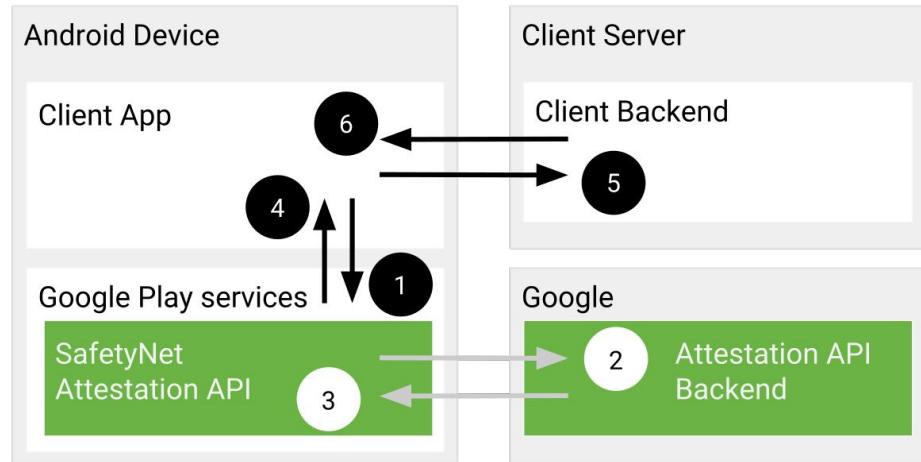
SafetyNet, Android Protected  
Confirmations, WebAuthn



# SafetyNet Overview

# What is SafetyNet?

A refresher



“SafetyNet provides a set of services and APIs that help protect your [Android] app against security threats, including device tampering, bad URLs, potentially harmful apps, and fake users.”

- Protect against security threats with SafetyNet

# SafetyNet APIs

1. [SafetyNet Attestation API](#)
2. [SafetyNet Safe Browsing API](#)
3. [SafetyNet reCAPTCHA API](#)
4. [SafetyNet Verify Apps API](#)

# SafetyNet APIs

1. [SafetyNet Attestation API](#)
2. [SafetyNet Safe Browsing API](#)
3. [SafetyNet reCAPTCHA API](#)
4. [SafetyNet Verify Apps API](#)

# SafetyNet Attestation API

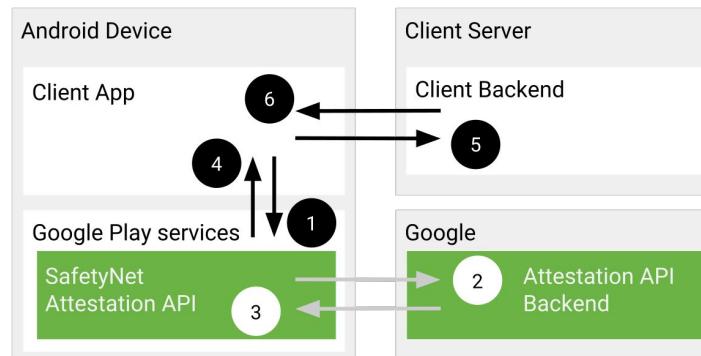
“The SafetyNet Attestation API is an **anti-abuse** API that allows app developers to assess the Android device their app is running on. The API should be used as a part of your abuse detection system to help determine whether your servers are interacting with **your genuine app** running on a **genuine Android device**.

The SafetyNet Attestation API provides a **cryptographically-signed attestation**, assessing the device's integrity...”

[SafetyNet Attestation API](#)

# How the SafetyNet Attestation API Works

1. Server requests attestation from mobile device
2. Mobile device does some health checks, produces a signed blob
3. Mobile device provides signed (JWS) blob along with intermediate certificates
4. Server checks payload and validates signature and certificate chain



# JSON Web Signatures (JWS)

A refresher

DUO LABS

[[Docs](#)] [[txt](#)|[pdf](#)] [[draft-ietf-jose...](#)] [[Tracker](#)] [[Diff1](#)] [[Diff2](#)] [[IPR](#)]

PROPOSED STANDARD

Internet Engineering Task Force (IETF)  
Request for Comments: 7515  
Category: Standards Track  
ISSN: 2070-1721

M. Jones  
Microsoft  
J. Bradley  
Ping Identity  
N. Sakimura  
NRI  
May 2015

## JSON Web Signature (JWS)

### Abstract

JSON Web Signature (JWS) represents content secured with digital signatures or Message Authentication Codes (MACs) using JSON-based data structures. Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification and an IANA registry defined by that specification. Related encryption capabilities are described in the separate JSON Web Encryption (JWE) specification.

### Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in [Section 2 of RFC 5741](#).

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at <http://www.rfc-editor.org/info/rfc7515>.

# JSON Web Signatures (JWS)

- Part of the [JavaScript Object Signing and Encryption \(JOSE\)](#) framework.

“JSON Web Signature (JWS) represents content secured with **digital signatures** or Message Authentication Codes (MACs) using JSON-based data structures.”

[JSON Web Signature \(JWS\) – RFC 7515](#)

# JSON Web Signatures (JWS)

- A JWS is a named tuple consisting of three logical values
  - JOSE Header
  - JWS Payload
  - JWS Signature
- Two serialization formats are supported
  - JWS JSON Serialization
  - **JWS Compact Serialization**

# JWS Compact Serialization

```
BASE64URL(UTF8(JWS Protected Header)) || '.' ||
BASE64URL(JWS Payload) || '.' ||
BASE64URL(JWS Signature)
```

**eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9**

.

**eyJwYXlsb2FkIjoiVGhpcyBpcyBteSBtZXNzYWdlLij9**

.

**bqHXShjW6Gcp\_Ck0NR7qLVvJy-D42mfo3NHsC7hiI0**

{

**"typ": "JWT",**

**"alg": "HS256"**

}

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9  
.  
**eyJwYXlsb2FkIjoiVGhpcyBpcyBteSBtZXNzYWdlLij9**  
.br/>**bqHXZhjW6Gcp\_Ck0NR7qLVvJy-D42mfo3NHsC7hiI0**

```
{  
  "payload": "This is my message."  
}
```

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

.

eyJwYXlsb2FkIjoiVGhpcyBpcyBteSBtZXNzYWdlLij9

.

bqHXShjW6Gcp\_Ck0NR7qLVvJy-D42mfo3NHsC7hiI0

signature

# SafetyNet JWS Usage

How Android SafetyNet uses  
JSON Web Signatures.

## PAYOUT: DATA

```
"nonce": "Yv4fvzUuGIzXJ4LBojK1dJvT5G8=" ,  
"timestampMs": 1557869580251 ,  
"apkPackageName": "com.mooney.safetynetexploration" ,  
"apkDigestSha256":  
"OdOsBSUpV9ipmUo0aRCSSWXjkIQYeAqzMUC9q9dXdEk=" ,  
"ctsProfileMatch": true ,  
"apkCertificateDigestSha256": [  
    "18P5sFPk8lRAK0fL++mIXI2sAmZ+xRu/cgqLzvdmx0A="  
],  
"basicIntegrity": true  
}
```

A SafetyNet attestation  
includes the **X.509**  
**certificate chain** in the  
JWS header.

# SafetyNet JWS Payload

```
{  
    "nonce": "Yv4fvzUuGIZXJ4LBOjK1dJvT568=",  
    "timestampMs": 1557869580251,  
    "apkPackageName": "com.mooney.safetynetexploration",  
    "apkDigestSha256": "OdOsbsUpV9ipmUoOaRCSsWXjKIQYeAqzMUC9q9dXdEk=",  
    "ctsProfileMatch    "apkCertificateDigestSha256": ["18P5sFPk81RAKOfL++mIXI2 sAmZ+xRu/cgqlzvdmx0A="],  
    "basicIntegrity}
```

# Checklist items

---

*Last updated in March 2019.*

- Your service uses [other signals](#), in addition to the SafetyNet Attestation API, to detect abuse.
- You've [applied for an API key](#), [requested quota](#) for your project, and used the correct associated API key(s) in your app.
- Your app uses the [SafetyNetClient](#), and not the deprecated [SafetyNetApi](#).
- Your app [verifies that the latest version of Google Play services is installed](#).



**Note:** A `minApkVersion` of **13000000** must be verified when using app-restricted API keys.

- Your app creates and uses large [nonces](#)—16 bytes or longer—that are either generated on your server or better yet, a part of your nonce is derived from the data you're sending to your server.
- Your app handles transient errors by retrying the request with an increasing amount of time between retries (exponential backoff).
- You're [verifying the results](#) of the API on a server that you control.
- You've implemented a JWS signature validator in your own server, such as the one in the [code samples](#) that we offer.
- At a minimum, your server verifies the timestamp, nonce, APK name, and APK signing certificate hash(es) included in the attestation response.

#### 4.1.6. "x5c" (X.509 Certificate Chain) Header Parameter

The "x5c" (X.509 certificate chain) Header Parameter contains the X.509 public key certificate or certificate chain [[RFC5280](#)] corresponding to the key used to digitally sign the JWS. The certificate or certificate chain is represented as a JSON array of

Jones, et al.

Standards Track

[Page 11]

---

[RFC 7515](#)

JSON Web Signature (JWS)

May 2015

certificate value strings. Each string in the array is a base64-encoded ([Section 4 of \[RFC4648\]](#). -- not base64url-encoded) DER [[ITU.X690.2008](#)] PKIX certificate value. The certificate containing the public key corresponding to the key used to digitally sign the JWS MUST be the first certificate. This MAY be followed by additional certificates, with each subsequent certificate being the one used to certify the previous one. The recipient MUST validate the certificate chain according to [RFC 5280](#) [[RFC5280](#)] and consider the certificate or certificate chain to be invalid if any validation failure occurs. Use of this Header Parameter is OPTIONAL.

See [Appendix B](#) for an example "x5c" value.

### ***Attestation Certificate***

A X.509 Certificate for the **attestation key pair** used by an authenticator to attest to its manufacture and capabilities. At registration time, the authenticator uses the **attestation private key** to sign the Relying Party-specific credential public key (and additional data) that it generates and returns via the authenticatorMakeCredential operation. Relying Parties use the **attestation public key** conveyed in the attestation certificate to verify the attestation signature. Note that in the case of self attestation, the authenticator has no distinct attestation key pair nor attestation certificate, see self attestation for details.

# pyOpenSSL and the X509Store

An observation, and the genesis  
of our research.



# pyOpenSSL

- Part of the Python Cryptographic Authority, along with other great projects like [cryptography](#).
- Thin wrapper around a subset of the OpenSSL library.
- **Note:** The [Python Cryptographic Authority](#) recommends not using pyOpenSSL for anything other than making TLS connections.



# pyOpenSSL and the X509Store Class

- X509Store
  - “An X.509 store is used to describe a context in which to verify a certificate. A description of a context may include a set of certificates to trust, a set of certificate revocation lists, verification flags and more.”
- X509StoreContext
  - “An X.509 store context is used to carry out the actual verification process of a certificate in a described context.”

# Verifying Certificate Chains With Python

- A cursory glance for how to verify certificate chains with Python will likely result in something like this:

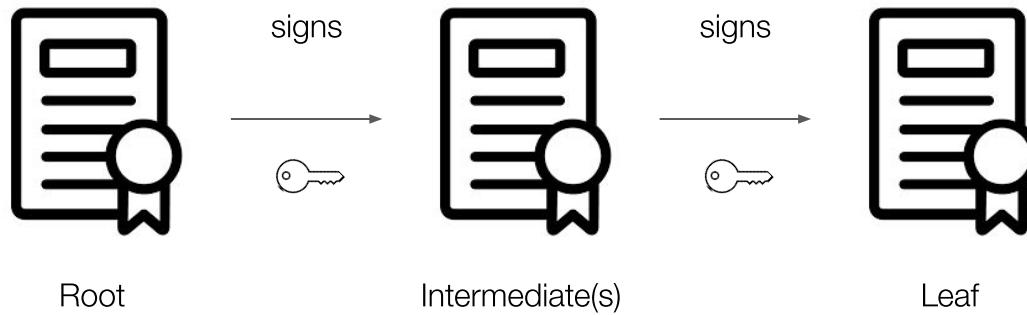
```
root_cert = load_certificate(FILETYPE_PEM, root_cert_pem)
intermediate_cert = load_certificate(FILETYPE_PEM, intermediate_cert_pem)
leaf_cert = load_certificate(FILETYPE_PEM, leaf_cert_pem)

store = X509Store()
store.add_cert(root_cert)
store.add_cert(intermediate_cert)
store_ctx = X509StoreContext(store, leaf_cert)

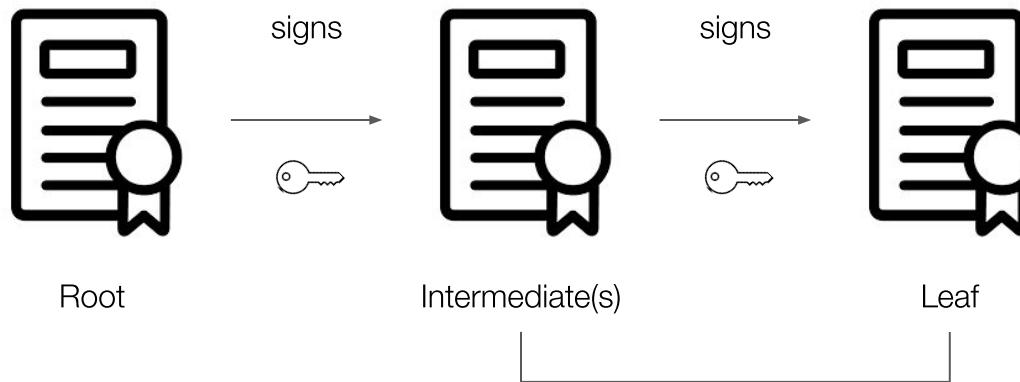
print(store_ctx.verify_certificate())
```

This pattern treats any  
*intermediate certs* as  
**trusted roots.**

# Assumptions About Certificate Chain Validation

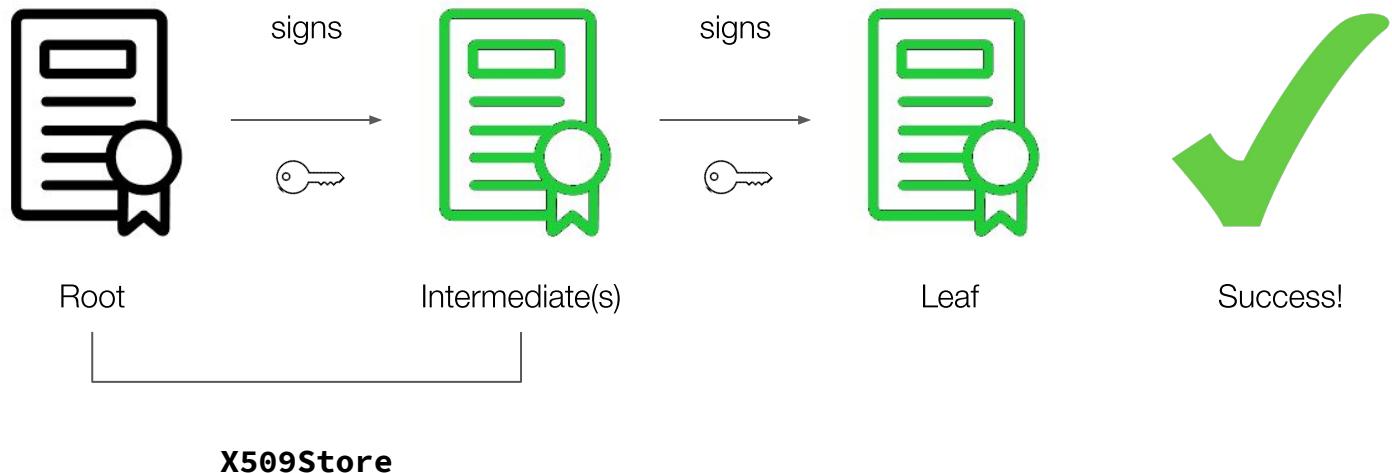


# But What If?

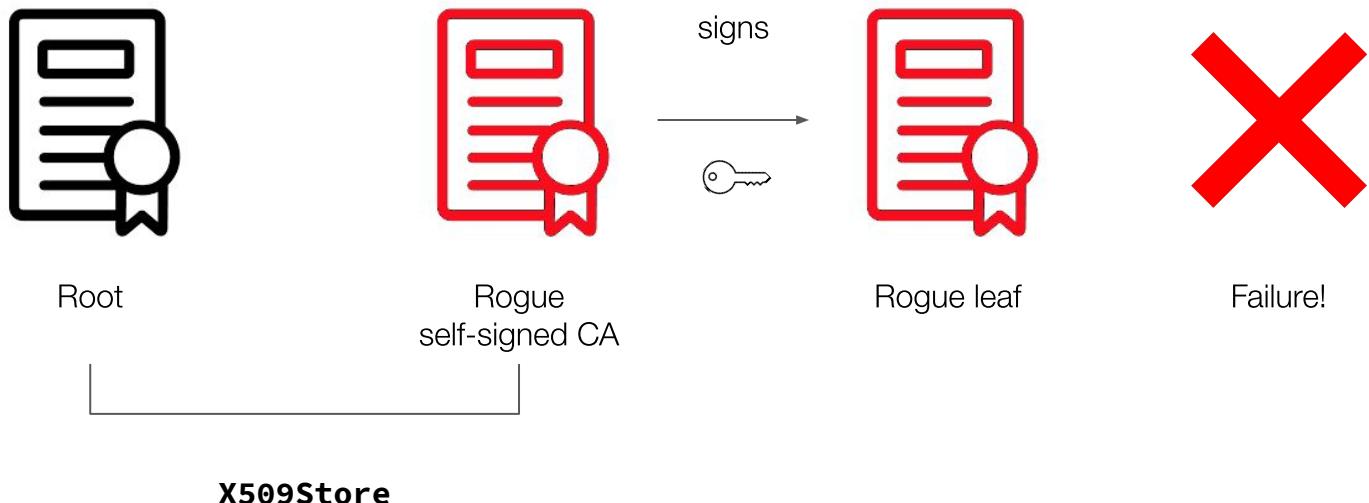


If controlled by adversary

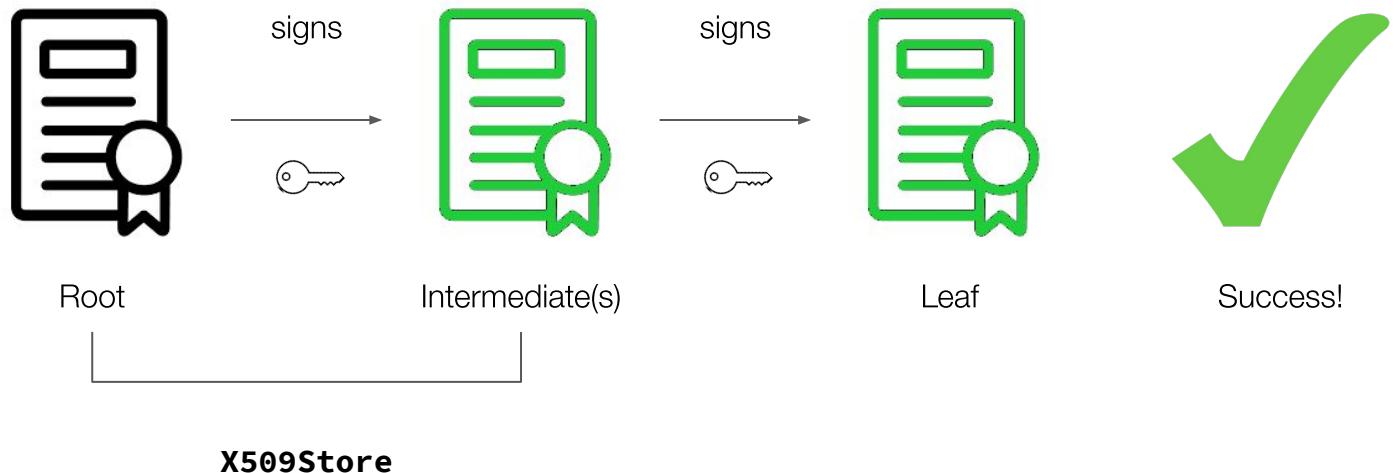
# How Developers Think It's Working



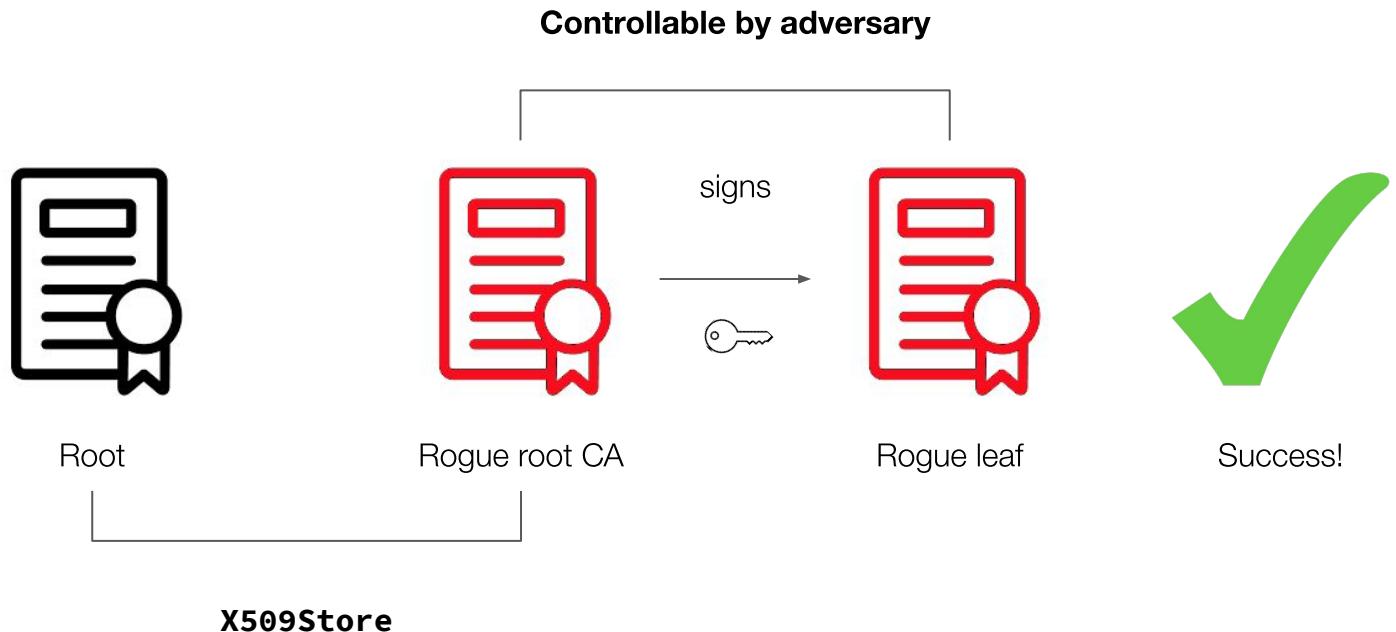
# How Developers Think It's Working



# How It Will Actually Work



# How It Will Actually Work



There is *still* no  
good way to do  
this.

*We also made this same mistake.*  
Fortunately, we caught it in  
development. Not everyone is so lucky.

# pyOpenSSL PR #473

- We remembered a colleague of ours, [Adam Goodman](#), talking about this apparent limitation in the pyOpenSSL API, so we checked with him.
- Adam [opened a PR](#) on the pyOpenSSL GitHub repository to add untrusted certificate chain support to X509StoreContext.
  - This was in *June of 2016*.
  - This is not for lack of caring by the project maintainers, it's just a sensitive change in a cryptographic library, and not many people are qualified to review it.
  - **This is a good reason.**
- We ended up just using his fork in our code.

# The right way to do things

...with a non-obvious API.

```
def test_issuer():
    """
    Issue a certificate, fetch its chain, and verify the chain and
    certificate against test/test-root.pem. Note: This test only handles chains
    of length exactly 1.
    """
    certr, authzs = auth_and_issue([random_domain()])
    cert = urllib2.urlopen(certr.uri).read()
    # In the future the chain URI will use HTTPS so include the root certificate
    # for the WFE's PKI. Note: We use the requests library here so we honor the
    # REQUESTS_CA_BUNDLE passed by test.sh.
    chain = requests.get(certr.cert_chain_uri).content
    parsed_chain = OpenSSL.crypto.load_certificate(OpenSSL.crypto.FILETYPE_ASN1, chain)
    parsed_cert = OpenSSL.crypto.load_certificate(OpenSSL.crypto.FILETYPE_ASN1, cert)
    parsed_root = OpenSSL.crypto.load_certificate(OpenSSL.crypto.FILETYPE_PEM,
                                                open("test/test-root.pem").read())

    store = OpenSSL.crypto.X509Store()
    store.add_cert(parsed_root)

    # Check the chain certificate before adding it to the store.
    store_ctx = OpenSSL.crypto.X509StoreContext(store, parsed_chain)
    store_ctx.verify_certificate()
    store.add_cert(parsed_chain)

    # Now check the end-entity certificate.
    store_ctx = OpenSSL.crypto.X509StoreContext(store, parsed_cert)
    store_ctx.verify_certificate()
```

# Doing the Right Thing With a Non-obvious API

- Adam also pointed out to us that he's seen an example of how to use pyOpenSSL to correctly validate a certificate chain.
  - This was in a test in the [letsencrypt/boulder](#) project.
  - The way they accomplished this is not obvious, and would require an understanding of the limitations of this API.
  - Also, **this is the only place we've seen the API being used correctly**, (so far).

# Doing the Right Thing With a Non-obvious API

```
store = OpenSSL.crypto.X509Store()
store.add_cert(parsed_root)

# Check the chain certificate before adding it to the store.
store_ctx = OpenSSL.crypto.X509StoreContext(store, parsed_chain)
store_ctx.verify_certificate()
store.add_cert(parsed_chain)

# Now check the end-entity certificate.
store_ctx = OpenSSL.crypto.X509StoreContext(store, parsed_cert)
store_ctx.verify_certificate()
```

Avoid making  
developers do this  
themselves.

# What do we need to take into consideration?

- Correct path building
- Name validation (CN matching, etc.)
  - Null byte vulnerability
- Basic Constraints
  - IsCA, path length
- Usage Flags
- Revocation
- Permitted hash, signature algorithms

# Some purpose-built software

- Web browsers are great at this!
  - Mostly calling into OS-provided libraries
- Your OS has a battle-tested way to do this
  - Mostly used for establishing TLS connections
- For Python: [certvalidator](#)
- For more specific tasks (JWS validation), use libraries like Google's [jws](#)

# More bad advice on the internet

DUO LABS

Google search results for "verify certificate chain python":

- How to validate / verify an X509 Certificate chain of trust in ...**  
https://stackoverflow.com/questions/11900000/how-to-validate-x509-certificate-chain-of-trust... ▾  
2 answers  
Oct 19, 2011 While the response of Avi Das is valid for the trivial case of verifying a single trust anchor with a single leaf certificate, it places trust in the intermediate certificate.
- Python requests SSL error - certificate verify failed**  
Oct 6, 2017
- Python Requests - How to use system certificates (debian/ubuntu ...**  
Mar 23, 2017
- Verify SSL/X.509 certificate is signed by another certificate ...**  
Jul 19, 2016
- Validate SSL certificates with Python**  
Nov 8, 2011
- More results from stackoverflow.com**

**Wrong**

**Wrong**

**Wrong**

**Validate x509 Certificate in Python - 38911 Basic Bytes Free**  
www.yethennerg.com/article/38911-validate-x509-certificate-in-python/ ▾  
May 15, 2016 I need to validate a x509 certificate's chain of trust in python. TL;DR version is that you can use PyOpenSSL. The code below gives an example.

**Validate x509 certificate using pyOpenSSL · GitHub**  
https://gist.github.com/luilunes/04af5928fb163e49b1b8ef03b30d421 ▾  
Jan 4, 2017 Validate x509 certificate using pyOpenSSL. GitHub Gist: ... Raw. cert-check.py ... can you get the full certificate chain?about the chain info.

**Add x509 Certificate Validation - issue #2381 · pyca/cryptography**  
https://github.com/pyca/cryptography/issues/2381 ▾

**Not done**

## Verify a certificate with chain with golang crypto library

### verify\_certificate.go

```
1 package main
2
3 import (
4     "crypto/x509"
5     "encoding/pem"
6     "io/ioutil"
7     "log"
8     "os"
9 )
10
11 func main() {
12     log.Printf("Usage: verify_certificate SERVER_NAME CERT.pem CHAIN.pem")
13
14     serverName := os.Args[1]
15
16     certPEM, err := ioutil.ReadFile(os.Args[2])
17     if err != nil {
18         log.Fatal(err)
19     }
20
21     rootPEM, err := ioutil.ReadFile(os.Args[3])
22     if err != nil {
```



dimalinux commented on May 27

...

This example is not solving what people are searching for when they find it. The so-called "chain" in this example is trusted and all of its certs are put into the root store. While it is common to place some intermediate certs into a root store for faster verification, certs in the root store do not form a chain. Any certificate in the root store is trusted absolutely without having traverse further up a chain. Hence the word "root".

Can you modify the example to do what the title says? Start with a root certificate and verify a certificate that has one or more intermediate certificates attached to it as a chain.

2

I looked into pyopenssl library and found this for certificate chain validation. The following example is from [their tests](#) and seem to do what you want, which is validating chain of trust to a trusted root certificate. Here are the relevant docs for [X509Store](#) and [X509StoreContext](#)

```
from OpenSSL.crypto import load_certificate, load_privatekey
from OpenSSL.crypto import X509Store, X509StoreContext
from six import u, b, binary_type, PY3
root_cert_pem = b("""-----BEGIN CERTIFICATE-----
```

share improve this answer

answered Jun 8 '15 at 22:07



Avi Das

1,437 ● 11 ● 18

So, I can't get this example to work -- am I missing something? the verify\_cert call is always returning `None` (for the certificate provided in your example and my own certificate that I tested with). I had to add the `FILETYPE_PEM` import at the top along with your other imports from OpenSSL.crypto. I also tried remove the extraneous newlines (after the "END CERTIFICATE" lines) in your cert strings but it is still returning none. Thank you very much for your thoughts and time on this! – [speznot](#) Jun 8 '15 at 22:28

Ah! I had missed the comment staring me directly in the face in that documentation saying that `None` is a valid response -- my apologies. I have been trying to test with my own certificates and am getting "unable to get [local] issuer certificate" errors. On the command line I am using something like this to verify successfully:  
`openssl verify -untrusted intermediate_cert.pem -CAfile rootcert.pem tovalidate.pem` (without the `-untrusted` switch it fails with similar errors I am seeing) -- is it correct that in your example `intermediate_server_cert` is the cert that I am validating? – [speznot](#) Jun 8 '15 at 23:02

Correct, `intermediate_server_cert` is the cert that is getting validated in this example. I think that error generally means that a particular certificate is missing somewhere in the chain. That is strange though, I have tried with a different example and it did manage to resolve for me. – [Avi Das](#) Jun 9 '15 at 0:50

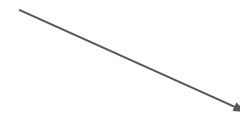
Thank you very much for your help @Avi. I am still working on this, trying to get past that error. I have pasted the certificate that I am attempting to validate here [pastebin.com/z9TbDPV](http://pastebin.com/z9TbDPV) if you'd like to attempt it I'd be very grateful. – [speznot](#) Jun 9 '15 at 1:24

- 1 You are right, this is a completely wrong answer. **Putting trust in an intermediate certificate is a very, very bad idea.** I have added an answer that is correct, or at least less incorrect. – [ralphje](#) Mar 14 '18 at 16:15

[show 3 more comments](#)

```
63         if cert.IsCA {
64             roots.AddCert(cert)
65         } else {
66             intermediates.AddCert(cert)
67         }
68     }
69     opts := x509.VerifyOptions{
70         Intermediates: intermediates,
71         Roots:         roots,
72     }
```

## Part of the Intel “Security Libraries for Data Center”



```
63         if cert.IsCA {
64             roots.AddCert(cert)
65         } else {
66             intermediates.AddCert(cert)
67         }
68     }
69     opts := x509.VerifyOptions{
70         Intermediates: intermediates,
71         Roots:         roots,
72     }
```

# Misuse Resistance

Primitives and APIs

- Misuse resistant primitives
- Misuse resistant APIs
- Well-supported libraries and good documentation

# Primitives

- Deterministic signature schemes
  - EdDSA vs ECDSA (avoiding the PS3 problem)
- Certain representations of data
  - For example, compressed curve points
- AES-GCM-SIV
  - “Occasional nonce duplication tolerant” per [Adam Langley](#)

# APIs / Libraries

- libsodium
  - Important to note that the primitives can still fail spectacularly with nonce reuse
- Tony Arcieri's Miscreant
  - Implementation of primitives where the *primitives themselves* are designed with misuse resistance in mind
- Tink
  - “Using crypto in your application shouldn't have to feel like juggling chainsaws in the dark”
- Noise framework
  - Provable building blocks for crypto protocols
- General high level OS libraries
  - If your goal is “make a TLS connection,” it is likely that most questions related to certificate chains will be automatically answered for you

# Quantifying the Use of SafetyNet

Analyzing Thousands of Apps



# Amassing a list of Android Apps

- There are a lot of freemium sites that perform app store analytics, but unfortunately those sources don't allow for easy reproducibility or scale
- We opted to use [AndroidRank](#), which provides similar data for free
- Their website provided us the most popular apps for **32** general application categories and **17** gaming categories
- Our entire list was composed of **24,296** applications

# Building a Corpus of Applications

- Downloading ~24k apps from the Google Play Store directly wasn't feasible
- We opted to use two Play Store mirroring sites: [apkmonk](#) and [Apkpure](#)
- Using these sources, we were able to download APKs for 98% of the apps we found on AndroidRank



# Analyzing Apps - Overview

- Android Package (APK) is the package format for Android apps
- APKs ≈ Zip files
- These files contain the resources, assets and compiled versions of the source code and libraries



Diagram of APK file structure

# Analyzing APKs - Properties files

- Properties file contain configuration information for apps
- In the case of the SafetyNet API, it details the version that the app is using



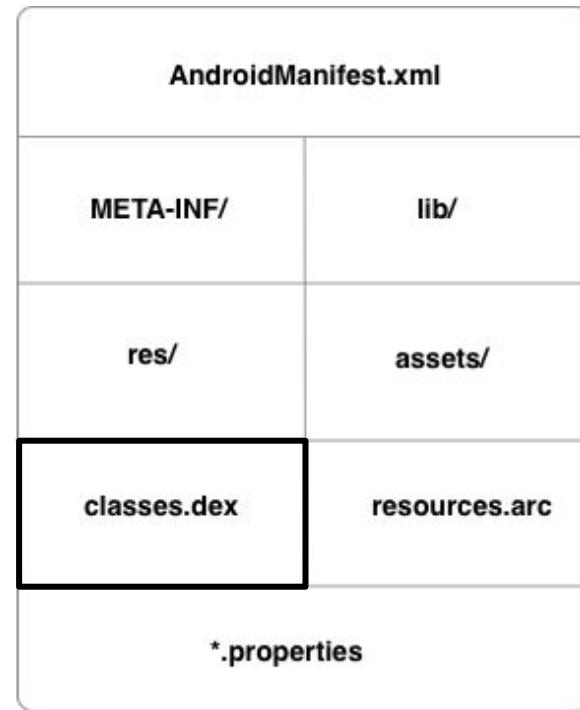
# Analyzing APKs - Manifest file

- Required in every Android app
- Provides a lot of the crucial information concerning apps:
  - Activities, content providers, permissions the app requests, hardware/software the app requires, etc.
- It also may include SafetyNet API keys
  - com.google.android.safetynet.ATTEST\_API\_KEY
  - com.google.android.safetynet.API\_KEY



# Analyzing APKs - `classes.dex`

- `classes.dex` is a Dalvik Executable file that contains the compiled application code
  - This includes both original source code and other libraries
- By analyzing the string IDs, method IDs, and class definitions, we can find use of the SafetyNet APIs



4% of apps are  
using SafetyNet.

# Results from Initial Approach

Application Category	Percent using a SafetyNet API
Finance	18.52
Comics	12.63
Dating	11.00
Shopping	9.85
Gaming	5.23

# Limitations with Initial Approach

- Any type of regex is going to be brittle and will potentially bring about a lot of false negatives
- Searching for strings just doesn't work when code is obfuscated, as some source code is
- All results are biased towards files that store API authentication and configuration information in manifest files or property files

# From String to Static Analysis

- Because of the limitations of string analysis, we decided to use a third-party library analysis tool: **LibScout**
- LibScout works by extracting profiles from an original library and applying a matching algorithm to check how much the two match
- The resulting output is a similarity score that is between 0 and 1

# Results from LibScout

- Using LibScout, we found that **7.1%** of apps were using at least one of the SafetyNet APIs
- Applications in the Gaming category used SafetyNet the most at **11.3%**
- The majority (**87%**) of Android Apps that we analyzed were using an older version of the SafetyNet API

# General Limitations

- The corpus of Android apps that we collected is different from [the study run in 2017](#)
- Our list of apps is not a random sample of Android applications, so it's difficult to generalize our results

# Some tooling

- SafetyNet Android example
- Flask SafetyNet server
- MITM tools

[github.com/duo-labs/chain-of-fools](https://github.com/duo-labs/chain-of-fools)

# Forging SafetyNet Attestations

# Forging SafetyNet Attestations

- We can do this by
  - Modifying in-flight JWS to inject our own rogue CA certificate and leaf into the JOSE Header
    - **x5c** parameter
  - Transform JWS payload to set **basicIntegrity** and **ctsProfileMatch** to **true**
    - Or false, if we want to make SafetyNet checks fail when they shouldn't!
  - Re-sign the payload with our rogue certificate private key, then swap out the JWS signature

# MITM Tools

- `rogue_ca.py`
  - Helper utilities for generating self-signed certificates and signing payloads, generating cert chains
- `jwsmodify.py`
  - Modify in-flight JWS requests
  - Apply a transformation function to JWS payload contents
  - Automatically re-sign and modify X.509 chain
- `jwsmodify_mitmproxy_addon.py`
  - mitmproxy Addon for jwsmodify

# Demo

# Conclusion

# Conclusion

- Ideally, developers shouldn't have to worry about cryptographic implementation details like validating certificate chains. Frameworks and vendor tooling should *abstract as much of this away as possible*.
  - If you do, choose *misuse-resistant primitives and/or APIs*.
- It's relatively easy to take advantage of incorrect certificate chain validation logic.
  - Forging SafetyNet Attestations is just one example. Other examples include [Android Protected Confirmation](#) and [WebAuthn attestation](#).
- Android SafetyNet usage is *steadily increasing*, with Gaming and Finance being the biggest adopters.
- Certificate chain validation is hard to get right!
  - Try not to assign blame if someone gets this wrong. Let's work together to make things better!

# Thank you!

@JustSayO

@futureimperfect

@wellhydrated

