

Certificate and Public Key Pinning

From OWASP

Certificate and Public Key Pinning is a technical guide to implementing certificate and public key pinning as discussed at the *Virginia chapter's* (<https://www.owasp.org/index.php/Virginia>) presentation *Securing Wireless Channels in the Mobile Space*. This guide is focused on providing clear, simple, actionable guidance for securing the channel in a hostile environment where actors could be malicious and the conference of trust a liability. Additional presentation material included supplement with code excerpts, Android sample program, iOS sample program, .Net sample program, and OpenSSL sample program.

A cheat sheet is available at [Pinning Cheat Sheet](#).

- 1 Introduction
- 2 What's the problem?
 - 2.1 Patient 0
 - 2.2 The Cures
- 3 What Is Pinning?
 - 3.1 When Do You Pin?
 - 3.2 When Do You Whitelist?
 - 3.3 How Do You Pin?
- 4 What Should Be Pinned?
 - 4.1 Encodings/Formats
 - 4.2 Certificate
 - 4.3 Public Key
 - 4.4 Hashing
- 5 What About X509?
 - 5.1 Mandatory Checks
 - 5.2 Optional Checks
 - 5.3 Public Key Checks
- 6 Examples of Pinning
 - 6.1 HTTP pinning
 - 6.2 Android
 - 6.3 iOS
 - 6.4 .Net
 - 6.5 OpenSSL
- 7 Pinning Alternatives
 - 7.1 SRP
 - 7.2 PSK
- 8 Miscellaneous
 - 8.1 Ephemeral Keys
 - 8.2 Pinning Gaps
 - 8.3 No Relationship ^@\$!
 - 8.4 More Information?
 - 8.5 Format Conversions
- 9 References
- 10 Authors and Primary Editors

Introduction

Secure channels are a cornerstone to users and employees working remotely and on the go. Users and developers expect end-to-end security when sending and receiving data - especially sensitive data on channels protected by VPN, SSL, or TLS. While organizations which control DNS and CA have likely reduced risk to trivial levels under most threat models, users and developers subjugated to other's DNS and a public CA hierarchy are exposed to non-trivial amounts of risk. In fact, history has shown those relying on outside services have suffered chronic breaches in their secure channels.

The pandemic abuse of trust has resulted in users, developers and applications making security related decisions on untrusted input. The situation is somewhat of a paradox: entities such as DNS and CAs are trusted and supposed to supply trusted input; yet their input cannot be trusted. Relying on untrusted input for security related decisions is not only bad karma, it violates a number of secure coding principals (see, for example, OWASP's Injection Theory and Data Validation).

Pinning effectively removes the "conference of trust". An application which pins a certificate or public key no longer needs to depend on others - such as DNS or CAs - when making security decisions relating to a peer's identity. For those familiar with SSH, you should realize that public key pinning is nearly identical to SSH's `StrictHostKeyChecking` option. SSH had it right the entire time, and the rest of the world is beginning to realize the virtues of directly identifying a host or service by its public key.

Others who actively engage in pinning include Google and its browser Chrome. Chrome was successful in detecting the DigiNotar compromise which uncovered suspected interception by the Iranian government on its citizens. The initial report of the compromise can be found at *Is This MITM Attack to Gmail's SSL?*

(<https://productforums.google.com/d/topic/gmail/3J3r2JqFNTw/discussion>); and Google Security's immediate response at *An update on attempted man-in-the-middle attacks* (<https://googleonlinesecurity.blogspot.com/2011/08/update-on-attempted-man-in-middle.html>).

What's the problem?

Users, developers, and applications expect end-to-end security on their secure channels, but some secure channels are not meeting the expectation. Specifically, channels built using well known protocols such as VPN, SSL, and TLS can be vulnerable to a number of attacks.

Examples of past failures are listed on the discussion tab for this article. This cheat sheet does not attempt to catalogue the failures in the industry, investigate the design flaws in the scaffolding, justify the lack of accountability or liability with the providers, explain the race to the bottom in services, or demystify the collusion between, for example, Browsers and CAs. For additional reading, please visit *PKI is Broken* (<http://www.cs.auckland.ac.nz/~pgut001/pubs/pkitutorial.pdf>) and *The Internet is Broken* (<http://blog.cryptographyengineering.com/2012/02/how-to-fix-internet.html>).

Patient 0

The original problem was the *Key Distribution Problem*. Insecure communications can be transformed into a secure communication problem with encryption. Encrypted communications can be transformed into an identity problem with signatures. The identity problem terminates at the key distribution problem. They are the same problem.

The Cures

There are three cures for the key distribution problem. First is to have first hand knowledge of your partner or peer (i.e., a peer, server or service). This could be solved with SneakerNet. Unfortunately, SneakerNet does not scale and cannot be used to solve the key distribution problem.

The second is to rely on others, and it has two variants: (1) web of trust, and (2) hierarchy of trust. Web of Trust and Hierarchy of Trust solve the key distribution problem in a sterile environment. However, Web of Trust and Hierarchy of Trust each requires us to rely on others - or **confer trust**. In practice, trusting others is showing to be problematic.

What Is Pinning?

Pinning is the process of associating a host with their *expected* X509 certificate or public key. Once a certificate or public key is known or seen for a host, the certificate or public key is associated or 'pinned' to the host. If more than one certificate or public key is acceptable, then the program holds a *pinset* (taking from Jon Larimer and Kenny Root Google I/O talk (<https://developers.google.com/events/io/sessions/gooio2012/107/>)). In this case, the advertised identity must match one of the elements in the pinset.

A host or service's certificate or public key can be added to an application at development time, or it can be added upon first encountering the certificate or public key. The former - adding at development time - is preferred since *preloading* the certificate or public key *out of band* usually means the attacker cannot taint the pin. If the certificate or public key is added upon first encounter, you will be using *key continuity*. Key continuity can fail if the attacker has a privileged position during the first encounter.

Pinning leverages knowledge of the pre-existing relationship between the user and an organization or service to help make better security related decisions. Because you already have information on the server or service, you don't need to rely on generalized mechanisms meant to solve the *key distribution* problem. That is, you don't need to turn to DNS for name/address mappings or CAs for bindings and status. One exception is revocation and it is discussed below in Pinning Gaps.

It is also worth mention that Pinning is not Stapling. Stapling sends both the certificate and OCSP responder information in the same request to avoid the additional fetches the client should perform during path validations.

When Do You Pin?

You should pin anytime you want to be relatively certain of the remote host's identity or when operating in a hostile environment. Since one or both are almost always true, you should probably pin all the time.

A perfect case in point: during the two weeks or so of preparation for the presentation and cheat sheet, we've observed three relevant and related failures. First was Nokia/Opera willfully breaking the secure channel (<http://gaurangkp.wordpress.com/2013/01/09/nokia-https-mitm/>); second was DigiCert issuing a code signing certificate for malware (<http://blog.malwarebytes.org/intelligence/2013/02/digital-certificates-and-malware-a-dangerous-mix/>); and third was Bit9's loss of its root signing key (<http://krebsonsecurity.com/2013/02/security-firm-bit9-hacked-used-to-spread-malware/>). The environment is not only hostile, it's toxic.

When Do You Whitelist?

If you are working for an organization which practices "egress filtering" as part of a Data Loss Prevention (DLP) strategy, you will likely encounter *Interception Proxies*. I like to refer to these things as "**good**" **bad guys** (as opposed to "**bad**" **bad guys**) since both break end-to-end security and we can't tell them apart. In this case, **do not** offer to whitelist the interception proxy since it defeats your security goals. Add the interception proxy's public key to your pinset after being **instructed** to do so by the folks in Risk Acceptance.

Note: if you whitelist a certificate or public key for a different host (for example, to accommodate an interception proxy), you are no longer pinning the expected certificates and keys for the host. Security and integrity on the channel could suffer, and it surely breaks end-to-end security expectations of users and organizations.

For more reading on interception proxies, the additional risk they bestow, and how they fail, see Dr. Matthew Green's *How do Interception Proxies fail?* (<http://blog.cryptographyengineering.com/2012/03/how-do-interception-proxies-fail.html>) and Jeff Jarmoc's BlackHat talk *SSL/TLS Interception Proxies and Transitive Trust* (<https://www.blackhat.com/html/bh-eu-12/bh-eu-12-archives.html#jarmoc>).

How Do You Pin?

The idea is to re-use the existing protocols and infrastructure, but use them in a hardened manner. For re-use, a program would keep doing the things it used to do when establishing a secure connection.

To harden the channel, the program would take advantage of the `onConnect` callback offered by a library, framework or platform. In the callback, the program would verify the remote host's identity by validating its certificate or public key. While pinning does not have to occur in an `onConnect` callback, its often most convenient because the underlying connection information is readily available.

What Should Be Pinned?

The first thing to decide is what should be pinned. For this choice, you have two options: you can (1) pin the certificate; or (2) pin the public key. If you choose public keys, you have two additional choices: (a) pin the `subjectPublicKeyInfo`; or (b) pin one of the concrete types such as `RSAPublicKey` or `DSAPublicKey`.

The three choices are explained below in more detail. I would encourage you to pin the `subjectPublicKeyInfo` because it has the public parameters (such as `{e,n}` for an RSA public key) **and** contextual information such as an algorithm and OID. The context will help you keep your bearings at times, and Figure 1 below shows the additional information available.

Encodings/Formats

For the purposes of this article, the objects are in X509-compatible presentation format (PKCS#1 defers to X509, both of which use ASN.1). If you have a PEM encoded object (for example, `-----BEGIN CERTIFICATE-----`, `-----END CERTIFICATE-----`), then convert the object to DER encoding. Conversion using OpenSSL is offered below in Format Conversions.

A certificate is an object which binds an entity (such as a person or organization) to a public key via a signature. The certificate is DER encoded, and has associated data or attributes such as *Subject* (who is identified or bound), *Issuer* (who signed it), *Validity* (*NotBefore* and *NotAfter*), and a *Public Key*.

A certificate has a *subjectPublicKeyInfo*. The `subjectPublicKeyInfo` is a key with additional information. The ASN.1 type includes an *Algorithm ID*, a *Version*, and an extensible format to hold a concrete public key. Figures 1 and 2 below show different views of the same RSA key, which is the `subjectPublicKeyInfo`. The key is for the site `random.org` (<https://www.random.org>), and it is used in the sample programs and listings below.

```

riemann::pubkey-pin-openssl$ dumpasn1.exe random-org.der
0 290: SEQUENCE {
4 13: SEQUENCE {
6 9: OBJECT IDENTIFIER rsaEncryption (1 2 840 113549 1 1)
17 0: NULL
:
:
19 271: BIT STRING, encapsulates {
24 266: SEQUENCE {
28 257: INTEGER
:
: 00 B3 5E A8 AD AF 4C B6 DB 86 06 8A 83 6F 3C 85
: 5A 54 5B 1F 0C C8 AF B1 9E 38 21 3B AC 4D 55 C3
: F2 F1 9D F6 DE E8 2E AD 67 F7 0A 99 01 31 B6 BC
: AC 1A 91 16 AC C8 83 86 2F 00 59 31 99 DF 19 CE
: 02 7C 8E AA AE 8E 31 21 F7 F3 29 21 94 64 E6 57
: 2C BF 66 E8 E2 29 EA C2 99 2D 07 95 C4 F2 3D F0
: FE 72 B6 CE EF 45 7E BA 0B 90 29 61 9E 03 95 B8
: 60 98 51 84 9D D6 21 45 89 A2 CE BA 4F 7A 7D CC
: [ Another 129 bytes skipped ]
289 3: INTEGER 65537
:
: }
:
: }
0 warnings, 0 errors.

```

Figure 1: subjectPublicKeyInfo dumped with dumpasn1

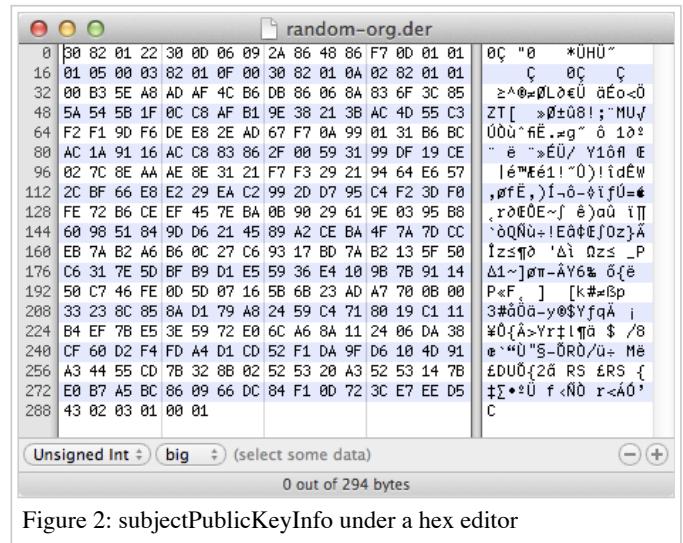


Figure 2: subjectPublicKeyInfo under a hex editor

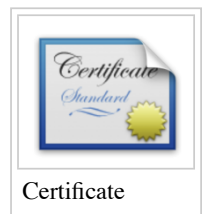
The concrete public key is an encoded public key. The key format will usually be specified elsewhere - for example, PKCS#1 in the case of RSA Public Keys. In the case of an RSA public key, the type is *RSAPublicKey* and the parameters $\{e, n\}$ will be ASN.1 encoded. Figures 1 and 2 above clearly show the modulus (n at line 28) and exponent (e at line 289). For DSA, the concrete type is *DSAPublicKey* and the ASN.1 encoded parameters would be $\{p, q, g, y\}$.

Final takeaways: (1) a certificate binds an entity to a public key; (2) a certificate has a *subjectPublicKeyInfo*; and (3) a *subjectPublicKeyInfo* has a concrete public key. For those who want to learn more, a more in-depth discussion from a programmer's perspective can be found at the Code Project's article *Cryptographic Interoperability: Keys* (<http://www.codeproject.com/Articles/25487/Cryptographic-Interoperability-Keys>).

Certificate

The certificate is easiest to pin. You can fetch the certificate out of band for the website, have the IT folks email your company certificate to you, use `openssl s_client` to retrieve the certificate etc. When the certificate expires, you would update your application. Assuming your application has no bugs or security defects, the application would be updated every year or two.

At runtime, you retrieve the website or server's certificate in the callback. Within the callback, you compare the retrieved certificate with the certificate embedded within the program. If the comparison fails, then fail the method or function.



There is a downside to pinning a certificate. If the site rotates its certificate on a regular basis, then your application would need to be updated regularly. For example, Google rotates its certificates, so you will need to update your application about once a month (if it depended on Google services). Even though Google rotates its certificates, the underlying public keys (within the certificate) remain static.

Public Key

Public key pinning is more flexible but a little trickier due to the extra steps necessary to extract the public key from a certificate. As with a certificate, the program checks the extracted public key with its embedded copy of the public key.

There are two downsides to public key pinning. First, it's harder to work with keys (versus certificates) since you usually must extract the key from the certificate. Extraction is a minor inconvenience in Java and .Net, but it's uncomfortable in Cocoa/CocoaTouch and OpenSSL. Second, the key is static and may violate key rotation policies.



Hashing

While the three choices above used DER encoding, it's also acceptable to use a hash of the information (or other transforms). In fact, the original sample programs were written using digested certificates and public keys. The samples were changed to allow a programmer to inspect the objects with tools like `dumpasn1` and other ASN.1 decoders.

Hashing also provides three additional benefits. First, hashing allows you to anonymize a certificate or public key. This might be important if your application is concerned about leaking information during decompilation and re-engineering.

Second, a digested certificate fingerprint is often available as a native API for many libraries, so its convenient to use.

Finally, an organization might want to supply a reserve (or back-up) identity in case the primary identity is compromised. Hashing ensures your adversaries do not see the reserved certificate or public key in advance of its use. In fact, Google's IETF draft *websec-key-pinning* uses the technique.

What About X509?

PKI{X} and the Internet form an intersection. What Internet users expect and what they receive from CAs could vary wildly. For example, an Internet user has security goals, while a CA has revenue goals and legal goals. Many are surprised to learn that the user is often required to perform host identity verification even though the CA issued the certificate (the details are buried in CA warranties on their certificates and their Certification Practice Statement (CPS)).

There are a number of PKI profiles available. For the Internet, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL)", also known as RFC 5280 (<http://tools.ietf.org/rfc/rfc5280.txt>), is of interest. Since a certificate is specified in the ITU's X509 standard, there are lots of mandatory and optional fields available for validation from both bodies. Because of the disjoint goals among groups, the next section provides guidance.

Mandatory Checks

All X509 verifications must include:

- A path validation check. The check verifies all the signatures on certificates in the chain are valid under a given PKI. The check begins at the server or service's certificate (the leaf), and proceeds back to a trusted root certificate (the root).
- A validity check, or the `notBefore` and `notAfter` fields. The `notAfter` field is especially important since a CA will not warrant the certificate after the date, and it does not have to provide CRL/OCSP updates after the date.
- Revocation status. As with `notAfter`, revocation is important because the CA will not warrant a certificate once it is listed as revoked. The IETF approved way of checking a certificate's revocation is OCSP and specified in RFC 2560 (<http://tools.ietf.org/rfc/rfc2560.txt>).

Optional Checks

[Mulling over what else to present, and the best way to present it. Subject name? DNS lookups? Key Usage? Algorithms? Geolocation based on IP? Check back soon.] In the model which pre-dated PKIX RFC-5280, X.509v1 there was strong binding of the certificate Subject name to the X.500 Directory. With the update to X.509v3, the Directory is still the standard for authentication of caCertificate attributes, versus accepting a self signed root. Geo-location is important, the fake certificate for Google was given a location of Florida, instead of Mountain View, CA. The binding of the certificate to the Directory can anchor the root caCertificate, in effect "pin" it, to a valid entity that can have demonstrable attributes such as location. This is detailed in RFC-1255. Additional fields specified, such as the subject alternative field, for example a RFC-822 email address, or DNS name, can be located in the DNS, but the actual heavy lifting is done by the X.500 Directory, which is used currently as a cross-certificate trust conduit at the Federal Bridge between major communities of interest, that are not Internet focused. While those cross-certificates are valuable in validation between trust communities, a self-signed root, still needs to be either pinned, curated in trust bundle such as in web browser software secure storage or represented by a federated community. The Directory can play a role to fill in gaps to validate caCertificates, either locally, or nationally under an administrative domain such as `c=US`. By divorcing the subject from the Directory entry, problems begin to arise in which pinning plays a key role to ensure that client and server have the same reference points.

Public Key Checks

Quod vide (q.v.). Verifying the identity of a host with knowledge of its associated/expected public key is pinning.

Examples of Pinning

This section demonstrates certificate and public key pinning in Android Java, iOS, .Net, and OpenSSL. All programs attempt to connect to random.org (<https://www.random.org>) and fetch bytes (Dr. Mads Haahr participates in AOSP's pinning program, so the site should have a static key). The programs enjoy a pre-existing relationship with the site (more correctly, *a priori* knowledge), so they include a copy of the site's public key and pin the identity on the key.

Parameter validation, return value checking, and error checking have been omitted in the code below, but is present in the sample programs. So the sample code is ready for copy/paste. By far, the most uncomfortable languages are C-based: iOS and OpenSSL.

HTTP pinning

RFC 7469 (<http://www.rfc-editor.org/rfc/rfc7469.txt>) introduced a new HTTP header that allows SSL servers to declare hashes of their certificates with time scope in which these certificates should not be changed. For example:

```
Public-Key-Pins: max-age=2592000;
pin-sha256="E9CZ9INDbd+2eRQozYqgbQ2yXLVKB9+xcprMF+44U1g=";
pin-sha256="LPJNul+wow4m6DsQxbninhSWHlwfp0JecwQzYpOLmCQ=";
report-uri="http://example.com/pkp-report"
```

Please note that RFC 7469 (<http://www.rfc-editor.org/rfc/rfc7469.txt>) is controversial since it allows overrides for locally installed authorities. That is, it allows an adversary or other party who successfully phishes the user to override a known good pinset with non-authentic or fraudulent information. Second, the reporting mechanism is suppressed from broken pinsets, so a complying user agent will be complicit in the cover up after the fact. That is, the reporting of the broken pinset is called out as **MUST NOT** report [1 (https://en.wikipedia.org/w/index.php?title=HTTP_Public_Key_Pinning)].

Android

This example is using the concept from [developer.android.com unknown CA implementation document](https://developer.android.com/training/articles/security-ssl.html#UnknownCa) (<https://developer.android.com/training/articles/security-ssl.html#UnknownCa>). Basically you can teach `HttpsURLConnection` to trust a specific set of CAs.

```
public class KeyPinStore {
    private static KeyPinStore instance = null;
    private SSLContext sslContext = SSLContext.getInstance("TLS");

    public static synchronized KeyPinStore getInstance() throws CertificateException, IOException, KeyStoreException, NoSuchAlgorithmException {
        if (instance == null){
            instance = new KeyPinStore();
        }
        return instance;
    }

    private KeyPinStore() throws CertificateException, IOException, KeyStoreException, NoSuchAlgorithmException, KeyManagementException {
        // Load CAs from an InputStream
        // (could be from a resource or ByteArrayInputStream or ...)
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        // randomCA.crt should be in the Assets directory
        InputStream caInput = new BufferedInputStream(MainActivity.context.getAssets().open("randomCA.crt"));
        Certificate ca;
        try {
            ca = cf.generateCertificate(caInput);
            System.out.println("ca=" + ((X509Certificate) ca).getSubjectDN());
        } finally {
            caInput.close();
        }

        // Create a KeyStore containing our trusted CAs
        String keyStoreType = KeyStore.getDefaultType();
        KeyStore keyStore = KeyStore.getInstance(keyStoreType);
        keyStore.load(null, null);
        keyStore.setCertificateEntry("ca", ca);

        // Create a TrustManager that trusts the CAs in our KeyStore
        String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
        TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
        tmf.init(keyStore);

        // Create an SSLContext that uses our TrustManager
        // SSLContext context = SSLContext.getInstance("TLS");
        sslContext.init(null, tmf.getTrustManagers(), null);
    }

    public SSLContext getContext(){
        return sslContext;
    }
}
```

You can check this Android app `pubkey-pin-android` (<https://github.com/riramar/pubkey-pin-android>) on Github as full example.

iOS

An open-source SSL pinning library for iOS and OS X was released at Black Hat 2015, which provides an easy-to-use API for deploying pinning within an App: <https://github.com/datatheorem/TrustKit>.

Otherwise and when using `NSURLConnection`, iOS pinning is performed through a `NSURLConnectionDelegate`. The delegate must implement `connection:canAuthenticateAgainstProtectionSpace:` and `connection:didReceiveAuthenticationChallenge:`. Within `connection:didReceiveAuthenticationChallenge:`, the delegate must call `SecTrustEvaluate` to perform customary X509 checks.

Download: iOS sample program.

```

-(IBAction)fetchButtonTapped:(id)sender
{
    NSString* requestString = @"https://www.random.org/integers/?
        num=16&min=0&max=255&col=16&base=16&format=plain&rnd=new";
    NSURL* requestUrl = [NSURL URLWithString:requestString];

    NSURLRequest* request = [NSURLRequest requestWithURL:requestUrl
        cachePolicy:NSURLRequestReloadIgnoringLocalCacheData
        timeoutInterval:10.0f];

    NSURLConnection* connection = [[NSURLConnection alloc] initWithRequest:request delegate:self];
}

-(BOOL)connection:(NSURLConnection *)connection canAuthenticateAgainstProtectionSpace:
    (NSURLProtectionSpace*)space
{
    return [[space authenticationMethod] isEqualToString: NSURLAuthenticationMethodServerTrust];
}

-(void)connection:(NSURLConnection *)connection didReceiveAuthenticationChallenge:
    (NSURLAuthenticationChallenge *)challenge
{
    if ([[challenge protectionSpace] authenticationMethod] isEqualToString: NSURLAuthenticationMethodServerTrust))
    {
        do
        {
            SecTrustRef serverTrust = [[challenge protectionSpace] serverTrust];
            if(nil == serverTrust)
                break; /* failed */

            OSStatus status = SecTrustEvaluate(serverTrust, NULL);
            if(!(errSecSuccess == status))
                break; /* failed */

            SecCertificateRef serverCertificate = SecTrustGetCertificateAtIndex(serverTrust, 0);
            if(nil == serverCertificate)
                break; /* failed */

            CFDataRef serverCertificateData = SecCertificateCopyData(serverCertificate);
            [(id)serverCertificateData autorelease];
            if(nil == serverCertificateData)
                break; /* failed */

            const UInt8* const data = CFDataGetBytePtr(serverCertificateData);
            const CFIndex size = CFDataGetLength(serverCertificateData);
            NSData* cert1 = [NSData dataWithBytes:data length:(NSUInteger)size];

            NSString *file = [[NSBundle mainBundle] pathForResource:@"random-org" ofType:@"der"];
            NSData* cert2 = [NSData dataWithContentsOfFile:file];

            if(nil == cert1 || nil == cert2)
                break; /* failed */

            const BOOL equal = [cert1 isEqualToData:cert2];
            if(!equal)
                break; /* failed */

            // The only good exit point
            return [[challenge sender] useCredential: [NSURLCredential credentialForTrust: serverTrust]
                forAuthenticationChallenge: challenge];
        } while(0);

        // Bad dog
        return [[challenge sender] cancelAuthenticationChallenge: challenge];
    }
}

```

.Net

.Net pinning can be achieved by using `ServicePointManager` as shown below.

Download: .Net sample program.

```

// Encoded RSAPublicKey
private static String PUB_KEY = "30818902818100C4A06B7B52F8D17DC1CCB47362" +
    "C64AB799AAE19E245A7559E9CEEC7D8AA4DF07CB0B21FDFD763C63A313A668FE9D764E" +
    "D913C51A676788DB62AF624F422C2F112C1316922AA5D37823CD9F43D1FC54513D14B2" +
    "9E36991F08A042C42EAAEE5FE8E2CB10167174A359CEBF6FACC2C9CA933AD403137EE" +

```

```

    "2C3F4CBED9460129C72B0203010001";

public static void Main(string[] args)
{
    ServicePointManager.ServerCertificateValidationCallback = PinPublicKey;
    WebRequest wr = WebRequest.Create("https://encrypted.google.com/");
    wr.GetResponse();
}

public static bool PinPublicKey(object sender, X509Certificate certificate, X509Chain chain,
                                SslPolicyErrors sslPolicyErrors)
{
    if (null == certificate)
        return false;

    String pk = certificate.GetPublicKeyString();
    if (pk.Equals(PUB_KEY))
        return true;

    // Bad dog
    return false;
}

```

OpenSSL

Pinning can occur at one of two places with OpenSSL. First is the user supplied `verify_callback`. Second is after the connection is established via `SSL_get_peer_certificate`. Either method will allow you to access the peer's certificate.

Though OpenSSL performs the X509 checks, you must fail the connection and tear down the socket on error. By design, a server that does not supply a certificate will result in `x509_v_ok` with a **NULL** certificate. To check the result of the customary verification: (1) you must call `SSL_get_verify_result` and verify the return code is `x509_v_ok`; and (2) you must call `SSL_get_peer_certificate` and verify the certificate is **non-NULL**.

Download: OpenSSL sample program.

```

int pkp_pin_peer_pubkey(SSL* ssl)
{
    if(NULL == ssl) return FALSE;

    X509* cert = NULL;
    FILE* fp = NULL;

    /* Scratch */
    int len1 = 0, len2 = 0;
    unsigned char *buff1 = NULL, *buff2 = NULL;

    /* Result is returned to caller */
    int ret = 0, result = FALSE;

    do
    {
        /* http://www.openssl.org/docs/ssl/SSL_get_peer_certificate.html */
        cert = SSL_get_peer_certificate(ssl);
        if(!(cert != NULL))
            break; /* failed */

        /* Begin Gyration to get the subjectPublicKeyInfo */
        /* Thanks to Viktor Dukhovni on the OpenSSL mailing list */

        /* http://groups.google.com/group/openssl.users/browse_thread/thread/d61858dae102c6c7 */
        len1 = i2d_X509_PUBKEY(X509_get_X509_PUBKEY(cert), NULL);
        if(!(len1 > 0))
            break; /* failed */

        /* scratch */
        unsigned char* temp = NULL;

        /* http://www.openssl.org/docs/crypto/buffer.html */
        buff1 = temp = OPENSSL_malloc(len1);
        if(!(buff1 != NULL))
            break; /* failed */

        /* http://www.openssl.org/docs/crypto/d2i_X509.html */
        len2 = i2d_X509_PUBKEY(X509_get_X509_PUBKEY(cert), &temp);

        /* These checks are verifying we got back the same values as when we sized the buffer. */
        /* Its pretty weak since they should always be the same. But it gives us something to test. */
        if(!(len1 == len2) && (temp != NULL) && ((temp - buff1) == len1))
            break; /* failed */

        /* End Gyration */

        /* See the warning above!!! */
        /* http://pubs.opengroup.org/onlinepubs/009696699/functions/fopen.html */
        fp = fopen("random-org.der", "rx");
    } while(0);
}

```



```

if(NULL ==fp) {
    fp = fopen("random-org.der", "r");

if(!(NULL != fp))
    break; /* failed */

/* Seek to eof to determine the file's size */
/* http://pubs.opengroup.org/onlinepubs/009696699/functions/fseek.html */
ret = fseek(fp, 0, SEEK_END);
if(! (0 == ret))
    break; /* failed */

/* Fetch the file's size */
/* http://pubs.opengroup.org/onlinepubs/009696699/functions/ftell.html */
long size = ftell(fp);

/* Arbitrary size, but should be relatively small (less than 1K or 2K) */
if!(size != -1 && size > 0 && size < 2048))
    break; /* failed */

/* Rewind to beginning to perform the read */
/* http://pubs.opengroup.org/onlinepubs/009696699/functions/fseek.html */
ret = fseek(fp, 0, SEEK_SET);
if(! (0 == ret))
    break; /* failed */

/* Re-use buff2 and len2 */
buff2 = NULL; len2 = (int)size;

/* http://www.openssl.org/docs/crypto/buffer.html */
buff2 = OPENSSL_malloc(len2);
if(!(buff2 != NULL))
    break; /* failed */

/* http://pubs.opengroup.org/onlinepubs/009696699/functions/fread.html */
/* Returns number of elements read, which should be 1 */
ret = (int)fread(buff2, (size_t)len2, 1, fp);
if!(ret == 1))
    break; /* failed */

/* Re-use size. MIN and MAX macro below... */
size = len1 < len2 ? len1 : len2;

/***** PAYDIRT *****/
/***** PAYDIRT *****/
if(len1 != (int)size || len2 != (int)size || 0 != memcmp(buff1, buff2, (size_t)size))
    break; /* failed */

/* The one good exit point */
result = TRUE;

} while(0);

if(fp != NULL)
    fclose(fp);

/* http://www.openssl.org/docs/crypto/buffer.html */
if(NULL != buff2)
    OPENSSL_free(buff2);

/* http://www.openssl.org/docs/crypto/buffer.html */
if(NULL != buff1)
    OPENSSL_free(buff1);

/* http://www.openssl.org/docs/crypto/X509_new.html */
if(NULL != cert)
    X509_free(cert);

return result;
}

```

Pinning Alternatives

Not all applications use split key cryptography. Fortunately, there are protocols which allow you to set up a secure channel based on knowledge of passwords and pre-shared secrets (rather than putting the secret on the wire in a basic authentication scheme). Two are listed below - SRP and PSK. SRP and PSK have 88 cipher suites assigned to them by IANA for TLS (<http://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-3>), so there's no shortage of choices.

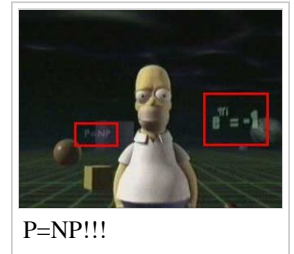
Hex Code	Cipher Suite Name	Status	Reference
0xC0, 0x1A	TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA	Y	[RFC5054]
0xC0, 0x1B	TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA	Y	[RFC5054]
0xC0, 0x1C	TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA	Y	[RFC5054]
0xC0, 0x1D	TLS_SRP_SHA_WITH_AES_128_CBC_SHA	Y	[RFC5054]
0xC0, 0x1E	TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA	Y	[RFC5054]
0xC0, 0x1F	TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA	Y	[RFC5054]
0xC0, 0x20	TLS_SRP_SHA_WITH_AES_256_CBC_SHA	Y	[RFC5054]
0xC0, 0x21	TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA	Y	[RFC5054]
0xC0, 0x22	TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA	Y	[RFC5054]
0xC0, 0x23	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	Y	[RFC5289]
0xC0, 0x24	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	Y	[RFC5289]
0xC0, 0x25	TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	Y	[RFC5289]
0xC0, 0x26	TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	Y	[RFC5289]

Figure 3: IANA reserved cipher suites for SRP and PSK

SRP

Secure Remote Password (SRP) is a Password Authenticated Key Exchange (PAKE) by Thomas Wu based upon Diffie-Hellman. The protocol is standardized in RFC 5054 (<https://tools.ietf.org/rfc/rfc5054.txt>) and available in the OpenSSL library (among others). In the SRP scheme, the server uses a verifier which consists of a $\{\text{salt}, \text{hash}(\text{password})\}$ pair. The user has the password and receives the salt from the server. With lots of hand waving, both parties select per-instance random values (nonces) and execute the protocol using $g^{\{(salt + password)/verifier\} + nonces}$ rather than traditional Diffie-Hellman using g^{ab} .

Diffie-Hellman based schemes are part of a family of problems based on Discrete Logs (DL), which are logarithms over a finite field. DL schemes are appealing because they are known to be hard (unless $P=NP$, which would cause computational number theorists to have a cow).



PSK

PSK is Pre-Shared Key and specified in RFC 4279 (<https://tools.ietf.org/rfc/rfc4279.txt>) and RFC 4764 (<https://tools.ietf.org/rfc/rfc4764.txt>). The shared secret is used as a pre-master secret in TLS-PSK for SSL/TLS; or used to key a block cipher in EAP-PSK. EAP-PSK is designed for authentication over insecure networks such as IEEE 802.11.

Miscellaneous

This sections covers administrivia and miscellaneous items related to pinning.

Ephemeral Keys

Ephemeral keys are temporary keys used for one instance of a protocol execution and then thrown away. An ephemeral key has the benefit of providing forward secrecy, meaning a compromise of the site or service's long term (static) signing key does not facilitate decrypting past messages because the key was temporary and discarded (once the session terminated).

Ephemeral keys do not affect pinning because the Ephemeral key is delivered in a separate `serverKeyExchange` message. In addition, the ephemeral key is a key and not a certificate, so it does not change the construction of the certificate chain. That is, the certificate of interest will still be located at `certificates[0]`.

Pinning Gaps

There are two gaps when pinning due to reuse of the existing infrastructure and protocols. First, an explicit challenge is **not** sent by the program to the peer server based on the server's public information. So the program never knows if the peer can actually decrypt messages. However, the shortcoming is usually academic in practice since an adversary will receive messages it can't decrypt.

Second is revocation. Clients don't usually engage in revocation checking, so it could be possible to use a known bad certificate or key in a pinset. Even if revocation is active, Certificate Revocation Lists (CRLs) and Online Certificate Status Protocol (OCSP) can be defeated in a hostile environment. An application can take steps to remediate, with the primary means being freshness. That is, an application should be updated and distributed immediately when a critical security parameter changes.

No Relationship ^@\$!

If you don't have a pre-existing relationship, all is not lost. First, you can pin a host or server's certificate or public key the first time you encounter it. If the bad guy was not active when you encountered the certificate or public key, he or she will not be successful with future funny business.

Second, bad certificates are being spotted quicker in the field due to projects like Chromium (<http://www.chromium.org>) and Certificate Patrol (<https://addons.mozilla.org/en-us/firefox/addon/certificate-patrol/>), and initiatives like the EFF's SSL Observatory (<https://www.eff.org/observatory>).

Third, help is on its way, and there are a number of futures that will assist with the endeavors:

- Public Key Pinning (<http://www.ietf.org/id/draft-ietf-websec-key-pinning-09.txt>) – an extension to the HTTP protocol allowing web host operators to instruct user agents (UAs) to remember ("pin") the hosts' cryptographic identities for a given period of time.
- DNS-based Authentication of Named Entities (DANE) (<https://datatracker.ietf.org/doc/rfc6698/>) - uses Secure DNS to associate Certificates with Domain Names For S/MIME, SMTP with TLS, DNSSEC and TLSA records.
- Sovereign Keys (<http://www.eff.org/sovereign-keys>) - operates by providing an optional and secure way of associating domain names with public keys via DNSSEC. PKI (hierarchical) is still used. Semi-centralized with append only logging.
- Convergence (<http://convergence.io>) – different [geographical] views of a site and its associated data (certificates and public keys). Web of Trust is used. Semi-centralized.

While Sovereign Keys and Convergence still require us to confer trust to outside parties, the parties involved do not serve share holders or covet revenue streams. Their interests are industry transparency and user security.

More Information?

Pinning is an *old new thing* that has been shaken, stirred, and repackaged. While "pinning" and "pinsets" are relatively new terms for old things, Jon Larimer and Kenny Root spent time on the subject at Google I/O 2012 with their talk *Security and Privacy in Android Apps* (<https://developers.google.com/events/io/sessions/gooio2012/107/>).

Format Conversions

As a convenience to readers, the following with convert between PEM and DER format using OpenSSL.

```
# Public key, X509
$ openssl genrsa -out rsa-openssl.pem 3072
$ openssl rsa -in rsa-openssl.pem -pubout -outform DER -out rsa-openssl.der

# Private key, PKCS#8
$ openssl genrsa -out rsa-openssl.pem 3072
$ openssl pkcs8 -nocrypt -in rsa-openssl.pem -inform PEM -topk8 -outform DER -out rsa-openssl.der
```

References

- OWASP Injection Theory
- OWASP Data Validation
- OWASP Transport Layer Protection Cheat Sheet
- IETF Public Key Pinning (<http://www.ietf.org/id/draft-ietf-websec-key-pinning-09.txt>)
- IETF RFC 5054 (SRP) (<http://www.ietf.org/rfc/rfc5054.txt>)
- IETF RFC 4764 (EAP-PSK) (<http://www.ietf.org/rfc/rfc4764.txt>)
- IETF RFC 1421 (PEM Encoding) (<http://www.ietf.org/rfc/rfc1421.txt>)
- IETF RFC 5280 (Internet X.509, PKIX) (<http://www.ietf.org/rfc/rfc5280.txt>)
- IETF RFC 4648 (Base16, Base32, and Base64 Encodings) (<http://www.ietf.org/rfc/rfc4648.txt>)
- IETF RFC 3279 (PKI, X509 Algorithms and CRL Profiles) (<http://www.ietf.org/rfc/rfc3279.txt>)
- IETF RFC 4055 (PKI, X509 Additional Algorithms and CRL Profiles) (<http://www.ietf.org/rfc/rfc4055.txt>)
- IETF RFC 2246 (TLS 1.0) (<http://www.ietf.org/rfc/rfc2246.txt>)
- IETF RFC 4346 (TLS 1.1) (<http://www.ietf.org/rfc/rfc4346.txt>)
- IETF RFC 5246 (TLS 1.2) (<http://www.ietf.org/rfc/rfc5246.txt>)
- IETF RFC 6698, Draft (DANE) (<http://www.ietf.org/rfc/rfc6698.txt>)
- EFF Sovereign Keys (<http://www.eff.org/sovereign-keys>)
- Thoughtcrime Labs Convergence (<http://convergence.io/>)
- RSA Laboratories PKCS#1, RSA Encryption Standard (<http://www.rsa.com/rsalabs/node.asp?id=2125>)
- RSA Laboratories PKCS#6, Extended-Certificate Syntax Standard (<http://www.rsa.com/rsalabs/node.asp?id=2128>)
- ITU Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) (<http://www.itu.int/rec/T-REC-X.690-200811-I/en>)

- TOR Project Detecting Certificate Authority Compromises and Web Browser Collusion (<https://blog.torproject.org/blog/detecting-certificate-authority-compromises-and-web-browser-collusion>)
- Code Project Cryptographic Interoperability: Keys (<http://www.codeproject.com/Articles/25487/Cryptographic-Interoperability-Keys>)
- Google I/O Security and Privacy in Android Apps (<https://developers.google.com/events/io/sessions/gooio2012/107/>)
- Trevor Perrin Transparency, Trust Agility, Pinning (Recent Developments in Server Authentication) (<https://crypto.stanford.edu/RealWorldCrypto/slides/perrin.pdf>)
- Dr. Peter Gutmann's PKI is Broken (<http://www.cs.auckland.ac.nz/~pgut001/pubs/pkitutorial.pdf>)
- Dr. Matthew Green's The Internet is Broken (<http://blog.cryptographyengineering.com/2012/02/how-to-fix-internet.html>)
- Dr. Matthew Green's How do Interception Proxies fail? (<http://blog.cryptographyengineering.com/2012/03/how-do-interception-proxies-fail.html>)
- Presentation: SSL Pinning implementation and bypasses for iOS and Android (<http://www.slideshare.net/anantshri/ssl-pinning-and-bypasses-android-and-ios>)

Authors and Primary Editors

- Jeffrey Walton - [jeffrey, owasp.org](mailto:jeffrey@owasp.org)
- JohnSteven - [john, owasp.org](mailto:john@owasp.org)
- Jim Manico - [jim, owasp.org](mailto:jim@owasp.org)
- Kevin Wall - [kevin, owasp.org](mailto:kevin@owasp.org)
- Ricardo Iramar - [ricardo.iramar, owasp.org](mailto:ricardo.iramar@owasp.org)

Retrieved from "https://www.owasp.org/index.php?title=Certificate_and_Public_Key_Pinning&oldid=218535"

Category: Control

-
- This page was last modified on 5 July 2016, at 02:00.
 - Content is available under Creative Commons Attribution-ShareAlike unless otherwise noted.