Using Networking Securely

Whether you are writing a banking app or a game, if your program uses networking, it should be secure. For all but the most trivial pieces of data, it's impossible for software to determine whether a user's data is confidential, embarrassing, or even dangerous. Large numbers of seemingly insignificant pieces of information can, in aggregate, be a much greater concern than the sum of its parts.

For these reasons, always assume that every piece of data your program encounters could contain a bank account number or a password, and secure it accordingly.

Some attacks your program might encounter include:

- Snooping—Attacks in which a third party sniffs your program's data in transit.
- Man-in-the-middle attacks—Attacks in which a third party interposes its own computer between your program and a server. Man-in-the-middle attacks include:

  Spoofing and phishing—Creating false servers that masquerade as legitimate servers.
  Tampering—Modifying data between the server and your program.
  Session hijacking—Capturing authentication information and using it to pose as your users.

- Injection attacks—Attacks in which specially crafted data can cause client or server software to execute commands other than the inspected ones. This commonly occurs when the program talks to a script interpreter, such as a shell or a SQL database server.
- Buffer overflows and numeric overflows—Attacks in which specially crafted data can cause a program to read or write data in parts of its address space where it shouldn't, potentially leading to execution of arbitrary executable code, disclosure of private information, or both.

This chapter explains how to defend against snooping and man-in-the-middle attacks. To learn more about injection attacks, buffer overflows, and other aspects of software security, read *Secure Coding Guide*.

Enabling TLS or SSL

The Transport Layer Security (TLS) protocol provides data encryption for socket-based communication, along with authentication of servers and (optionally) clients to prevent spoofing.

OS X and iOS also provide support for the Secure Sockets Layer (SSL) protocol. Because TLS is the successor to SSL, OS X and iOS use TLS by default if both protocols are supported.

> **Note:** TLS and SSL are primarily designed for use in a client-server model. It's more difficult to ensure secure communication in a peer-to-peer environment with these protocols.

Connecting Securely to a URL

Connecting to a URL via TLS is trivial. When you create an `NSURLRequest` object to provide to the `initWithRequest:delegate:` method, specify `https` as the scheme of the URL instead of `http`. The connection uses TLS automatically with no additional configuration.

Connecting Securely Using Streams

You can use TLS with an `NSStream` object by calling `setProperty:forKey:` on it. Specify `NSStreamSocketSecurityLevelNegotiatedSSL` as the `property` parameter and `NSStreamSocketSecurityLevelKey` as the `key` parameter. If you need to work around compatibility bugs, you can also specify a more specific protocol, such as `NSStreamSocketSecurityLevelTLSv1`.

Connecting Securely Using BSD Sockets

When making secure connections, if possible, you should use `NSStream` (as described in the previous section) instead of using sockets directly. However, if you must work with BSD sockets directly, you must perform the SSL or TLS encryption and decryption yourself. Depending on your platform, there are two ways to do this:

- In OS X, or in iOS 5 and later, you can use the Secure Transport API in the Security framework to handle your SSL and TLS handshaking, encryption, and decryption. See *Secure Transport Reference* for details.
- In iOS and OS X, you can download an open source SSL or TLS implementation, such as OpenSSL and include a compiled copy of that library (or some portion thereof) in your app bundle (or alongside your nonbundled program). Be sure to comply with the licensing terms of any third-party libraries you might use.

Note: Although a version of OpenSSL libraries is included as part of OS X, the OpenSSL library does not guarantee binary compatibility across different versions of OpenSSL. For this reason, linking to the built-in copy of OpenSSL is deprecated as of OS X v10.7. If you want to use OpenSSL, provide your own copy of the library so that you can control precisely which version of OpenSSL your program is linked against.

Using Other Security Protocols in OS X

In addition to the default Secure Transport implementation of TLS, the following network security protocols are available in OS X:

- The Kerberos protocol is available via the Kerberos framework. This protocol provides support for single sign-on authentication over a network. For more information, read *Security Overview* and *Authentication, Authorization, and Permissions Guide*.
- The Secure Shell (SSH) protocol is available. This protocol is commonly used for logging in to remote hosts using the Terminal app. See `ssh` for more information.
- The OpenSSL implementation of TLS is available, but the preinstalled OpenSSL library is deprecated in OS X v10.7 and later for binary compatibility reasons. If you require OpenSSL, provide your own copy of this library instead, and statically link it into your program.

Common Mistakes

There are a number of common mistakes developers make when writing secure networking code. This section provides suggestions for avoiding several of these mistakes.

Be Careful Who You Trust

If your app sends or receives potentially confidential data to or from a server, be certain that it authenticates the server to ensure that it has not been spoofed. Be sure your server authenticates the client correctly to avoid providing data to the wrong user. Also, be certain that the connection is established using appropriate encryption.

Similarly, be sure that you store data only when necessary and provide it only to the minimum extent necessary to perform a task. For example, to maximize privacy of users' personal information, you might store your web servers' databases on separate servers, configured to accept SQL queries only from your web servers, and with limited connectivity to the Internet as a whole. In other words, use proper privilege separation.

For more information, read Designing Secure Helpers and Daemons in *Secure Coding Guide*.

Be Careful What Data You Trust

Every program is at risk of attack by someone providing malformed or malicious content. This is particularly true if your program obtains data from untrusted servers, or if your program obtains untrusted data from trusted servers (forum posts, for example).

To protect against such attacks, your program should carefully examine all data received from the network or from disk (because the user might have downloaded that data). If the data appears malformed in any way, do not process it as you would other data.

For more information, read Validating Input and Interprocess Communication in *Secure Coding Guide*.

Know That Many Tiny Leaks Can Add Up to a Flood

Always take steps to ensure that the contents of your app's Internet traffic remains private. Although certain information may seem harmless by itself, a skilled attacker can combine that information with other information to discover trends that might be a far greater cause for concern than any single data point by itself—a process known as *data mining*.

For example, if someone wants to break into your house, a single post from the library on a Saturday evening is probably harmless, but posts from the library at about the same time of day every Saturday for several weeks in a row might not be so harmless, because they indicate your habits.

The data need not even be all about the same thing to cause harm. Knowing that someone likes to watch a particular TV show might be harmless, but a complete profile of the sorts of shows that someone enjoys, the products he or she buys, and the friends he or she interacts with might correlate strongly with some attribute that the person considers private, such as religion or sexual orientation. In one particularly impressive (and possibly apocryphal) story, a retail chain reportedly recognized that a man's daughter was pregnant based on her purchasing decisions even before her father did.

The risk of disclosure in aggregate is particularly problematic when it comes to things like identity theft. Your app might leak a phone number, another app might leak a postal address, and so on. After the attacker has amassed enough information about a victim, he or she can use social engineering techniques to convince a third party to give him or her even more information, resulting in a feedback loop of information gathering with devastating consequences.

For this reason, it's important that your app use encrypted communication at all times, for all connections, unless it is infeasible to do so. You never know when that seemingly harmless piece of information, when combined with another seemingly harmless piece of information, might prove damaging or hurtful.

Install Certificates Correctly

When connecting to a server using TLS or SSL, if your app gets an error saying that the certificate authority that signed your certificate is unknown, assuming that you obtained your certificate from a reputable certificate authority, this almost invariably means your certificate chain is missing or incomplete.

When your server accepts a connection encrypted with TLS or SSL, it provides two things: your server's SSL certificate and a complete chain of SSL certificates, beginning with your server's certificate and ending with a certificate signed by one of the trusted anchor certificates recognized by the operating system. If there are certificates missing in your chain, you'll get this error because the certificates earlier in the chain cannot be verified without the certificates later in the chain.

To see what your server is actually sending out, type the following command in Terminal (replacing `www.example.com` with your actual domain name) and press Return:

```
openssl s_client -showcerts -connect www.example.com:443
```

When you type this command, you should see your server's certificate, followed by a series of intermediate certificates. If you don't, check your server's configuration. To obtain the correct certificates to put in your server's certificate chain file, contact the certificate authority that provided your server's SSL certificate.

Never Disable Certificate Chain Validation (Unless You Validate Them Yourself)

Disabling chain validation eliminates any benefit you might otherwise have gotten from using a secure connection. The resulting connection is no safer than sending the request using unencrypted HTTP because it provides no protection from spoofing by a fake server.

If you are using server certificates from a trusted certificate authority, be sure your certificates are installed correctly (see the previous section).

If you are working with self-signed certificates temporarily, you should add them to your test machines' trusted anchors list. In OS X, you can do this using the Keychain Access utility. In iOS, you can use the `SecTrustCopyAnchorCertificates`, `SecTrustCreateWithCertificates`, and `SecTrustSetAnchorCertificates` functions within your program.

If you need to specifically allow a single self-signed certificate or a certificate signed for a different (specific) host, or if you need to allow a certificate only for a single connection, you can learn safe ways to do this by reading Overriding TLS Chain Validation Correctly.