Over-the-Air Profile Delivery and Configuration

Developer

Contents

Figures and Listings

Introduction

A configuration profile is an XML file that allows you to distribute configuration information to iOS-based devices. If you need to configure a large number of devices or to provide lots of custom email settings, network settings, or certificates to a large number of devices, configuration profiles are an easy way to do it.

An iOS configuration profile contains a number of settings that you can specify, including:

- Passcode policies
- Restrictions on device features (disabling the camera, for example)
- Wi-Fi settings
- VPN settings
- Email server settings
- Exchange settings
- LDAP directory service settings
- CalDAV calendar service settings
- Web clips
- Credentials and keys
- Advanced cellular network settings

Note: Configuration profiles are in property list format, with data values stored in Base64 encoding. The `.plist` format can be read and written by any XML library.

For more information about the contents of these profiles, read *Configuration Profile Reference*.

There are four ways to deploy configuration profiles:

- By physically connecting the device
- In an email message
- On a webpage
- Using over-the air configuration as described in this document

iOS supports both encrypted and unencrypted profiles. Encrypted profiles guarantee data integrity and protect sensitive policy information from prying eyes. Encrypted configuration profiles are signed with the public key associated with a device's identity certificate. This public key can be obtained in one of two ways: by connecting via USB to a computer running the iPhone Configuration Utility (iPCU) or using over-the-air enrollment.

If it is practical to connect each device to a single computer before deployment, you can use the iPhone Configuration Utility (iPCU) to encrypt profiles specific to each device. Later, you can securely deliver updated profiles via email or a webpage. (If you don't care about encrypted profiles, you can use iPCU without connecting the devices.)

If this manual enrollment meets your needs, you should read "iPhone Configuration Utility" and other documents in the *Enterprise Deployment* subcategory instead.

Although the other methods offer a simple way to configure devices for enterprise use, in large-scale deployments, you'll want to automate the deployment process.

iOS over-the-air enrollment and configuration provides an automated way to configure devices securely within the enterprise. This process provides IT with assurance that only trusted users are accessing corporate services and that their devices are properly configured to comply with established policies. Because configuration profiles can be both encrypted and locked, the settings cannot be removed, altered, or shared with others.

More importantly, for geographically distributed enterprises, an over-the-air profile service allows you to enroll iOS-based devices without physically connecting them to an iPhone Configuration Utility host.

The profile service described in this document creates a configuration on the fly; the device then downloads that configuration. The device remembers the enrollment URL so that it can update its configuration from the server in the future if the configuration expires or a VPN connection failure occurs.

This document describes the over-the-air enrollment process. With this process, administrators can instruct users to begin the process of enrollment by providing a URL via email or SMS notification. When users agree to the profile installation, their devices are automatically enrolled and configured in a single session.

Organization of This Document

This document takes you through the process of setting up a server to deliver encrypted custom profiles to iOS-based devices over the air.

- "Over-the-Air Profile Delivery Concepts" (page 8) explains the terminology and basic security concepts involved in over-the-air enrollment and profile delivery.

- "Creating a Profile Server for Over-The-Air Enrollment and Configuration" (page 14) describes the reference implementation of a profile server, piece by piece, in chronological order of execution, from device authentication and enrollment to profile delivery.
- "Configuration Profile Examples" (page 30) provides sample profiles and code to generate profiles.

This document assumes a basic knowledge of Ruby programming, XML, property lists, the iPhone Configuration Utility, and OpenSSL.

See Also

For more information, see the following pages:

Cisco: Digital Certificates PKI for IPSec VPNs (PDF)

Wikipedia: Public key infrastructure

IETF SCEP protocol specification

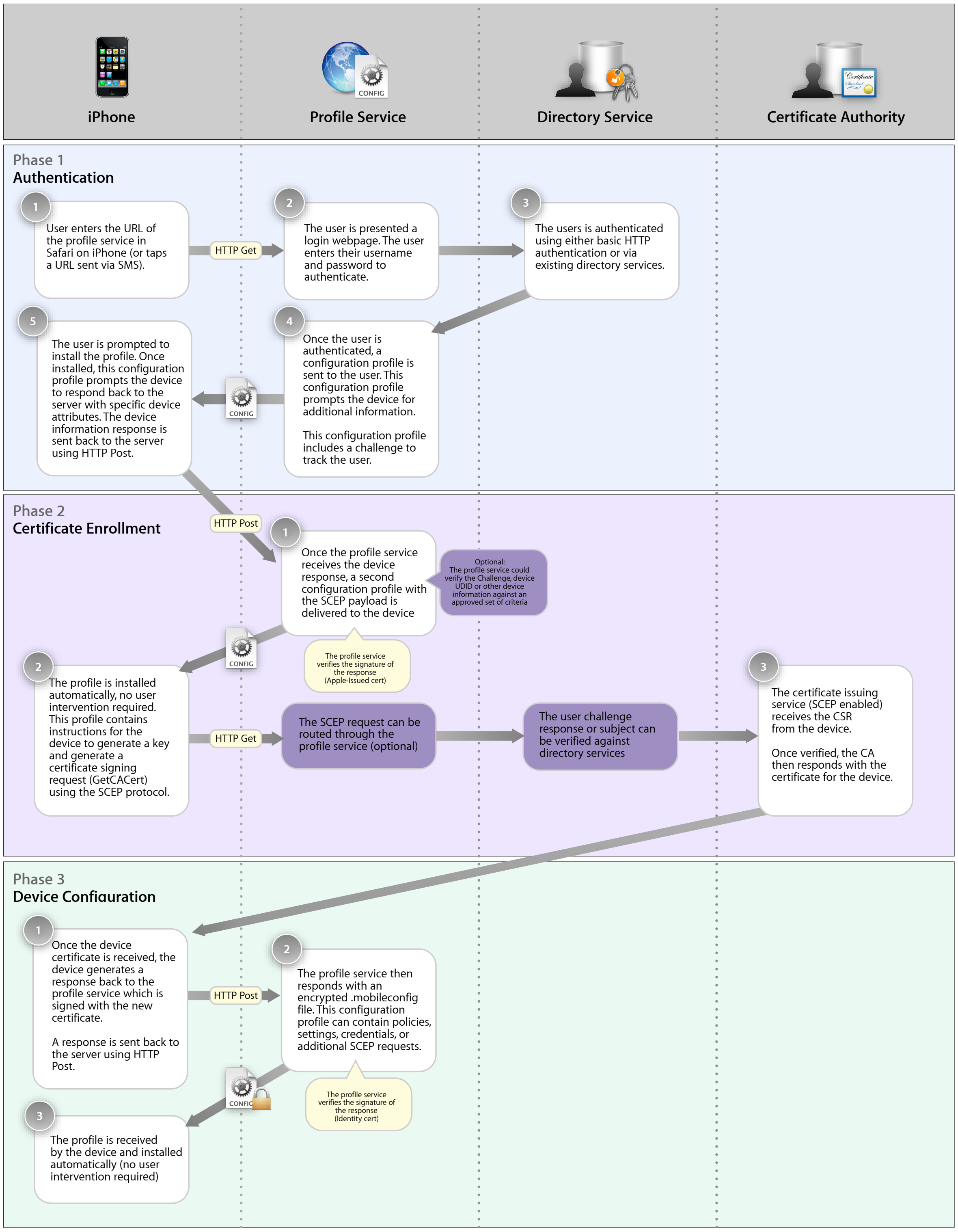
Additional information and resources for iOS-based devices in the enterprise are available at http://www.apple.com/iphone/business/, including *Configuration Profile Reference*. This appendix specifies the format of `.mobileconfig` files for developers who want to create their own tools.

Over-the-Air Profile Delivery Concepts

The process of over-the-air enrollment and configuration is divided into three phases: authentication, enrollment, and device configuration. These phases are described in the sections that follow.

This process is illustrated in "Directory Services."

Figure 1-1 Device registration process



Phase 1
Authentication

1 User enters the URL of the profile service in Safari on iPhone (or taps a URL sent via SMS).

2 The user is presented a login webpage. The user enters their username and password to authenticate.

3 The users is authenticated using either basic HTTP authentication or via existing directory services.

4 Once the user is authenticated, a configuration profile is sent to the user. This configuration profile prompts the device for additional information.

This configuration profile includes a challenge to track the user.

5 The user is prompted to install the profile. Once installed, this configuration profile prompts the device to respond back to the server with specific device attributes. The device information response is sent back to the server using HTTP Post.

Phase 2
Certificate Enrollment

1 Once the profile service receives the device response, a second configuration profile with the SCEP payload is delivered to the device

Optional:
The profile service could verify the Challenge, device UDID or other device information against an approved set of criteria

The profile service verifies the signature of the response (Apple-Issued cert)

2 The profile is installed automatically, no user intervention required. This profile contains instructions for the device to generate a key and generate a certificate signing request (GetCACert) using the SCEP protocol.

The SCEP request can be routed through the profile service (optional)

The user challenge response or subject can be verified against directory services

3 The certificate issuing service (SCEP enabled) receives the CSR from the device.

Once verified, the CA then responds with the certificate for the device.

Phase 3
Device Configuration

1 Once the device certificate is received, the device generates a response back to the profile service which is signed with the new certificate.

A response is sent back to the server using HTTP Post.

2 The profile service then responds with an encrypted .mobileconfig file. This configuration profile can contain policies, settings, credentials, or additional SCEP requests.

The profile service verifies the signature of the response (Identity cert)

3 The profile is received by the device and installed automatically (no user intervention required)

Phase 1: Authentication

The authentication phase serves two purposes. First, it ensures that incoming enrollment requests are from authorized users. Second, it captures information about the user's device for use in the certificate enrollment process.

When enrolling a device, the server can require authentication by the user, the device, or both.

User authentication can be enforced at the time the user goes to the enroll URL. To authenticate the user, you can use any web authentication scheme, whether part of HTTP (for example, basic auth or NTML) or a separate authentication scheme implemented by a CGI script. You can even use a hybrid scheme, such as combining digest authentication with a CGI-managed list of authorized users.

You can also check the device against a list of authorized devices if desired.

The steps in the authentication phase are:

1. The user visits the root URL (`/`) and gets a welcome message. This message is provided by the handler described in "Welcome Page (/) URL Handler" (page 17).
2. The user visits the certificate authority URL (`/CA`) to get the root cert (if needed). On the server, this URL is handled by code described in "Root Certificate (/CA) URL Handler" (page 18).) This is only required for self-signed root CA certs.
3. The user visits the enrollment URL (`/enroll`) to start the enrollment process. In this step, the user is prompted to authenticate himself or herself using HTTP basic auth (in this example) or using existing directory services.
4. The server's enrollment handler (described in "Enrollment (/enroll) URL Handler" (page 18)) determines whether the user is allowed to enroll a device. If the device is allowed to enroll, the server sends it a profile service payload (Listing 2-5 (page 20)).

   This profile service payload (`.mobileconfig`) contains a request for additional device-specific information that the device must provide in the next step.

   The payload may, at your option, include a `Challenge` token so that the server can associate the request with the original user. This allows you to customize the configuration process for each user, if desired. This value should be something that can be verified, but is not easily guessable. For example, you could store a random value into a database and associate it with the user's login credentials. The details of this functionality are site specific.

   The device attributes that the service can request are the iOS version, Wi-Fi device ID (MAC Address), product type (for example, iPhone 3GS returns `iPhone2,1`), phone equipment ID (IMEI), and SIM card identifier (ICCID).

   For a sample profile service payload, see "Sample Phase 1 Server Response" in *Enterprise Deployment Guide*.

Phase 2: Certificate Enrollment (X.509 Identities and SCEP)

In the second phase, enrollment, a device contacts the certificate authority and obtains a signed X.509 identity certificate, which is then used for encryption.

To acquire an identity, a device first generates an asymmetric key pair and stores it in its keychain. The secrets in this keychain can be read only by that specific device.

The device then sends its public key to a certificate authority (CA), which sends back a signed X.509 certificate. This certificate, coupled with the private key on the device, form an identity.

To make this exchange possible, iOS supports the simple certificate enrollment protocol (SCEP). SCEP is a communication protocol that provides a networked front end to a private certificate authority. Support for SCEP is provided by a number of certificate authorities, and there are complete open-source software implementations of certificate authorities with SCEP support.

The front end service can be set up to gate access by means of a challenge, which in practice is an authorization token (a one-time password, or a signed/encrypted blob containing user/device info) to allow automatic issuing of a certificate.

The steps in the enrollment phase are:

1. The user accepts the installation of the profile received in phase 1.
2. The device looks up the requested attributes, adds the challenge response (if provided), signs the response using the device's built-in identity (an Apple-issued certificate), and sends it back to the profile distribution service using HTTP POST.

   Note: If the device has been registered previously and is merely requesting a new configuration, it signs the request with the certificate previously provided by the CA in Phase 3.

   In the case of this example, the device sends this response to the `/profile` URL.

   For a sample profile from this phase, see "Sample Phase 2 Device Response" in *Enterprise Deployment Guide*.
3. The server's profile request handler (described in "Profile Request (/profile) URL Handler" (page 21)) sends back a configuration profile that instructs the device to enroll using SCEP as described in "Phase 2: Certificate Enrollment (X.509 Identities and SCEP)" (page 11). The server should sign this profile.

   For a sample configuration profile, see "Sample Phase 3 Server Response With SCEP Specifications" in *Enterprise Deployment Guide*.
4. The device enrolls using SCEP, resulting in a valid identity certificate installed on the device.

Phase 3: Device Configuration and Encrypted Profiles

The third phase of over-the-air profile delivery and configuration is the actual profile delivery itself. In this phase, the server sends a profile that has been customized for a particular device.

In some environments, it is important to make sure that corporate settings and policies are protected from prying eyes. To provide this protection, iOS allows you to encrypt profiles so that they can be read only by a single device.

An encrypted profile is just like a normal configuration profile except that the configuration profile payload is encrypted with the public key associated with the device's X.509 identity.

To keep adversaries from modifying the content, the encrypted configuration profile is signed by the service. For encryption and signing, iOS uses the Cryptographic Message Syntax (CMS), a standard that is also used in S/MIME. Payloads are encrypted using PKCS#7 enveloped data. Profiles are signed using PKCS#7 signed data.

The steps in the device configuration phase are:

1. The device sends a signed request for the `/profile` handler again to request the final profile. (The request is signed with the identity certificate obtained in the previous step.)

   For a sample profile for this phase, see "Sample Phase 4 Device Response" in *Enterprise Deployment Guide* .

2. The server's profile request handler (described in "Profile Request (/profile) URL Handler" (page 21)) sends the final encrypted profile to the device.

After Enrollment

Upon receiving the final encrypted profile, the device installs it. Reconfiguration occurs automatically if the profile expires or if a VPN connection attempt fails.

Settings enforced by a configuration profile cannot be changed on the device. To change these settings, you must install an updated profile.

Configuration profile updates are not pushed to users automatically. If you need to make other profile updates without waiting for the profiles to expire, you must distribute them manually or require users to re-enroll. (Devices also pull new profiles upon a VPN authentication failure.)

You can distribute updated profiles to your users in an email attachment or a webpage. To update a profile, the following conditions must be met:

- The profile identifier must match.

  For more information about the identifier, see "General Settings" in *Enterprise Deployment Guide* .

- If the profile was signed, the replacement profile must also be signed by the same issuer.

Depending on the General Settings payload you specified when creating the profile, it may be possible for the user to remove the profiles.

- If the profile requires a password for removal, the user is prompted for that password upon tapping Remove.
- If the profile specifies that it cannot be removed by the user, the Remove button does not appear.

Important: Removing a configuration profile removes all information associated with the profile. This includes policies, Exchange account data stored on the device, VPN settings, certificates, mail messages, and other information.

For more information about profile security settings, see "General Settings" in *Enterprise Deployment Guide* .

Creating a Profile Server for Over-The-Air Enrollment and Configuration

When creating a profile server, you must perform several steps:

1. Configure your infrastructure. This is described in "Configuring the Infrastructure" (page 14).
2. Obtain an SSL Certificate for your server. This is described in "Obtaining an SSL Certificate" (page 15).
3. Create a template configuration profile. This is described in "Creating A Template Configuration Profile" (page 15).
4. Create the server code. The pieces of a server are described in "Starting the Server" (page 16) and "Profile Service Handlers" (page 17).
5. Add appropriate authentication specific to your environment.
6. Test the service.

The sections that follow take you through the various parts of the profile delivery service source code.

Configuring the Infrastructure

Implementing over-the-air enrollment and configuration requires you to integrate authentication, directory, and certificate services. The process can be deployed using standard web services, but you must set up several key systems ahead of time.

Directory Services

For user authentication, you can use basic HTTP authentication or integrate authentication with your existing directory services. Regardless of the services used, you will need to provide a web-based authentication method for your users to request enrollment.

Certificate Services

The process of enrollment requires deployment of standard x.509 identity certificates to iOS users. To do this, you will need a CA (certificate authority) to issue the device credentials using the Simple Certificate Enrollment Protocol (SCEP).

Cisco IOS and Microsoft Server 2003 (with the add-on for certificate services) both support SCEP. There are also a number of hosted PKI services that support SCEP, such as Verisign, Entrust, and RSA. For links to PKI, SCEP, and related topics read the "See Also" (page 7) section in "Introduction" (page 5).

Profile Services

To implement this process you will need to develop a profile service, which is an HTTP-based daemon that manages iOS-based device connections throughout the process, generates configuration profiles for the user, and verifies user credentials along the way.

There are a few key functions that the profile service needs to provide:

- Host a user-accessible website to support the HTTPS session
- Authenticate incoming user requests using a web-based authentication method (basic, or integrated with directory services)
- Generate the necessary configuration profiles (XML format) depending on the phase of the process
- Sign and encrypt configuration profiles using public key cryptography
- Track the user through the steps in the process (via timestamp and logging methods)
- Manage connections to the certificate authority or directory services

Obtaining an SSL Certificate

The first step in setting up a profile service is to obtain or generate an SSL certificate for the web server. When hosting a profile server, each iOS-based device must be able to make a secure connection to the server. The easiest way to do this is to get an SSL certificate from a public CA that is already trusted by iOS. For a complete list, see iOS 3.0: List of Available Trusted Root Certificates.

Alternatively, you can generate your own root certificate and self-sign it, though if you do, the user will be asked whether they trust the certificate.

Creating A Template Configuration Profile

The profile service uses a template configuration profile as the starting point, then modifies the profile for a specific device. You must create this template ahead of time and save it to a file on disk. The iPhone Configuration Utility provides an easy means of creating such a base profile.

In addition to general settings, this configuration profile should also define enterprise policies that you want to enforce. For company-owned equipment, it should be a locked profile to prevent the user from removing it from the device.

For more information about these profiles, read "Configuration Profile Format" in *Enterprise Deployment Guide* .

Starting the Server

After you have an SSL certificate, you must configure a web server to host your profile service and an SCEP-aware certificate authority to issue certificates.

The initialization function, `init`, loads the HTTP server's certificate and private SSL key. These keys and certificates are stored on disk for reuse together with the serial number of the last issued certificate. This function is shown in Listing 2-1.

Listing 2-1 Starting the web server

```
world = WEBrick::HTTPServer.new(
  :Port            => 8443,
  :DocumentRoot    => Dir::pwd + "/htdocs",
  :SSLEnable       => true,
  :SSLVerifyClient => OpenSSL::SSL::VERIFY_NONE,
  :SSLCertificate  => @@ssl_cert,
  :SSLPrivateKey   => @@ssl_key
)
```

This example starts the server on port `8443` so it does not have to run as root. The `:DocumentRoot` value should contain the path of an empty directory inside the profile service directory.

You should enable SSL and set the values of `SSLCertificate` and `SSLPrivateKey` to point to your actual SSL certificate and key that you obtained in "Obtaining an SSL Certificate" (page 15).

You should also disable client certificate authentication because the client device does not have a verifiable identity yet.

Profile Service Handlers

After you have a basic web server, you need to write handlers for several pages used in the enrollment and delivery process.

Phase 1: Authentication

Welcome Page (/) URL Handler

The welcome page is the first page new users see when they enter the site at the root level (/). A handler for this page is shown in Listing 2-2.

Listing 2-2 Handler for / URL

```
world.mount_proc("/") { |req, res|
    res['Content-Type'] = "text/html"
    res.body = <<WELCOME_MESSAGE

<style>
body { margin:40px 40px;font-family:Helvetica;}
h1 { font-size:80px; }
p { font-size:60px; }
a { text-decoration:none; }
</style>

<h1 >ACME Inc. Profile Service</h1>

<p>If you had to accept the certificate accessing this page, you should
download the <a href="/CA">root certificate</a> and install it so it becomes
trusted.

<p>We are using a self-signed
certificate here, for production it should be issued by a known CA.

<p>After that, go ahead and <a href="/enroll">enroll</a>

WELCOME_MESSAGE
}
```

If you used a self-signed certificate above, when the user goes to this page, Safari asks whether you want to trust the server's SSL certificate. Agreeing allows you to view the page. This is not sufficient for enrollment, however.

Regardless of whether the site certificate is self-signed or not, the enrollment process with the SCEP service also requires the device to trust the custom certificate authority's root certificate, which means adding the CA root certificate to the device's trusted anchors list. To do this, you must create a URL handler that provides the certificate with the correct MIME type.

Root Certificate (/CA) URL Handler

The link to `/CA` in the welcome page provides a means for the user to add the custom certificate authority's root certificate to the device's trusted anchors list. This is required for the SCEP stage of the enrollment process.

After Safari on iOS loads the root certificate from that URL, it asks the user for permission to add the new root certificate to the device's trusted anchors list. (You should access this page only over a secure connection.)

The handler in Listing 2-3 sends the root certificate.

Listing 2-3 Handler for `/CA` URL

```
world.mount_proc("/CA") { |req, res|
    res['Content-Type'] = "application/x-x509-ca-cert"
    res.body = @@root_cert.to_der
}
```

After the user has downloaded the root certificate from a trusted web server over HTTPS, the user can click to continue the enrollment process.

Enrollment (/enroll) URL Handler

Listing 2-4 provides a handler for the `/enroll` link on the welcome page.

Listing 2-4 Handler for `/enroll` URL

```
world.mount_proc("/enroll") { |req, res|
    HTTPAuth.basic_auth(req, res, "realm") {|user, password|
        user == 'apple' && password == 'apple'
    }

    res['Content-Type'] = "application/x-apple-aspen-config"
    configuration = profile_service_payload(req, "signed-auth-token")
    signed_profile = OpenSSL::PKCS7.sign(@@ssl_cert, @@ssl_key,
            configuration, [], OpenSSL::PKCS7::BINARY)
    res.body = signed_profile.to_der

}
```

The handler above performs very limited authentication to identify the user. The user logs in by sending the word `apple` as the user name and password over a connection authenticated with HTTP basic authentication. In a production server environment, you should instead tie this code into a directory service or some other account system.

This handler sets the MIME type of its response to `application/x-apple-aspen-config`, so Safari on iOS treats the response as a configuration profile.

The `profile_service_payload` function ("Profile Service Payload" (page 19)) produces a special configuration that tells the phone to enroll itself in the profile service. The literal string `"signed-auth-token"` should be replaced with an authorization token from the authentication service that verified the user's credentials.

Finally, this function signs the profile by calling `OpenSSL::PKCS7.sign` and sends the signed profile to the device.

Security Note: This exchange occurs through HTTPS to protect the user name, password, and signed authorization token, so signing with the SSL certificate does not provide additional security. It does, however, make sense if the profile service uses a different SSL certificate and resides on a separate HTTPS server.

Profile Service Payload

The first payload sent to the device (after establishing that it is allowed to enroll) is the profile service payload. This payload is sent by a call to `profile_service_payload(req, "signed-auth-token")` from the `/enroll` handler (Listing 2-4).

For a sample profile service payload, see "Sample Phase 1 Server Response" in *Enterprise Deployment Guide*.

Listing 2-5 `profile_service_payload` function

```
def profile_service_payload(request, challenge)
    payload = general_payload()


    payload['PayloadType'] = "Profile Service" # do not modify
    payload['PayloadIdentifier'] = "com.acme.mobileconfig.profile-service"


    # strings that show up in UI, customisable
    payload['PayloadDisplayName'] = "ACME Profile Service"
    payload['PayloadDescription'] = "Install this profile to enroll for secure
access to ACME Inc."


    payload_content = Hash.new
    payload_content['URL'] = "https://" + service_address(request) + "/profile"
    payload_content['DeviceAttributes'] = [
        "UDID",
        "VERSION"
=begin
        "PRODUCT",              # e.g. iPhone1,1 or iPod2,1
        "SERIAL",               # The device's serial number
        "MEID",                 # The device's Mobile Equipment Identifier
        "IMEI"
=end
        ];
    if (challenge && !challenge.empty?)
        payload_content['Challenge'] = challenge
    end


    payload['PayloadContent'] = payload_content
    Plist::Emit.dump(payload)
end
```

This function starts by calling `general_payload`, which sets the version and organization (these values don’t change on a given server) and returns a template payload that provides a UUID for the profile.

The payload content provides a URL where the device should send its identification (using HTTP POST), along with a list of attributes that the server expects the device to provide (software version, IMEI, and so on).

If an authorization token (representing a user authentication) is passed in from the caller (shown in Listing 2-4 (page 18)), that token is added as the `Challenge` attribute.

In response, the device sends back the list of requested attributes along with their values. If the server sent a `Challenge` value in its request, the device also includes this value along with the requested device attributes. Finally, to prove it is an iOS-based device, the device signs this identification with its device certificate. This response is sent to the handler for the `/profile` URL.

> **Values Note:** The payload type must not be changed; the phone expects to see the literal string `"Profile Service"`.
>
> The identifier should be changed to an appropriate reverse-DNS-style identifier. The identifier should remain constant for any given profile service.
>
> The display name and description values are presented in the user interface to explain to the user what is about to happen.

Phase 2: Certificate Enrollment

Profile Request (/profile) URL Handler

The handler for the `/profile` URL is called twice—once to send the device authentication request before the device is allowed to enroll using SCEP, then again after the SCEP step to deliver the final profile to the device.

In this handler, the profile server receives a PKCS#7 signed data payload from the device, which it then unpack and verifies. For a sample of this profile, see "Sample Phase 2 Device Response" in *Enterprise Deployment Guide* .

To make it easier to follow, the `/profile` handler is divided into smaller pieces. The first piece of this handler is shown in Listing 2-6.

Listing 2-6 Handler for `/profile` URL, part 1 of 7

```
world.mount_proc("/profile") { |req, res|

    # verify CMS blob, but don't check signer certificate
    p7sign = OpenSSL::PKCS7::PKCS7.new(req.body)
    store = OpenSSL::X509::Store.new
    p7sign.verify(nil, store, nil, OpenSSL::PKCS7::NOVERIFY)
    signers = p7sign.signers
```

Security Note: The reference implementation does not verify the signer here. You should verify it against a trust store made up of intermediates from the device certificate authority up to the root CA and the hierarchy you'll use to issue profile service identities.

If the device signed the request with a certificate that belongs to the hierarchy that issues profile service identities (that is, if this device has enrolled previously), execution follows the first path (shown in Listing 2-7). This path either issues an updated encrypted configuration or, as implemented here, redirects the device to enroll again. For testing purposes, any device that has gotten a profile previously must reenroll.

Listing 2-7 Handler for `/profile` URL, part 2 of 7

```
    # this should be checking whether the signer is a cert we issued
    #
    if (signers[0].issuer.to_s == @@root_cert.subject.to_s)
        print "Request from cert with serial #{signers[0].serial}"
            " seen previously:
#{@@issued_first_profile.include?(signers[0].serial.to_s)}"
            " (profiles issued to #{@@issued_first_profile.to_a}) \n"
        if (@@issued_first_profile.include?(signers[0].serial.to_s))
          res.set_redirect(WEBrick::HTTPStatus::MovedPermanently, "/enroll")
            print res
```

By this point, any previously fully enrolled clients have been redirected to the enrollment page to enroll again.

If the code gets past this step, it has received either a list of properties or a new request for a final profile.

In Listing 2-8, the encrypted profile is generated. Because this is part of phase 3 (device configuration), it is included here without further comment, and is explained further in "The /profile Handler Revisited" (page 25).

Listing 2-8 Handler for `/profile` URL, part 3 of 7

```
        else
            @@issued_first_profile.add(signers[0].serial.to_s)
            payload = client_cert_configuration_payload(req)
                        # vpn_configuration_payload(req)
            #File.open("payload", "w") { |f| f.write payload }
            encrypted_profile = OpenSSL::PKCS7.encrypt(p7sign.certificates,
                payload, OpenSSL::Cipher::Cipher::new("des-ede3-cbc"),
                OpenSSL::PKCS7::BINARY)
            configuration = configuration_payload(req, encrypted_profile.to_der)
        end
```

The code in Listing 2-9 handles the case where the device sent its identification. This part should ideally verify that the response was signed with a valid device certificate and should parse the attributes.

Listing 2-9 Handler for `/profile` URL, part 4 of 7

```
    else
        #File.open("signeddata", "w") { |f| f.write p7sign.data }
        device_attributes = Plist::parse_xml(p7sign.data)
        #print device_attributes
```

The next bit of code, Listing 2-10, is commented out with `=begin` and `=end`. It shows how you can restrict issuance of profiles to a single device (by its unique device ID, or UDID) and verify that the `Challenge` is the same as the `Challenge` value issued previously.

In a production environment, this is typically replaced by site-specific code that queries a directory service to validate the authorization token and queries a database of authorized UDID values for devices owned by your organization.

Listing 2-10 Handler for `/profile` URL, part 5 of 7

```

=begin
        # Limit issuing of profiles to one device and validate challenge
       if device_attributes['UDID'] == "213cee5cd11778bee2cd1cea624bcc0ab813d235"
 &&
            device_attributes['CHALLENGE'] == "signed-auth-token"
        end
=end
```

Next, the snippet in Listing 2-11 obtains a payload to send to the device that will tell it how to complete the enrollment process. The details of this configuration are described in the discussion of `encryption_cert_payload`.

Listing 2-11 Handler for `/profile` URL, part 6 of 7

```
        configuration = encryption_cert_payload(req, "")
    end
```

Finally, if this function has nothing to send, it raises an exception that makes the http request fail. Otherwise it signs the profile to be sent and returns it. These bits of code are shown in Listing 2-12 (page 24).

Listing 2-12 Handler for `/profile` URL, part 7 of 7

```
    if !configuration || configuration.empty?
        raise "you lose"
    else
    # we're either sending a configuration to enroll the profile service cert
    # or a profile specifically for this device
    res['Content-Type'] = "application/x-apple-aspen-config"

        signed_profile = OpenSSL::PKCS7.sign(@@ssl_cert, @@ssl_key,
            configuration, [], OpenSSL::PKCS7::BINARY)
        res.body = signed_profile.to_der
        File.open("profile.der", "w") { |f| f.write signed_profile.to_der }
    end
}
```

After this function sends the configuration to tell the device how to enroll, the device enrolls its identity using SCEP. Then, it sends a request for the `/profile` URL associated with this handler a second time to obtain the final profile.

The actual payload is described in "Configuration Profile Payload" (page 26) and "Encryption Certificate Payload" (page 26). For a sample configuration profile, see "Sample Phase 3 Server Response With SCEP Specifications" in *Enterprise Deployment Guide* .

Phase 3: Device Configuration

The /profile Handler Revisited

Previously, Listing 2-8 (page 22) showed the encrypted profile generation process. The code in question doesn't actually run until phase 3, however, so the details were deferred. This section revisits that section of the `/profile` handler and provides explanation.

The encrypted profile is generated as follows:

- A configuration is generated with a set of configuration payloads. (See "Configuration Profile Format" in *Enterprise Deployment Guide* to learn about the contents of these payloads in detail.)

  In this reference implementation, every device gets the same profile. If desired, however, the `Challenge` information can be used to identify the user requesting the profile, and the code can generate a profile specific to that user.

  Similarly, the device information provided can be used to generate a profile specific to a given device or a particular type of device (for example, providing a different profile for different iOS-based device models).
- The configuration is encrypted with the public key of the device that signed the original request.
- The encrypted blob of data is wrapped in a configuration profile.

The details of this encrypted blob are explained in the descriptions of `client_cert_configuration_payload` (Listing A-1 (page 30)) and `configuration_payload` ("Configuration Profile Payload" (page 26)).

Listing 2-13 Handler for `/profile` URL, part 3 of 7 (revisited)

```
        else
            @@issued_first_profile.add(signers[0].serial.to_s)
            payload = client_cert_configuration_payload(req)
                        # vpn_configuration_payload(req)

            #File.open("payload", "w") { |f| f.write payload }
            encrypted_profile = OpenSSL::PKCS7.encrypt(p7sign.certificates,
                payload, OpenSSL::Cipher::Cipher::new("des-ede3-cbc"),
                OpenSSL::PKCS7::BINARY)
            configuration = configuration_payload(req, encrypted_profile.to_der)
        end
```

Configuration Profile Payload

The configuration profile payload (provided by `configuration_payload`) resembles the profile service payload described in "Profile Service Payload" (page 19). The only difference is in the payload its carries.

For a sample profile for this phase, see "Sample Phase 4 Device Response" in *Enterprise Deployment Guide* .

Encryption Certificate Payload

Listing 2-14 describes the encryption certificate payload. This payload tells the client how to complete the enrollment process.

Listing 2-14 `encryption_cert_payload` function

```
def encryption_cert_payload(request, challenge)
    payload = general_payload()

    payload['PayloadIdentifier'] = "com.acme.encrypted-profile-service"
    payload['PayloadType'] = "Configuration" # do not modify

    # strings that show up in UI, customisable
    payload['PayloadDisplayName'] = "Profile Service Enroll"
    payload['PayloadDescription'] = "Enrolls identity for the encrypted profile
service"

    payload['PayloadContent'] = [scep_cert_payload(request, "Profile Service",
challenge)];
    Plist::Emit.dump(payload)
end
```

The `scep_cert_payload` function is described in "SCEP Certificate Payload" (page 26).

SCEP Certificate Payload

As the name of the `scep_cert_payload` function suggests, the function shown in Listing 2-15 produces an SCEP payload that gives the device the information it needs to enroll a certificate.

Listing 2-15 `scep_cert_payload` function

```
def scep_cert_payload(request, purpose, challenge)
    payload = general_payload()

    payload['PayloadIdentifier'] = "com.acme.encryption-cert-request"
    payload['PayloadType'] = "com.apple.security.scep" # do not modify
```

The payload type of `com.apple.security.scep` indicates an SCEP payload and the content specifies the parameters.

```
    # strings that show up in UI, customisable
    payload['PayloadDisplayName'] = purpose
    payload['PayloadDescription'] = "Provides device encryption identity"

    payload_content = Hash.new
    payload_content['URL'] = "https://" + service_address(request) + "/scep"
```

First and foremost, there is the base URL for the SCEP service, which for convenience is handled by the sample service as well. It looks a little different for IOS (`http://scep-server/cgi-bin/pkiclient.exe`) and Windows SCEP servers (`http://scep-server/certsrv/mscep/mscep.dll`).

```
=begin
    # scep instance NOTE: required for MS SCEP servers
    payload_content['Name'] = ""
=end
```

The service can provide different certificate issuing services parameterized on the `Name` value that becomes part of the final URL. In the case of Windows, this value needs to be set, although any value will do.

```
    payload_content['Subject'] = [ [ [ "O", "ACME Inc." ] ],
        [ [ "CN", purpose + " (" + UUIDTools::UUID.random_create().to_s + ")" ] ]
 ];
    if (!challenge.empty?)
        payload_content['Challenge'] = challenge
    end
```

The subject allows the client to specify the requested subject. In this case, it is populated by the profile service. Some services may not want to grant the client the ability to specify it, and may use the `Challenge` to encode the identity of the requester.

X.509 subjects are elaborate structures and are mimicked here as an array of arrays, to fully specify it. Each key-value pair is specified as an array. The key is the first element and is a string with a value that is either an ID (for example, "0.9.2342.19200300.100.1.25" is DC) or one of the recognized abbreviations (CN, C, ST, L, O, OU). The example above represents a subject that will often be displayed as "/O=ACME Inc./CN={purpose} ({random UUID})".

```
    payload_content['Keysize'] = 1024
```

Next up are some, simple parameters, although they require some consideration. Key size requests the device to generate a keypair of a certain size. Only 1024-bit and 2048-bit key sizes should be used. Keys larger than 2048 bits are not supported. In general, 1024-bit keys are recommended because of the overhead involved in generating 2048-bit keys.

```
    payload_content['Key Type'] = "RSA"
```

The key type should always be RSA because this reference implementation (and in practice, SCEP) only support RSA keys.

```
    payload_content['Key Usage'] = 5 # digital signature (1) | key encipherment
(4)
```

Key usage specifies the purposes the key can be used for and is a bit mask. Bit 0 (value 1) specifies digital signature, and bit 2 specifies key encipherment. Note that the MS SCEP server will only issue signature or encryption, not both.

```
=begin

    payload_content['CAFingerprint'] =
StringIO.new(OpenSSL::Digest::SHA1.new(@@root_cert.to_der).digest)

=end
```

SCEP can run over HTTP, as long as the CA cert is verified out of band. This functionality is currently disabled (as shown above) because iOS does not currently support this. This function supports such operation by adding the fingerprint to the SCEP payload that the phone downloads over HTTPS during enrollment, as shown below:

```
    payload['PayloadContent'] = payload_content;
    payload
end
```

```
            payload = client_cert_configuration_payload(req)
                        # vpn_configuration_payload(req)
```

Configuration Profile Examples

This appendix includes two sections. The first, "Configuration Profile Payload Code Example" (page 30), shows how to construct a basic profile payload programmatically. The second, "Sample Responses" (page 31), shows examples of property lists that might be exchanged during a typical SCEP enrollment session.

Configuration Profile Payload Code Example

This example of a configuration profile payload shown in Listing A-1 contains a webclip that points the user to an intranet site and provides a payload to allow the phone to enroll an SSL client authentication cert that will be required to access the protected assets.

Listing A-1 `client_cert_configuration_payload` function

```
def client_cert_configuration_payload(request)


    webclip_payload = general_payload()


    webclip_payload['PayloadIdentifier'] = "com.acme.webclip.intranet"

    webclip_payload['PayloadType'] = "com.apple.webClip.managed" # do not modify


    # strings that show up in UI, customisable

    webclip_payload['PayloadDisplayName'] = "ACME Inc."

    webclip_payload['PayloadDescription'] = "Creates a link to the ACME intranet
on the home screen"


    # allow user to remove webclip

    webclip_payload['IsRemovable'] = true


    # the link

    webclip_payload['Label'] = "ACME Inc."

    webclip_payload['URL'] = "https://" + service_address(request).split(":")[0]
# + ":4443/"
```

The webclip creates an icon that will take the user to the URL mentioned. In this case we allow the user to delete the webclip.

```
    client_cert_payload = scep_cert_payload(request, "Client Authentication",
"foo");
```

The client certificate is enrolled by creating an SCEP payload similar to the one used for decrypting an encrypted payload. In a real-world implementation, you typically add additional parameters to specify key usage, policies, and subject alternative names to make it easier for the server to match the enrolled identity with a particular user and that user's capabilities.

```
    Plist::Emit.dump([webclip_payload, client_cert_payload])


end
```

This function ends by dumping the raw array of payloads. The caller wraps them in a configuration profile and signs them, as shown in Listing 2-8 (page 22).

For more information about the types of payloads that are available, see the *iOS Enterprise Deployment Guide* at http://manuals.info.apple.com/en_US/Enterprise_Deployment_Guide.pdf.

Sample Responses

This section includes sample profiles that illustrate over-the-air enrollment and configuration phases. These are excerpts and your requirements will vary from the examples. For syntax assistance, see the details provided earlier in this appendix. For a description of each phase, see *Over-the-Air Profile Delivery and Configuration* .

Sample Phase 1 Server Response

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple Inc//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

    <dict>

        <key>PayloadContent</key>

        <dict>

            <key>URL</key>
            <string>https://profileserver.example.com/iphone</string>
            <key>DeviceAttributes</key>
            <array>
                <string>UDID</string>
                <string>IMEI</string>
                <string>ICCID</string>
                <string>VERSION</string>
                <string>PRODUCT</string>
            </array>
            <key>Challenge</key>

            <string>optional challenge</string>


or


            <data>base64-encoded</data>

        </dict>
        <key>PayloadOrganization</key>
        <string>Example Inc.</string>
        <key>PayloadDisplayName</key>
        <string>Profile Service</string>
        <key>PayloadVersion</key>
        <integer>1</integer>
        <key>PayloadUUID</key>
        <string>fdb376e5-b5bb-4d8c-829e-e90865f990c9</string>
        <key>PayloadIdentifier</key>
        <string>com.example.mobileconfig.profile-service</string>
        <key>PayloadDescription</key>
       <string>Enter device into the Example Inc encrypted profile service</string>
        <key>PayloadType</key>
        <string>Profile Service</string>
    </dict>
</plist>
```

Sample Phase 2 Device Response

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

    <dict>

        <key>UDID</key>

        <string></string>

        <key>VERSION</key>

        <string>7A182</string>

        <key>MAC_ADDRESS_EN0</key>

        <string>00:00:00:00:00:00</string>

        <key>CHALLENGE</key>


either:


        <string>String</string>


or:


        <data>"base64 encoded data"</data>


    </dict>

</plist>
```

Sample Phase 3 Server Response With SCEP Specifications

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple Inc//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

    <dict>

        <key>PayloadVersion</key>

        <integer>1</integer>

        <key>PayloadUUID</key>
        <string>Ignored</string>
        <key>PayloadType</key>
        <string>Configuration</string>
        <key>PayloadIdentifier</key>
        <string>Ignored</string>
        <key>PayloadContent</key>
        <array>
            <dict>
                <key>PayloadContent</key>
                <dict>
                    <key>URL</key>
                    <string>https://scep.example.com/scep</string>
                    <key>Name</key>
                    <string>EnrollmentCAInstance</string>
                    <key>Subject</key>
                    <array>
                        <array>
                            <array>
                                <string>O</string>
                                <string>Example, Inc.</string>
                            </array>
                        </array>
                        <array>
                            <array>
                                <string>CN</string>
                                <string>User Device Cert</string>
                            </array>
                        </array>
                    </array>
                    <key>Challenge</key>
                    <string>...</string>
                    <key>Keysize</key>
                    <integer>1024</integer>
                    <key>Key Type</key>
                    <string>RSA</string>
                    <key>Key Usage</key>
                    <integer>5</integer>
                </dict>
                <key>PayloadDescription</key>
                <string>Provides device encryption identity</string>
                <key>PayloadUUID</key>
                <string>fd8a6b9e-0fed-406f-9571-8ec98722b713</string>
                <key>PayloadType</key>
                <string>com.apple.security.scep</string>
                <key>PayloadDisplayName</key>
                <string>Encryption Identity</string>
                <key>PayloadVersion</key>
                <integer>1</integer>
                <key>PayloadOrganization</key>
                <string>Example, Inc.</string>
                <key>PayloadIdentifier</key>
                <string>com.example.profileservice.scep</string>
            </dict>
        </array>
    </dict>
</plist>
```

Sample Phase 4 Device Response

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
        <key>UDID</key>
        <string></string>
        <key>VERSION</key>
        <string>7A182</string>
        <key>MAC_ADDRESS_EN0</key>
        <string>00:00:00:00:00:00</string>
    </dict>
</plist>
```

Document Revision History

This table describes the changes to *Over-the-Air Profile Delivery and Configuration* .

Date	Notes
2010-08-03	Folded in samples from Enterprise Deployment Guide and made iOS name changes throughout.
2010-06-04	Added complete script as companion file.
2010-03-24	New document that describes how to build a server that generates profiles on the fly and delivers them to iPhone devices over the air.