



Adobe AIR Data Privacy and Security

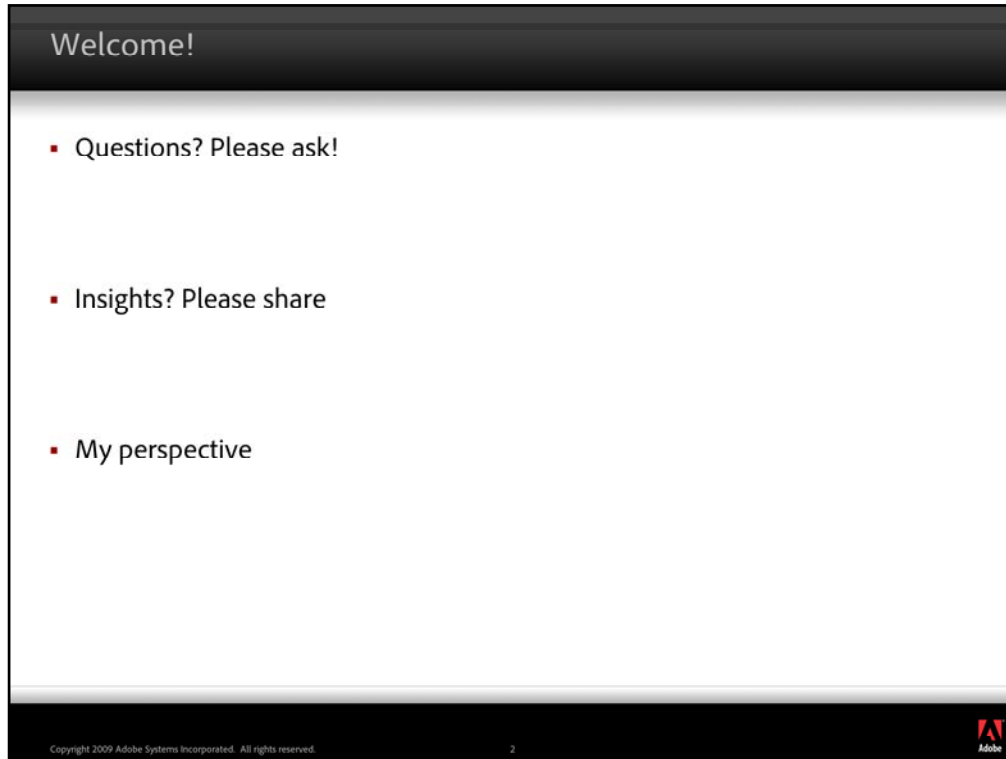
H. Paul Robertson
Sr. ActionScript Developer/Writer
Adobe Systems, Inc.

360Flex Indianapolis
May 20, 2009



Copyright 2009 Adobe Systems Incorporated. All rights reserved.

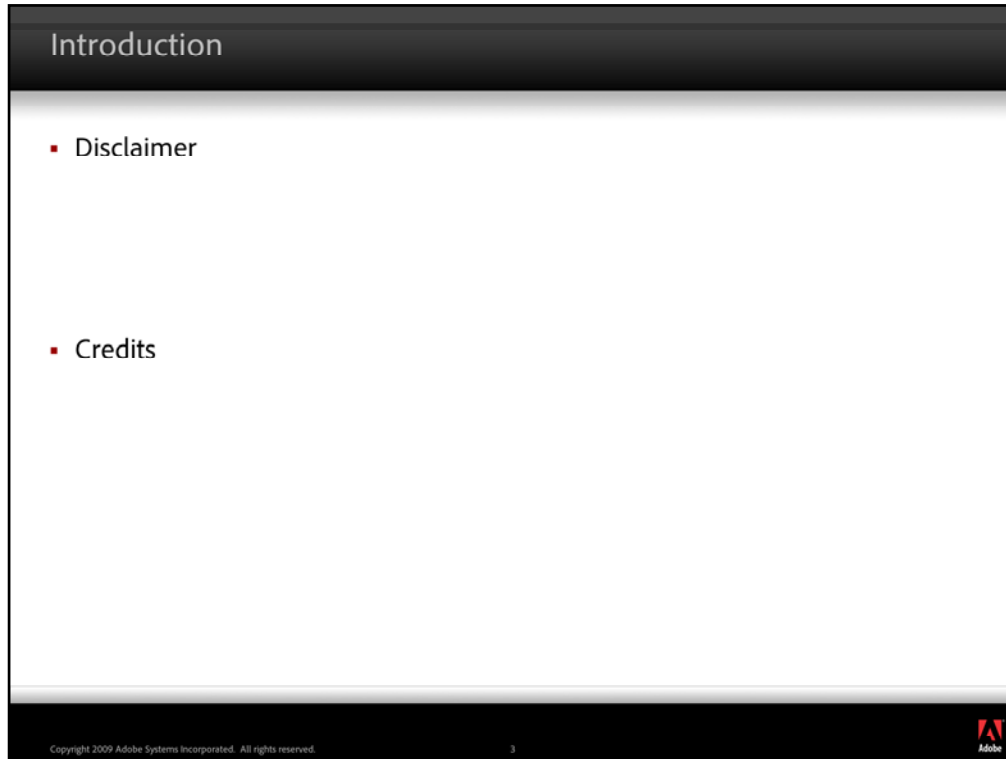
1



My background:

I worked for several years as a web application developer (server side as well as UI). I studied the types of security techniques that are important for that domain. However, once I started working in AIR I learned that there is another set of potential concerns, and another set of solutions to learn.

In particular, I was working on the documentation for the encrypted database functionality that was added in AIR 1.5 and I found myself questioning a lot of my previous assumptions about how to keep data secure and even what it means for an application to be "secure." That led to conversations with engineers from the AIR and Adobe secure software teams...which led me to this presentation, as a way to compile the things I had learned and share them with others.



Since I work on the AIR documentation team, I know that my colleagues and I don't intentionally "keep secrets." As much as time permits, we try to put the best information we can in the documentation.

Because of this, most of what I'm going to tell you is available in the documentation (or will be the next time we release an update).

In addition, one resource that was particularly valuable to me in preparing this presentation is Ethan Malasky and Peleus Uhley's MAX 2008 session "Maintaining security with Adobe AIR."

Background

Copyright 2009 Adobe Systems Incorporated. All rights reserved. Adobe confidential.

4



Background

What will (and won't) be covered

- AIR-specific functionality
- Security problems and solutions

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

5



This presentation is intended to cover AIR-specific functionality and privacy/security issues. There are some aspects of security that are important in AIR, and are also important in browser-based Flex applications. For example, in order to keep network communication secure, you should encrypt communication using SSL/TLS (i.e. <https://> urls). This is true in AIR, but it's also true in Flex and in any browser-based application, so in the interest of time I won't cover that type of information here.

The format I've chosen for the presentation is a "problem-and-solution" format. We'll go through a sequence of problems that potentially expose data privacy and security risks, then talk about the solutions that are available in Adobe AIR to work around or defend against those risks.



As general background, and by way of comparison, I want to talk briefly about the Flash Player security model. Flash Player has a security model based on the idea of security sandboxes. (This model is common among web browsers and browser plug-ins.)

The idea of browser security sandboxes is that any executable code that comes from a network source runs within its own security sandbox. The code can't access information about the user's computer. It also can't access code or data from remote sources other than its own sandbox.

(Examples: accessing local files; full-screen mode restrictions (user-triggered and esc))

For example, as of Flash Player 10 you can build an application that allows a user to select a file from the machine's hard drive and your application can access the contents of that file. However, your application can't arbitrarily select and open a file – the user must explicitly choose the file to open every time you open a file. Once the file is open, you don't have access to any information about the file other than its original file name – you can't learn where it came from, or anything about the file structure of the user's machine.

Opening a file is one of several operations that are available in Flash Player but can only be triggered by a user action – that is, the code that performs one of these operations must always be triggered by some sort of explicit user action such as clicking a mouse button or pressing a key on the keyboard.

Another example of Flash Player security restriction is Flash Player's full screen mode. You can have your app open full screen, but the action of going to full screen must be triggered by a user action. In addition, for several seconds while going to full-screen mode, a note appears indicating to the user that pressing the <esc> key allows them to exit full-screen mode. Finally, most keyboard input isn't allowed in full-screen mode.


These restrictions are very important. When you click on a link or type in a url you don't know what you're going to come across. It should not be possible for code from a web site to cause damage to your computer or access private data without your explicit consent.

Background

Adobe AIR design philosophy

- *Don't* avoid features simply because of "security risk"
- *Do* design features to default to safe behaviors
(Source: Oliver Goldman, AIR Lead Engineer)

- Exception: When AIR functions like a browser (e.g. SWF-in-HTML)

Copyright 2009 Adobe Systems Incorporated. All rights reserved.7

AIR design philosophy:

AIR apps, on the other hand, are different from web pages in a browser. In order to run an AIR app, you must explicitly opt in by choosing to install the application. Consequently, AIR apps are able to perform lots of actions that aren't available to Flash Player, browsers, etc. For example, any AIR app can delete every file from your hard drive (at least, every file that you have access to). (Note that this is no different than any other executable application that a user can install on his/her computer.)

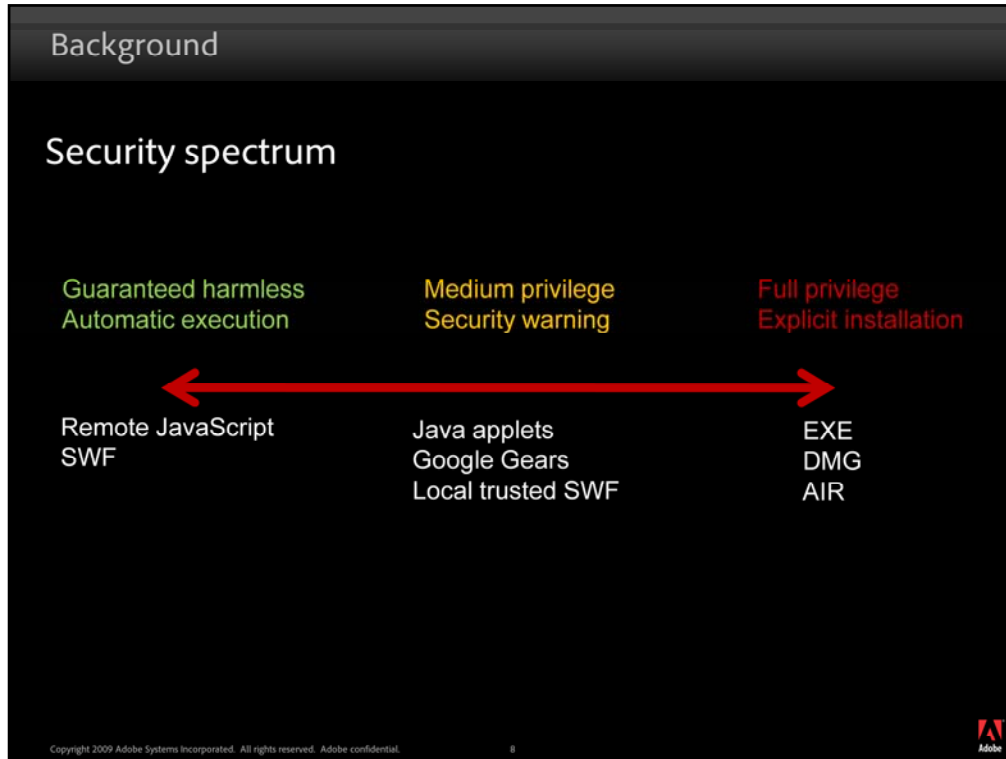
The AIR runtime isn't intentionally designed to be "insecure." However, end-user security is not generally a reason that particular functionality is excluded from AIR. As Oliver Goldman, lead engineer for Adobe AIR states on his blog:

"AIR applications are desktop applications and can already do dangerous things. It doesn't make sense for us to limit new features that aren't any more dangerous. We *do* design features to *default* to safe behaviors, but we don't reject features just because they might be dangerous."

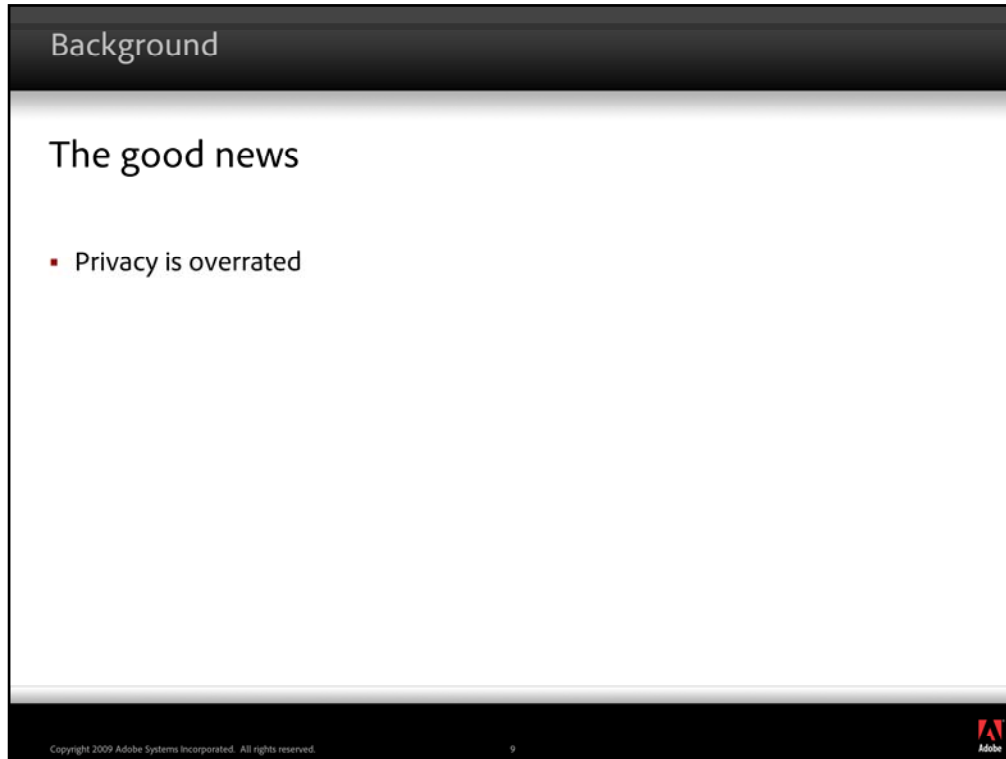
There are exceptions to the rule that content executing in AIR has full system access. A general way of describing the exceptions is that when AIR is functioning like a browser, it behaves like a browser – the content is sandboxed like browser content.

(examples: SWF-in-HTML; remote or non-application content)

I'll discuss this in more detail later in the presentation.



Here is a visual representation of the “security spectrum” – the levels of access that executable content have, and the user opt-in requirements corresponding to that level of access.



"Privacy is overrated"

That's probably overstating things a bit =)

However, it's definitely true that not every application needs the utmost level of privacy and security for the content it creates/manipulates/stores.

For example, many users create documents in MS Word or Excel. Normally these documents aren't encrypted in any way. The Word and Excel file formats are well-known and it's very easy to find tutorials online that teach how to write code that reads those file formats. (In fact, Microsoft gives away an SDK that makes it very simple for developers to create applications that read those files, not to mention giving them lots of other control over MS Office apps.)

But that's perfectly okay, because most of the documents created in Word and Excel don't need to be encrypted – they don't contain any private or secret information that couldn't be shared with others.

Similarly, Adobe Photoshop can open a number of image format files, but doesn't encrypt them in any way. A number of other applications can also open Photoshop's native file format (.psd), either because the developers paid to license the format from Adobe, or simply reverse-engineered the file format. In fact, for some time (I'm not sure if this is still true now) digital camera manufacturers did not document or license their proprietary file formats. Nevertheless, Adobe software was able to open the files. How? Adobe engineers simply reverse-engineered the proprietary file formats every time a new camera came out. Again, the camera manufacturers didn't help Adobe, but they didn't use any encryption to hide the image data either, because they knew that for most use cases the images don't need to be kept "secret."

Whenever you're designing an application, you need to carefully consider the goals and uses of your application in deciding how much and what kind of data privacy you need to provide.

Background

What is "private" data?

(Who are you keeping the data private from?)

- Anyone other than the single end user who owns the data
- People other than your application's users
- What about administrators? ("disgruntled IT employee")
- The end users themselves

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

10



An important step in deciding what level of privacy your application needs is figuring out who should, and more importantly who shouldn't, have access to the data. The answer to those questions defines what "data privacy and security" actually means for your application.

For a more detailed description of some of these scenarios, see the link titled "Considerations for using encryption with a database" (that section of the documentation is about encrypted databases, but the principles can be applied to general data privacy concerns.)

The first bullet describes what is perhaps the most restrictive scenario – where a user is storing private data and nobody else – even others who can access the machine, or others who have access to your application, should be able to access the data.

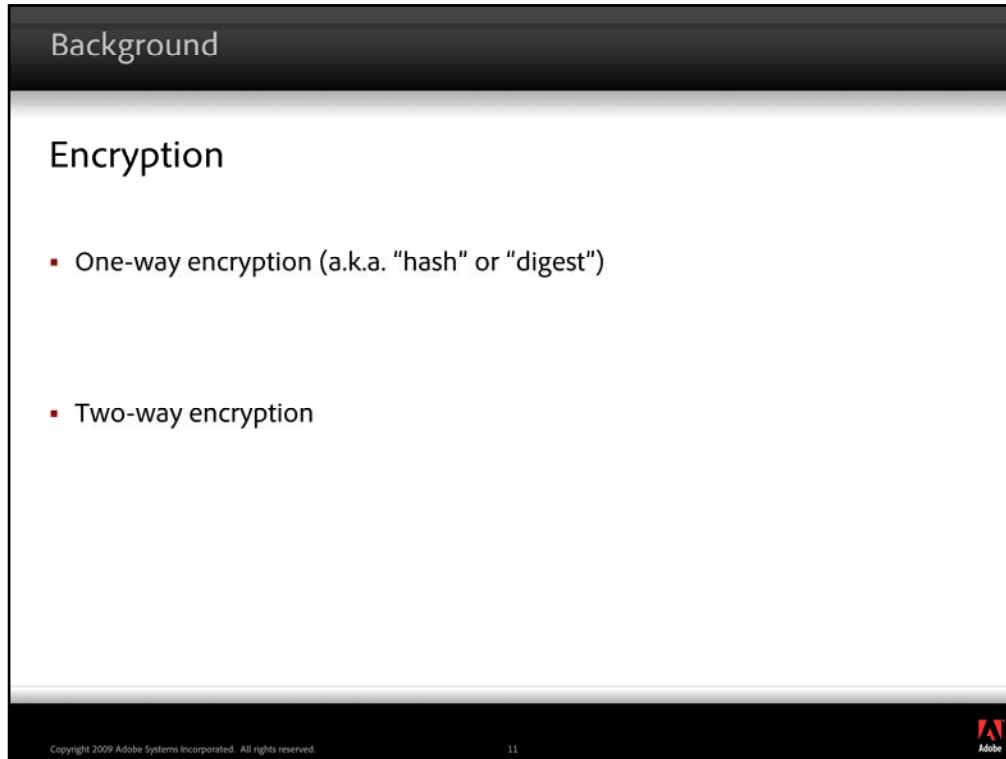
Another scenario is where it's okay for anyone who has a copy of your app to view the data, but you want to prevent anyone else from accessing it.

One set of users to keep in mind is the hypothetical "disgruntled IT employee" – someone who has "administrator" access to the machine so they can access all users' files and data, and can even impersonate other users' accounts and pretend to be that user.

One category that I didn't think much about at first was keeping data private from the end users themselves. After all, if they own the app, they own the data, right? Not always...

(example: Video rental app, where users might want to circumvent data security measures in order to be able to access content outside of the app's restrictions.)

This category is where AIR's security measures are weakest. In all other cases the end user is motivated to support the goal of keeping data private, or at least has no interest either way. But in this case, the user has a motivation to break the security of the application and may actively use attack techniques to do so. At this point, the security techniques available in AIR are not suitable for protecting against this kind of attack (at least, not by itself). You can use AIR in conjunction with FMRMS in order to implement a DRM solution.

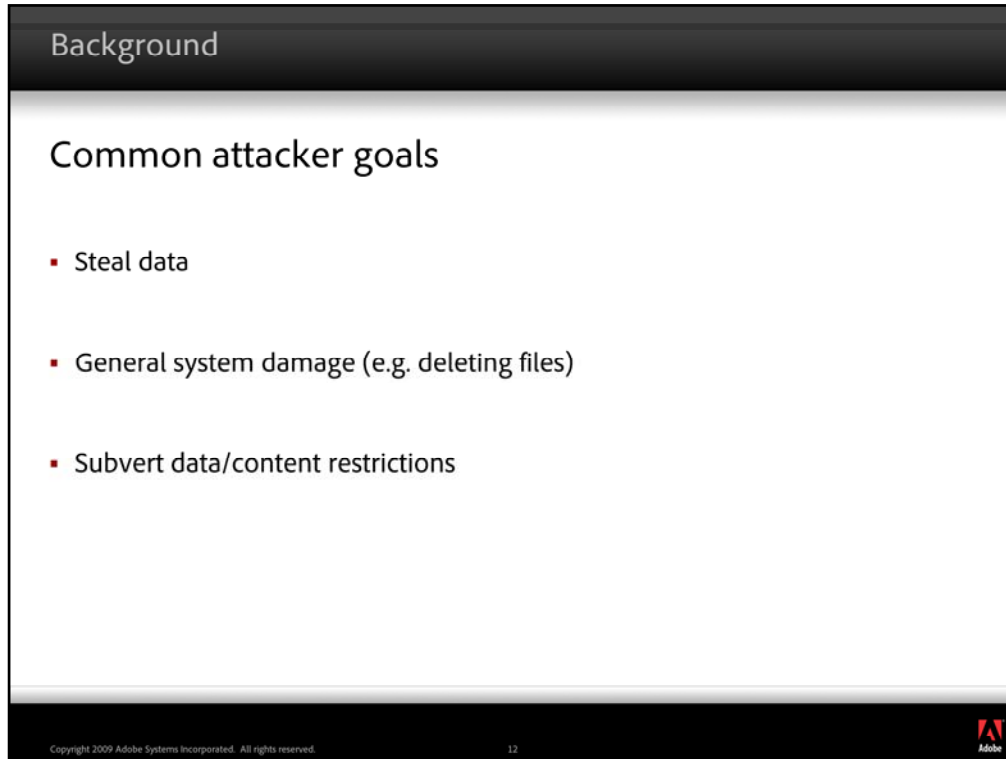


By way of background, I just wanted to cover encryption and specifically these two types since it's a topic that will be used throughout the presentation.

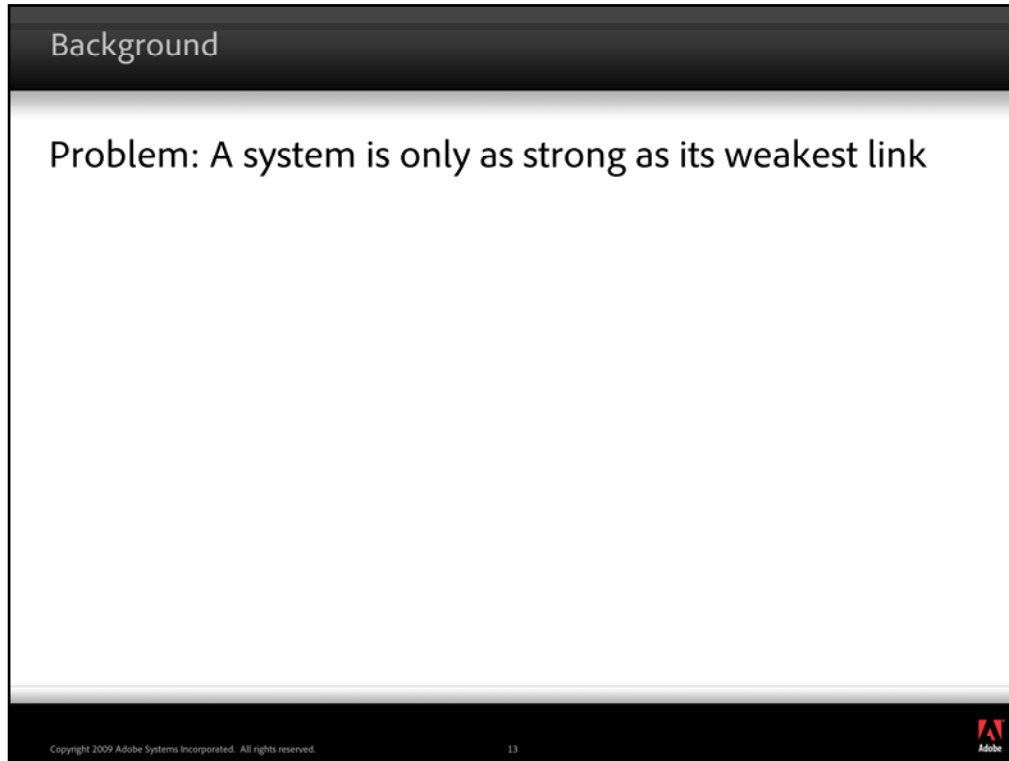
One-way encryption is a technique where you use an algorithm to create a mangled version of content. These algorithms are usually designed so that very similar inputs actually result in significantly different output. There is no corresponding algorithm that allows you to un-mangle the content (i.e. retrieve the original content from the encrypted version). Because of this, one-way encryption is only useful for validating data. For example, if you have an application where a user enters a password to log in, you can use one-way encryption to "hash" the user's password before storing it. That way you aren't actually storing the user's password in plain text, so even if someone was able to access the stored password, they couldn't figure out what it is. When the user attempts to log in, you simply hash the password they enter and compare it to the stored hash. If the two hashes match you know they entered the password correctly.

Two-way encryption mangles the content but also provides a way to retrieve the original content. Generally the algorithms that are used for two-way encryption require you to specify an encryption key that is essentially a password for the encrypted content – in order to decrypt the content, you pass the same encryption key to the decrypting algorithm, and it uses the key to unscramble your content.

In some cases the encrypting and decrypting keys are identical; in others there are two keys known as a public key and a private key, and the pair of keys work together – content encrypted with one key can be decrypted using the other key, and vice-versa.



This may seem obvious, but I just wanted to get these out in the open so we know what we're trying to prevent.



It's pretty much a cliché but it deserves to be said.

Background

Solution: Use thick chains

- Examine every part of your security solution in isolation and together
- Security is everyone's problem

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

14



For example, if you are using a user-entered password to generate an encryption key, consider imposing minimum length and complexity restrictions on passwords. A short password that uses only basic characters can be guessed quickly.

Background

Problem: Modern operating systems are multi-user

- Can't assume only one user per machine
(e.g. student computer labs)

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

15



As I alluded to earlier, when you're designing for security you have to always keep in mind that the operating system will let more than one user log in to a machine.

Unless you know for certain that your users are the only people with access to their machines, you may need to assume that others can access the machine as "guest" or even perhaps as "administrator" and access other users' data.

Background

Solution: Built-in OS security

- Access controls (e.g. file access) provided by the operating system
- Store private data in a user-specific location
(e.g. documents directory, application storage directory)

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

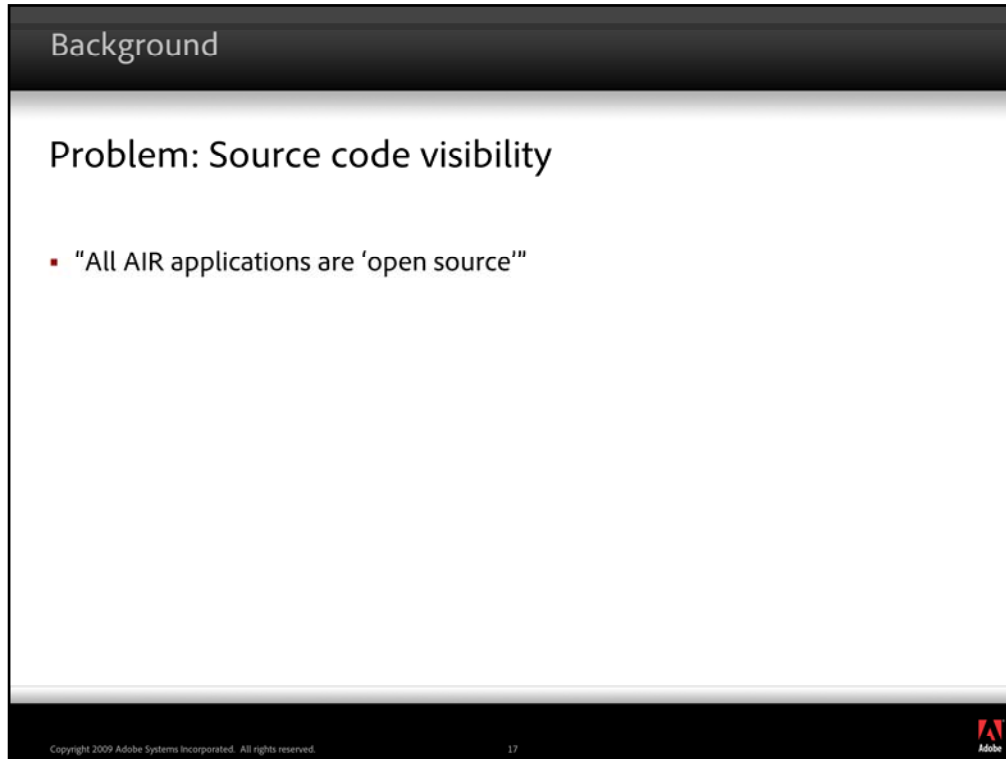
16



An AIR application executes with the access permissions and privileges of the user who runs the app. If the user has permission to read data from a certain folder, the AIR app can read data from that folder. Likewise, if an AIR app writes private data into a location where only the user has access (such as their “documents” or “application storage” directories) then other users who don’t have access to those areas won’t be able to access that data.

However, this protection is fairly limited. It protects against other users who don’t have access to the files, but doesn’t protect against “admin” users and doesn’t protect against malicious apps that are running with the user’s permissions.

The folder names for application storage directories look random at first glance. Practically speaking, however, the application storage directory is not obfuscated in any meaningful way. Other AIR apps, or any other app, could enumerate folder and file names easily enough.



A .air file is just a zip file – you can change the extension to .zip and open it using any ZIP file extractor.

Once the AIR application is installed, the unzipped source code is placed in the application directory. For HTML/JS applications, this means the HTML and JavaScript files. For a Flex- or Flash-based AIR app, this means the SWF file, which can be easily decompiled.

Demos: view HTML app source; decompile a SWF

Background

Conclusion: No "security through obscurity"

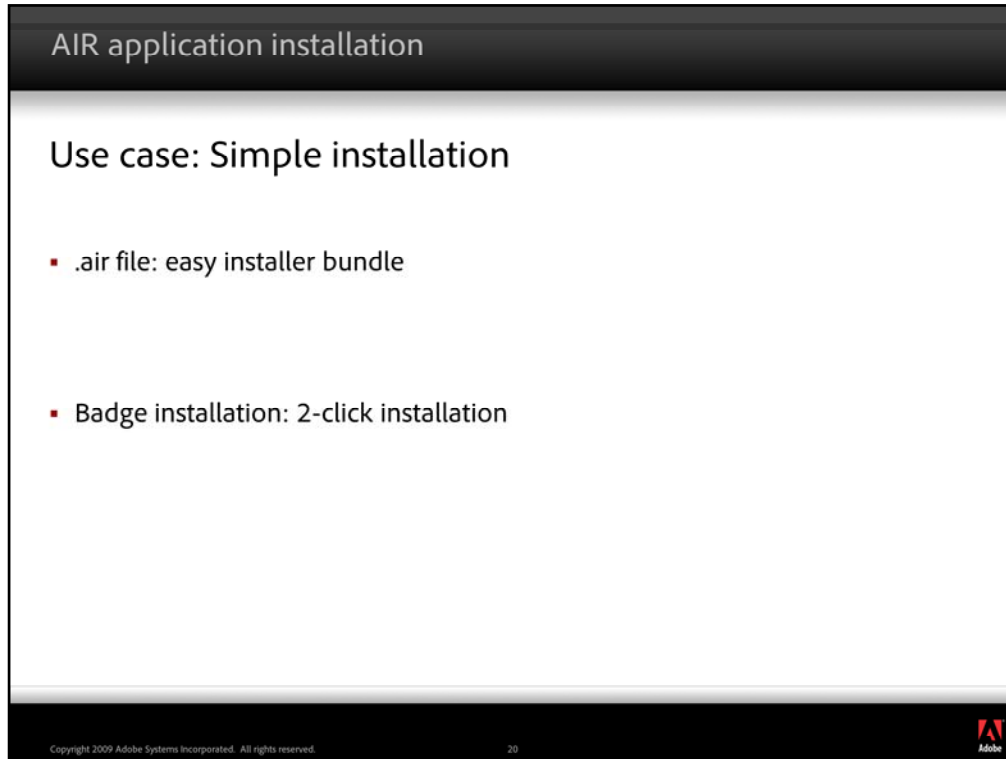
- Assume attackers have access to your source code
 - They know how you're encrypting data
 - Don't hard-code secrets (password, encryption key) in source code
-
- Nitro-LM by Simplified Logic claims to protect source code

AIR application installation

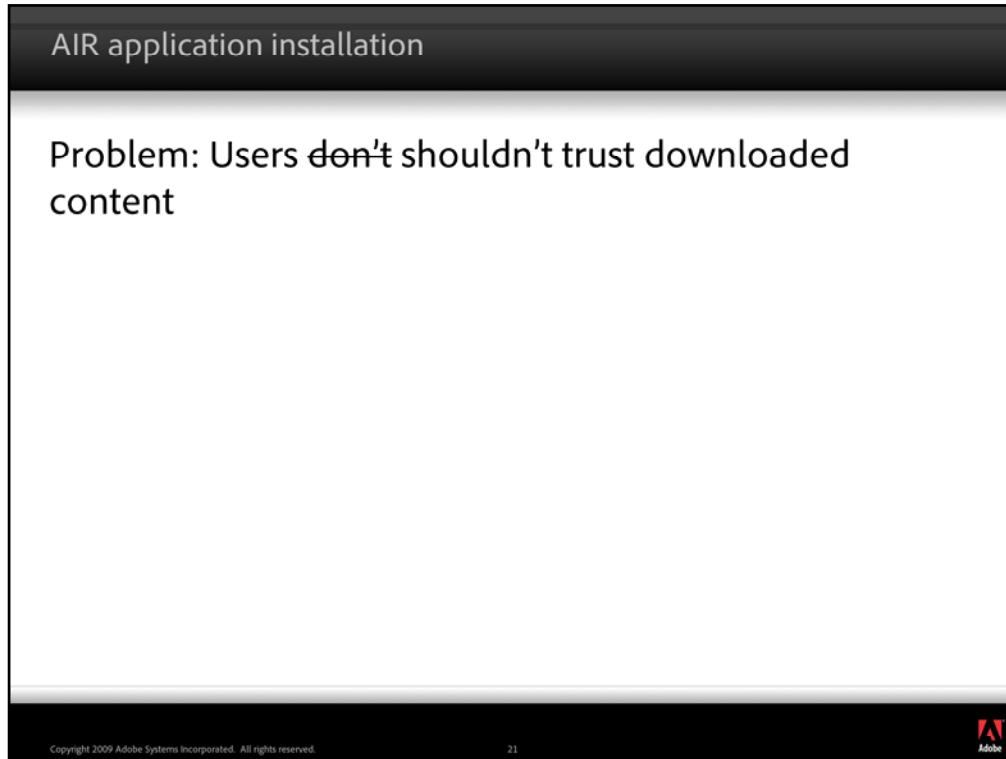
Copyright 2009 Adobe Systems Incorporated. All rights reserved. Adobe confidential.

19





One key part of the Adobe AIR “experience” is making it straightforward to download and install applications. AIR applications come as .air files, which can install on any supported operating system (developers don’t need to create OS-specific installers). Using the “badge installation” technique, a user can install your application in as little as two clicks.



The flip side to this simplicity is the question of user trust.

When a user downloads an AIR application from the Internet, the user has to make a decision as to whether to install the application. This is essentially a decision of trust – first of all do they trust that the .air file actually comes from who they think it comes from, and second do they in fact trust that person or company to not do anything bad to them.

AIR application installation

Solution: Signed applications

- All .air files must be signed
- AIR validates file contents using signature

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

22



To assist the user in making a trust decision, the AIR runtime requires all AIR apps to be signed with a code-signing certificate. This potentially provides two forms of reassurance to the user.

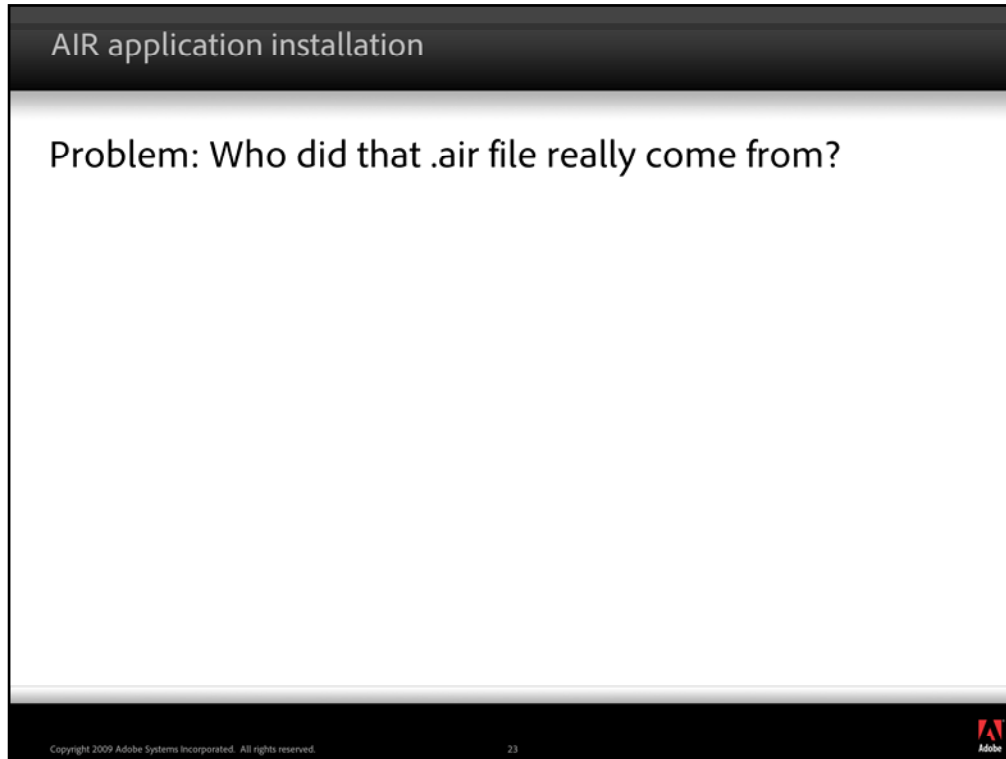
When downloading any file including an AIR application, you are at risk from attacks such as a “man in the middle” attack where an attacker computer intercepts communication between your machine and the server, potentially altering the server’s response before it gets to you without you realizing that you’re not communicating directly with the server.

How does AIR attempt to protect users from this? First, when a .air file is created, part of the package includes a digest (hash) of the file’s contents. That hash is created using the code-signing certificate. The public key of the certificate is included in the .air file bundle. While installing an AIR application, the AIR runtime validates the contents of the .air file by re-creating the hash using the certificate key that’s embedded in the .air file. If the digest matches, AIR can guarantee to the end user that the bits haven’t been altered since the application was signed. That way, the user can trust that the application hasn’t been modified by some third party in transit.

An important related note that should be pointed out is that AIR only validates the code at installation time, not each time the application runs. In other words, if after the application is installed someone is able to replace the SWF file with another one, when the user runs that app it will execute the replacement SWF file and not yours. Likewise, if you include supporting files in the .air bundle such as images, XML configuration files, etc., anyone who has access to the installation directory is able to modify or replace those files and alter your application’s behavior. If the application files have been tampered with after the app is installed, AIR won’t prevent the application from running.

Why doesn’t AIR validate the installed files every time?

- 1) Performance reasons – it takes time to hash and validate those files, and while that’s acceptable for the installation process it’s not acceptable for each time the application runs.
- 2) This is actually an issue for most applications – once it’s installed you can alter the installed files and there isn’t much that can be done to stop you.
- 3) In order to actually alter the files, a user usually needs permission to modify files in the general application installation location for their operating system (e.g. the “Program Files” directory on Windows, or “/Applications” on OS X). Standard user accounts don’t have permission to access these areas, so a user must be an administrator on the machine to make these types of changes. (However, a user can also choose to install an AIR app in other locations, so in that case the app is subject to the restrictions of the install location.)
- 4) Even if a user does have permission to access the install location, they would need to actually do something to make the changes. In other words, they would need to manually modify files/overlay files in the install location, or they would need to execute some application that makes the changes on their behalf (which could be intentional or could be by spoofing the user or by a virus). So the user ultimately would need to cooperate (intentionally or unwittingly) in order to accomplish this type of attack. This means that in most cases the user would need to have a reason to tamper with the app. For an app that stores private data about the user, the user has no motivation to tamper with the app – in fact they have good reason to “not” tamper with it.



The problem is, just guaranteeing that an app hasn't been modified doesn't really give much benefit by itself. An attacker could just as easily take an app, modify its contents, and re-sign it with another certificate. In that case, the app would still pass the validation step.


AIR application installation

Solution: Use a trusted (e.g. chained) certificate

- Chain of trust relationships
- Only guarantees identity – not trustworthiness

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

24



In order to protect users, they need to know not only that the content wasn't modified, but also who the content was actually signed by.

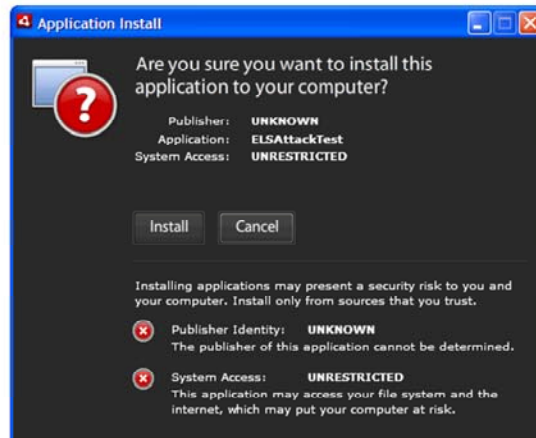
You can create your own code-signing certificate, in which case AIR can't vouch for you. You can also buy one from a certificate authority (CA). CAs verify the identity of a company or individual before issuing a certificate. The certificates they issue are in turn signed by their own certificate. If your computer recognizes a certificate authority (by having their certificate already in your computer's trust store), it trusts that CA and it trusts anybody that they trust.

Put another way, it's like the user's computer trusts Verisign, and they trust Thawte, and they trust another authority, who says you really are who you say you are. This is reflected in the installation dialog for the AIR app.

Note, however, that by issuing a certificate to a company, a CA isn't making any claims about whether they can actually be trusted – only that they really are who they say they are. So you know that my app really comes from Paul Robertson. At that point you as a user have to decide whether you trust me to not write something that will mess up your computer or steal your data.

AIR application installation

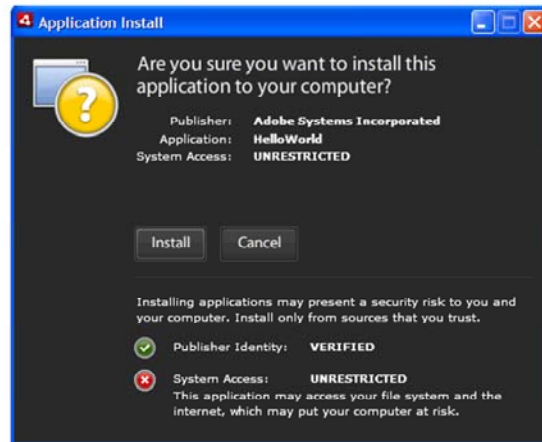
Solution: Use a trusted (e.g. chained) certificate



This is what you see when AIR can't establish a chain from an app's certificate to a trusted certificate.

AIR application installation

Solution: Use a trusted (e.g. chained) certificate



On the other hand, if a chain of trust is established, the bottom section changes to indicate that the publisher identity is “verified” and the publisher’s name appears in the top.

And no, there isn’t any way to change the red X that says “system access: UNRESTRICTED” or to change the yellow “caution” question mark to something else.

AIR application installation

Problem: Code-signing certificates are expensive!

- \$300/year for Versign and Thawte
- \$200/year for lesser-known CAs

AIR application installation

Solution: Free* code signing certificates

- Submit your application to the Adobe AIR Marketplace

*One year certificate. While supplies last (hurry, hurry, hurry!)

AIR application installation

Problem: Programmers aren't perfect

- Believe it or not, your code has bugs. Maybe even security holes (gasp!)
- Unlike web applications, you can't just update the server

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

29



Your code has bugs.

When you find a security issue, you don't automatically have a way of contacting the users of your application to notify them that they need to install the next version – you just have to make it available and hope they find out.

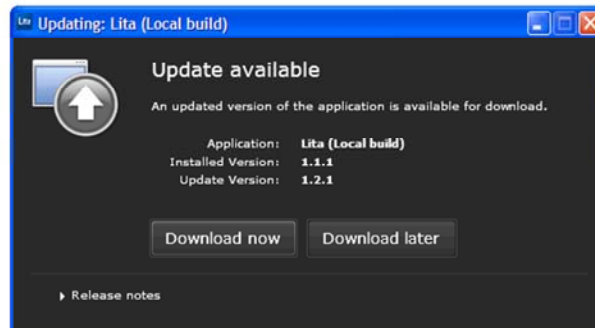
Solution: Plan for updates

- Assume your application will be a big hit
- Assume you will find security problems after release

Application installation

Solution: Plan for updates (cont.)

- Update frameworks make update functionality easy
air.update.* classes



AIR includes an easy and useful update framework, as well as a few more flexible approaches.

For example, the simplest version already includes all the dialogs you might need, so you don't even have to create the UI. You can configure it to make updates optional or not, to let users choose when to update (or not), etc.

If you add the few lines of code it takes to support updates in your app, then your users will pull down updates as soon as you make them available.

If you don't, then when you discover a bug or security issue, you'll have to rely on users manually discovering, downloading and installing the update.

Application installation

Solution: Plan for updates (cont.)

```
var appUpdater:ApplicationUpdaterUI = new ApplicationUpdaterUI();
appUpdater.configurationFile = new File("app:/config/update.xml");
appUpdater.addEventListener(ErrorEvent.ERROR, onError);
appUpdater.initialize();
```

```
<?xml version="1.0" encoding="utf-8"?>
<update xmlns="http://ns.adobe.com/air/framework/update/description/1.0">
  <version>2.0</version>
  <url>http://localhost/UpdateSampleFlex/UpdateSampleFlex2.air</url>
  <description><![CDATA[
Version 2.0. This new version includes:
  * Feature 1
  * Feature 2
  * Feature 3
]]></description>
</update>
```

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

32



This is (most of) the code it takes to add update functionality to your application.

The first code block is the ActionScript code you use to initialize the update framework. In this case, you would also include an XML file (not shown) in your app's source that defines which screens you want to show and other options. (You can set those options in code as well if you'd rather not have them in an XML file.)

The second block is another XML file that you post on your server. To push a new version, you just upload the .air file and the new version of this XML file.

Modular applications

Copyright 2009 Adobe Systems Incorporated. All rights reserved. Adobe confidential.


33



Modular applications

Use case: Make your application modular/extensible

- Download first-party modules
- Support third-party "plug-ins"

Copyright 2009 Adobe Systems Incorporated. All rights reserved.34

While it's not for everyone, some applications want to incorporate external code. This commonly takes a couple of forms:

- The app is architected as "modules" that can be downloaded when necessary and executed in the app
- Third party module or plug-in architecture allows other authors to create functionality that extends your app

Since AIR is built on web technologies, it's pretty easy to load external code, user interfaces, etc., for example using runtime shared libraries or Flex modules. This is a common pattern in browser-based Flex and Flash applications.

Modular applications

Problems: Same as application trust problems

- Attackers can modify code or replace it entirely before it reaches your application

Copyright 2009 Adobe Systems Incorporated. All rights reserved. 35

However, any time you're downloading and executing code, you're subject to the same risks that were described earlier for AIR applications in general – you need a way to a) know who the code is actually coming from, and b) validate that the code wasn't altered in transit.

To compound the problem, the user doesn't get to choose whether to install and run the code – that choice is entirely up to your application – even though the potential risks are identical (data theft or loss).

Requesting a module from a particular server isn't a guarantee of trust.

(Man-in-the-middle and DNS attacks again)

There are several parts to the solution for securely exposing this type of functionality.

Modular applications

Solution (built in): Security sandboxes

- Application code (in application directory): Application security sandbox
 - Full access to AIR APIs
- Loaded code: non-Application security sandbox
 - No access to AIR APIs (by default)

Copyright 2009 Adobe Systems Incorporated. All rights reserved. 36 Adobe

The first part of the solution is built in to AIR.

Somewhat akin to Flash Player, AIR uses a concept of security sandboxes in determining what privileges executing code has.

Code that is in the “application directory” (the directory into which AIR installed your application) runs in what’s known as the Application security sandbox. This code has full access to all AIR APIs.

Remember that it is possible that the code in the application directory has been modified since the app was installed (see slide 22 for details.)

All other code (specifically code that runs from other locations such as other folders on a users hard drive) runs in a non-Application security sandbox and doesn’t have any access to AIR APIs.

Solution (built in): Security sandboxes (cont.)

- These techniques import content into application sandbox
 - `Loader.loadBytes()`
 - `HTML.htmlText`
 - `HTML <script>` tag

There are a few techniques that allow you to “import” code rather than execute it directly. In theory these techniques could be used to allow you (or an attacker) to circumvent the security sandbox restrictions. If you were to use one of these techniques and an attacker managed to get their code in place of yours, their code would run in the application sandbox.

However, to prevent this risk, all “imported” code runs in a non-Application security sandbox, and doesn’t have access to AIR APIs. (It runs as though it is SWF content in a browser.)

Modular applications

Exception: When you're *absolutely certain* code is safe

- SWF content
 - `LoaderContext.allowLoadBytesCodeExecution`
- HTML content
 - `HTMLLoader.placeLoadStringContentInApplicationSandbox` property
- Code runs in application sandbox
- Not to be used lightly!
Treat all external code as third-party code

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

38




There are some cases where you really do want to import code and run it in the application sandbox.

These override controls are only to be used when you are absolutely positively 150% sure that the content is safe – that it came from a trusted source (and you can prove it!) and that it hasn't been tampered with.

Modular applications

Solution: Use signed modules

- Validates that code hasn't been altered
- Confirms identity of signer
- Use XMLSignatureValidator API

Copyright 2009 Adobe Systems Incorporated. All rights reserved.39

Another part of the solution is to require loaded code to be signed. Just as AIR apps are signed to provide two levels of trust, you can sign modules to get the same trust benefits.

The XMLSignatureValidator API uses the same logic and techniques that are used for validating an AIR application. You can sign your module code with the same code-signing certificate you use to sign an AIR app, and then you can check the signature within your app to determine if you trust the module.

(Note that this still doesn't automatically give the module application sandbox privileges – it just gives you a means to decide whether you trust the module code.)

Modular applications

Solution: expose APIs

- AIR's sandbox bridge lets you define APIs for inter-sandbox communication

Code in loader:

```
var child:Loader = new Loader();  
child.contentLoaderInfo.parentSandboxBridge = new MySandboxBridge();  
child.load(request);
```

Code in module:

```
var sandboxBridge:Object = loaderInfo.parentSandboxBridge;  
var value:String = sandboxBridge.someMethod();  
sandboxBridge.someProperty = 12;
```

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

40



The final solution that's available in AIR is the “sandbox bridge,” which allows you to expose APIs to code running in a different security sandbox. Using the sandbox bridge, code running in different security sandboxes can communicate. Both sides must opt in. All data passed between sandboxes is passed by value so the non-Application sandbox code isn't operating on instances in the Application stack.

The top code listing shows how you expose a sandbox bridge in your app. The `parentSandboxBridge` property is defined as an `Object` instance. In this case an instance of a custom class, `MySandboxBridge`, is actually assigned to the property so there is a defined API.

In the second listing, the module accesses the sandbox bridge object and calls a method and sets a property value.

Modular applications

Solution: expose APIs

- Be careful not to expose too much functionality

```
var bridge:Object = new Object();  
bridge.file = new File();  
child.contentLoaderInfo.parentSandboxBridge = bridge;
```

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

41



Remember, however, that any objects you pass to the external code can be used as if in the Application security sandbox. Before exposing any AIR-specific data types, think carefully about why those data types are AIR only (and expose the functionality in some other way instead).

For example, if you want to allow modules to store preferences, instead of giving them a File object to allow them to write preferences to a file, just expose a `setPreference()` method that lets them specify preference name and value pairs.

Local shared objects

Copyright 2009 Adobe Systems Incorporated. All rights reserved. Adobe confidential.

42



Local shared objects

Use case: Store values across sessions

- Use same syntax/code as in Flash Player

Problem: Local shared objects aren't encrypted

- Data serialized as AMF
- Anything that understands AMF can read it

Solution: Don't use local shared objects

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

43



This technique for storing data is straightforward to use, especially if you have an application that is already written for the browser or works in both browser and desktop.

However, it doesn't use any kind of encryption at all, so for any data that you want to keep private you shouldn't store it in LSOs.

Encrypted local store

Copyright 2009 Adobe Systems Incorporated. All rights reserved. Adobe confidential.

44



Encrypted local store

Use case: Store values across sessions (securely)

- Per user, per app publisher, stored separately per application
- No need to worry about encryption keys

```
var myData:ByteArray = new ByteArray();  
myData.writeUTF("my value");  
EncryptedLocalStore.setItem("key", myData);  
  
EncryptedLocalStore.getItem("key");
```

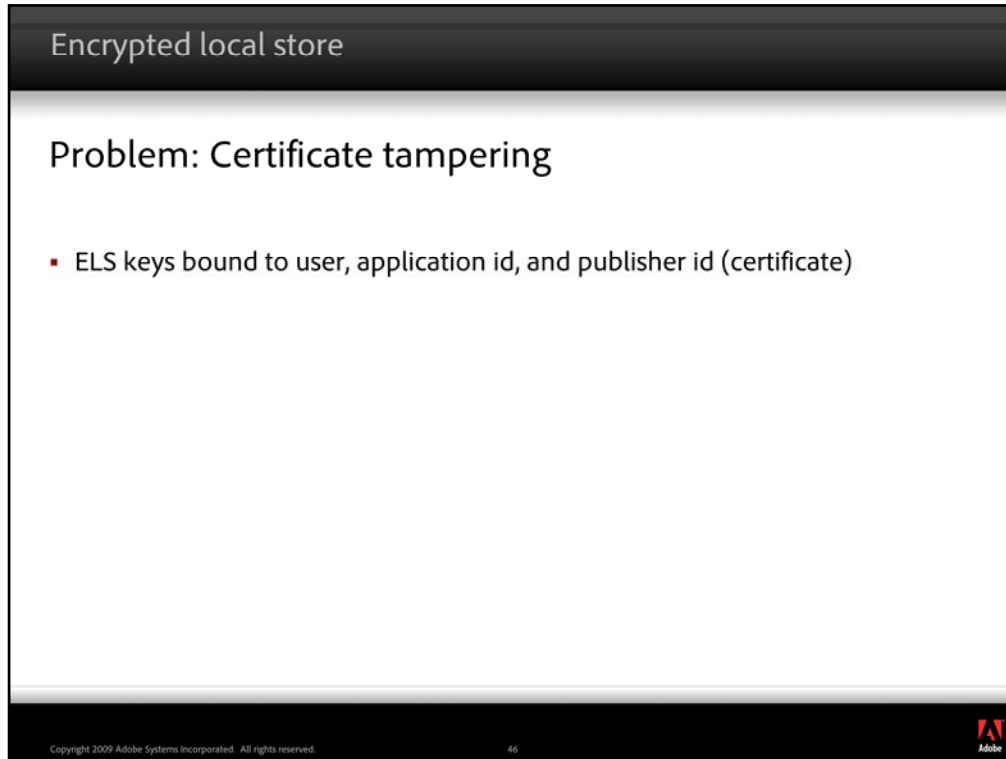
Copyright 2009 Adobe Systems Incorporated. All rights reserved. 45



The Encrypted Local Store is designed to fill the need for an encrypted alternative to LSOs.

It provides a two-way encryption mechanism designed for storing small secrets, such as remote credentials. It has a few benefits, including the fact that the values are encrypted per user and per application (so multiple users on the machine can't access each others' data even if they run the same app), and you don't have to worry about encryption key creation or storage at all because it's handled by AIR.

One limitation of working with the ELS is that you store values as ByteArray objects, so there is some complexity in reading and writing data. In addition, ELS is not designed for storing large amounts of data and it will have performance problems as the size of data grows large. Really it's just designed for small, simple values.



One potential area of weakness in the ELS is the problem described on slide 22, that an application's executable code can be modified after the app is installed. The publisher certificate (and publisher id that is derived from it) are part of what is used for encrypting ELS data. If someone modifies an app by replacing the installed version with a copy of the app signed with a different certificate, then any ELS data created by the app after the certificate is replaced is encrypted based on the attacker certificate rather than the original developer's certificate.

In that case the attacker still can't access ELS data stored by the app before the certificate was replaced, but any data written by the application afterward can be accessed. So the success of this type of attack depends on the certificate being replaced before the desired data is written to the ELS. To defend against this type of attack, store ELS data as soon as possible after the app is installed – which generally means the first time the app runs. (You don't necessarily have to store all data – it is sufficient for example to store an encryption key in the ELS on the first run. Then you can use that encryption key to encrypt other data later.)

Also note that there are additional steps involved in actually accessing the data, including copying the actual ELS data from the folder where it's stored to the ELS storage location of another attacker app.

As mentioned in the notes to slide 22, this type of attack would require user cooperation (or else require the user to be tricked e.g. by a virus) in order to succeed. Adding in the fact that the timing of the attack is critical, it's highly unlikely that this type of attack can succeed unless the user is a willing participant in the attack. So for storing user data, or encryption keys used to protect user data, the ELS is a valuable tool. However, for a DRM or similar scenario where you are protecting data from the user, it is potentially vulnerable.

Encrypted local store

Solution: Strong binding (?)

- Set `stronglyBound` parameter to `true`

```
EncryptedLocalStore.setItem("key", myData, true);
```

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

47



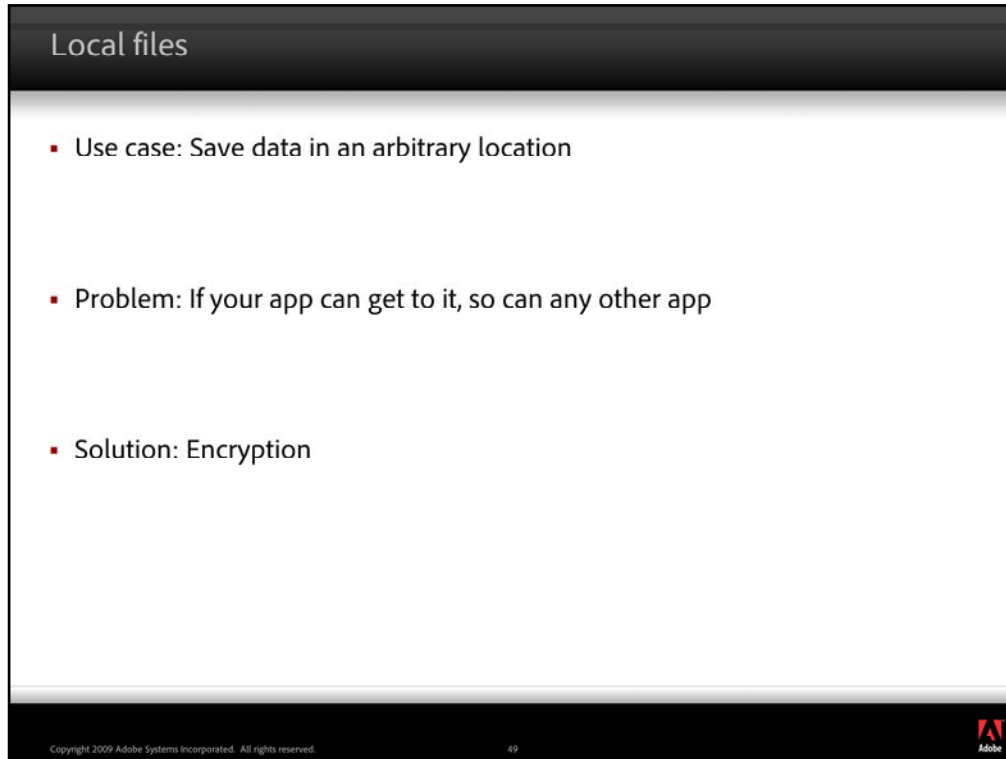
In order to make the ELS even stronger, you can set a third parameter to `true` when you're setting an ELS value. By using strong binding, the value is encrypted based on the user, publisher id, *and* additionally based on a hash of the files in the application directory. In that case, the value can't be retrieved except by an app with exactly the same source. This protects the data from certificate replacement, but it also means that when you update your app you won't be able to access the value either.

Local files

Copyright 2009 Adobe Systems Incorporated. All rights reserved. Adobe confidential.

48





This is all on one slide because it's pretty straightforward and obvious.

AIR lets you save data in files. You access the data as raw bytes, so you can structure it in any way you want. However, any other app can also read the files. They need to be able to reverse-engineer your data format in order to actually read the files, but it can be done.

The only solution is to encrypt your data (e.g. using two-way encryption) before writing it to the file, then decrypt it when you read it back.

Local SQL database (SQLite)

Copyright 2009 Adobe Systems Incorporated. All rights reserved. Adobe confidential.

50



Local SQL database (SQLite)

Use case: Store and query structured data locally

- Uses a language (SQL) and techniques that are familiar to web developers

```
var conn:SQLConnection = new SQLConnection();
conn.open(databaseFile);

var statement:SQLStatement = new SQLStatement();
statement.text = "SELECT * FROM myTable WHERE firstName = 'joe'";
statement.execute();
```

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

51




The local SQL database functionality makes it easy to store structured, relational data for your application, in a way that many web developers are familiar with.

As the code example shows, you create and open a `SQLConnection` instance to connect to a database, then you use a `SQLStatement` object to execute SQL commands against that database.

Local SQL database (SQLite)

Problem: SQL injection attack

```
statement.text = "SELECT * FROM myTable " +  
                "WHERE firstName = '" + input.text + "'";
```

Copyright 2009 Adobe Systems Incorporated. All rights reserved. 52 

One potential area of attack against an application is a SQL injection attack. This is an attack that is well-known for web applications as well so web database developers are probably familiar with it.

The weakness that allows this type of attack is when you concatenate user input into an SQL command, as in this code example where the text from an text field is added into the statement.

Local SQL database (SQLite)

Solution: Use statement parameters

```
statement.text = "SELECT * FROM myTable " +  
                "WHERE firstName = :firstName";  
statement.parameters[":firstName"] = input.text;
```

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

53




This type of attack is easy to prevent by using statement parameters instead of concatenation. In this example the statement defines a parameter named “:firstName”, and the parameter value is set separately. AIR doesn’t just use concatenation to plug in the parameter values. The binding of the parameter value into the statement happens within the database engine, so there is no risk of SQL injection.

Local SQL database (SQLite)

Problem: An AIR app can read another AIR app's database

- For that matter, any application that supports SQLite can read the databases

Copyright 2009 Adobe Systems Incorporated. All rights reserved. 54



The fact that AIR's database functionality is easy to use also makes it an easy target. Every application uses the same database format. Since AIR uses SQLite under the hood, any application that understands SQLite can read an AIR application's database. In fact, AIR includes functionality that lets you introspect a database to discover the table structure etc.

This is good because it means that developers can (and have) created useful tools that make it easier to create and work with databases in AIR. However, it means that if you're storing your app data in a local database, that data isn't private.

Local SQL database (SQLite)

Solution: Encrypted databases (added in AIR 1.5)

- You provide a 16-byte encryption key when creating the database

```
var encryptionKey:ByteArray = new ByteArray();  
encryptionKey.writeUTFBytes("abcdefghijklmnop");  
  
var conn:SQLConnection = new SQLConnection();  
conn.open(databaseFile, SQLMode.CREATE, false, 1024, encryptionKey);
```

- To re-open the database, provide the same encryption key

Copyright 2009 Adobe Systems Incorporated. All rights reserved.

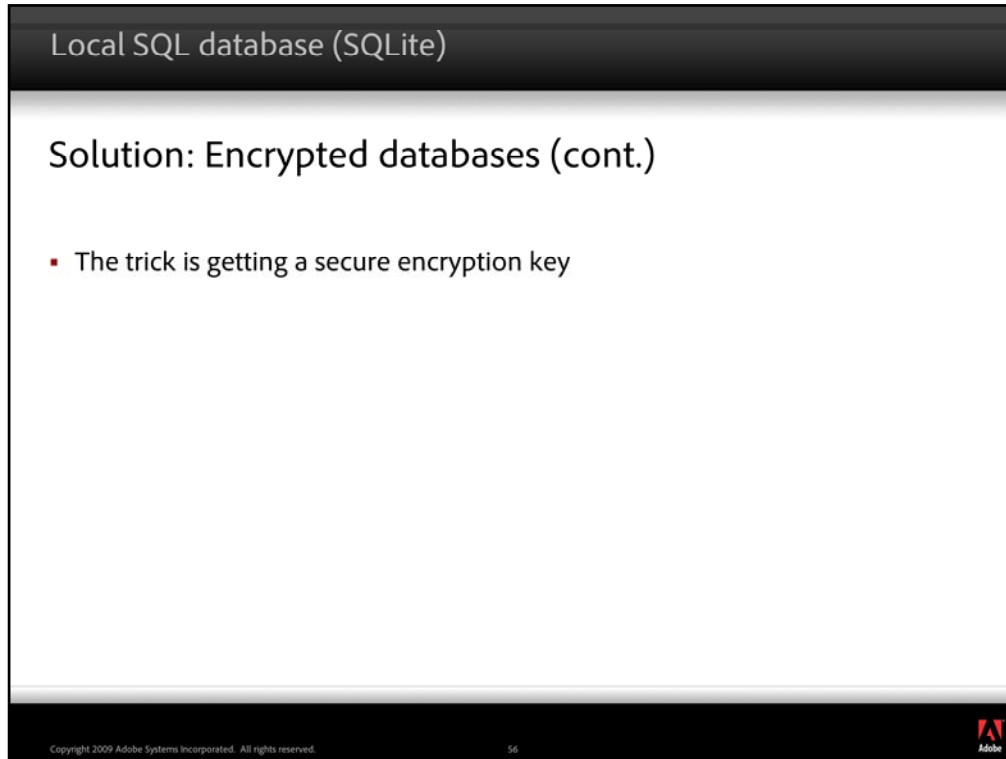
55



To protect database data, you can use an encrypted database.

When you create a database by opening a connection, you create it as an encrypted database by providing a 16-byte encryption key. Later, in order to re-open the database you must use the same encryption key or you won't be able to access the data.

(You can also change an encrypted database's encryption key by calling the `reencrypt()` method after connecting to it.)



Since you have to provide the encryption key, the biggest complexity with using an encrypted database is coming up with an encryption key in a way that provides sufficient security.

(See the documentation for details on the issues around this.)

Solution: Encrypted databases (cont.)

- Most secure technique for generating an encryption key:
 1. Get a strong password from the user (and validate it)
 2. Expand the password to 256 bits
 3. Get a salt value
 - a. First time: Randomly generate salt and store in ELS
 - b. Subsequent times: retrieve from ELS
 4. XOR password and salt
 5. Hash the value
 6. Extract 16 bytes (128 bits) from the hashed value
 7. Repeat (use this technique to derive the key from the password when re-opening the db)

There are a range of techniques for getting an encryption key, and different use cases for why you might favor one over the other.

While I was working on the encrypted database feature in AIR 1.5, the security engineers I worked with came up with the following list of steps to follow, which make up the most secure way to generate an encryption key. Why only focus on the most secure technique? Because it's easy to make an application less secure, and harder to make it more secure. On top of that, for any desired level of privacy you can always use the most secure technique to store the data and then share the data in other ways.

Solution: Encrypted databases (cont.)

- Most secure technique for generating an encryption key:

1. Get a strong password from the user (and validate it)
2. Expand the password to 256 bits
3. Get a salt value
 - a. First time: Randomly generate salt and store in ELS
 - b. Subsequent times: retrieve from ELS
4. XOR password and salt
5. Hash the value
6. Extract 16 bytes (128 bits) from the hashed value
7. Repeat (use this technique to derive the key from the password when re-opening the db)

There's a lot of complexity in these steps, so I won't take the time to describe them all. And if you want to use this technique, you don't have to implement them all either.

Solution: Encrypted databases (cont.)

- Most secure technique for generating an encryption key:

1. Get the EncryptionKeyGenerator class (part of as3corelib)
2. Use it as shown in the documentation

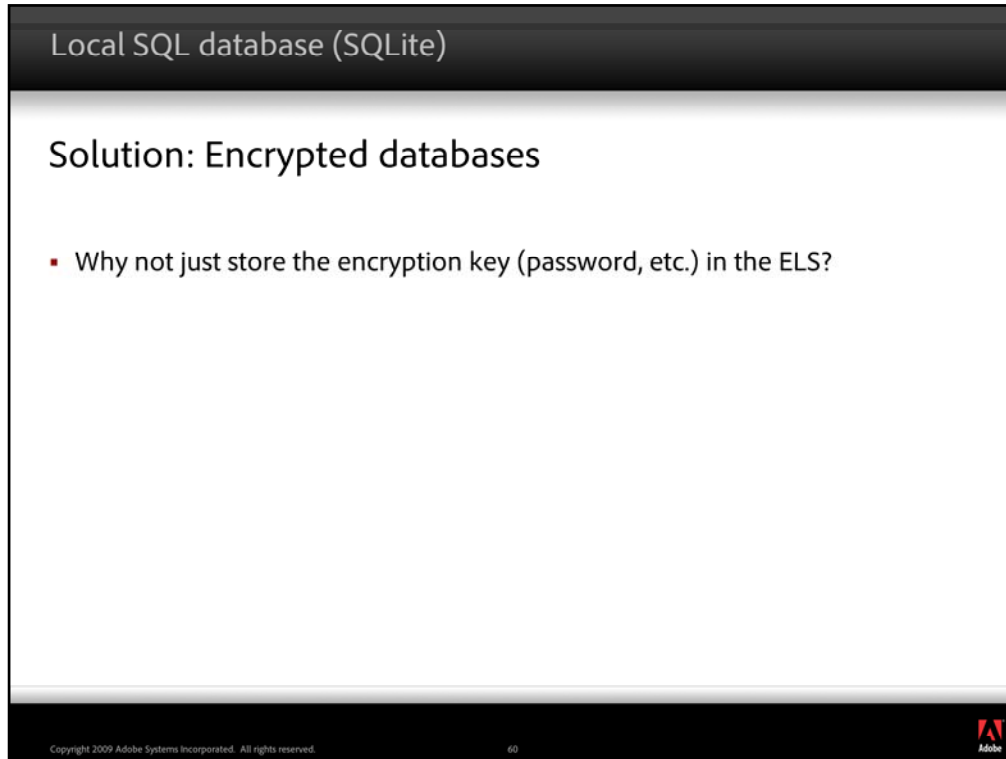
```
var keyGenerator:EncryptionKeyGenerator = new EncryptionKeyGenerator();  
var password:String = passwordInput.text;  
var encryptionKey:ByteArray = keyGenerator.getEncryptionKey(password);  
  
var conn:SQLConnection = new SQLConnection();  
conn.open(databaseFile, SQLMode.CREATE, false, 1024, encryptionKey);
```

In the process of working on the encrypted database feature I wrote an ActionScript class called the EncryptionKeyGenerator class that performs all those steps for you. The class is available as part of the as3corelib open source project. A complete example of how to use it is in the AIR documentation, but the basic steps are shown in this code listing:

- 1) Create an EncryptionKeyGenerator instance
- 2) Get a password from the user
- 3) Pass the password to the EncryptionKeyGenerator instance and it returns the encryption key.

If you're interested in understanding how the EncryptionKeyGenerator class works and how and why each of the steps listed on the previous slide are performed, that information is also included in the documentation.

Also, even though I'm mentioning the EncryptionKeyGenerator class in the section on encrypted databases, there's nothing specific to databases about it – you could use it to create an encryption key for any kind of two-way encryption.



The first question I asked was why not store the encryption key in the ELS, and then read it out every time. If you're going to do that you might as well just skip requesting a password and instead just create a random encryption key and store it in the ELS. However, if you do that then the data is vulnerable if an administrator impersonates another user account on the machine and runs the app.

For more details on this and other design decisions involved in the EncryptionKeyGenerator class, see the AIR documentation.

Thanks!

Thanks for listening and sharing!

Paul Robertson

<http://probertson.com/>

Copyright 2009 Adobe Systems Incorporated. All rights reserved. Adobe confidential.

61



