



SIN 12

**FAILING TO STORE AND
PROTECT DATA SECURELY**

OVERVIEW OF THE SIN

We often worry more about protecting information in transit than protecting the information while it is on disk, but the information spends more time stored on the system than it does in transit. There are a number of aspects you need to consider when storing data securely: permissions required to access the data, data encryption issues, and threats to stored secrets.

A variant of storing data securely is storing secrets in code, and we use the term “storing” very loosely! Of all the sins, this is the one that irks us the most, because it’s simply stupid. Many developers hardcode secret data into software, such as cryptographic keys and passwords, that they do not expect users to recover, believing that reverse engineering is too difficult to do. You may think it’s true, but if it’s not, those with malicious intent can reverse-engineer the code to divulge the secret data.

AFFECTED LANGUAGES

This is another one of those equal opportunity disasters. You can make data-access mistakes and embed secret data in any language.

THE SIN EXPLAINED

As you may have gathered, there are two major components to this sin; think of each as a *peccadillo*. Peccadillo #1 is weak access control mechanisms, and peccadillo #2 is hard-coding secret data. Let’s look at each in detail.

Weak Access Controls to “Protect” Secret Data

When it comes to the problem of setting access controls, there are significant cross-platform differences to consider. Current Windows operating systems support rich yet complex access control lists (ACLs). The complexity that is available cuts both ways. If you understand how to correctly use ACLs, you can solve complex problems that cannot be solved with simpler systems. If you don’t understand what you’re doing, the complexity of ACLs can be baffling and, worse yet, can lead you to make serious mistakes that may not adequately protect the underlying data.

While ACLs have traditionally been available on many UNIX systems through POSIX compliance, the access control system that has been uniformly supported is known as user-group-world. Unlike a Windows access control mask, which has a complex assortment of permissions, only 3 bits are used (not counting some nonstandard bits) to represent, read, write, and execute permission. The simplicity of the system means that some problems are difficult to solve, and forcing complex problems into simple solutions can lead to errors. The benefit is that the simpler the system, the easier it is to protect data. The Linux ext2 file system supports some additional permission attributes that go beyond the set of permissions that are commonly available.

Another difference between Windows systems and UNIX-based systems is that in UNIX-based systems, everything is a file—sockets, devices, and so on—and can be treated like a file. On a Windows system, there's a mind-boggling array of objects to consider, and each one of these has access control bits particular to that type of object. We won't get into the gory details of which bits apply to mutexes, events, threads, processes, process tokens, services, drivers, memory mapped sections, registry keys, files, event logs, and directories. As with many things, if you need to create an object with specialized permissions, you need to go Read The Fine Manual. The good news is that most of the time, the default permissions the operating system grants to most of these objects are the permissions you should be using.

ACLs and Permissions Redux

The user-group-world system grants permissions first based on the effective user ID (EUID) of the process that created the file. The group permissions depend on whether the operating system uses the effective process group ID, or the group ID of the directory that the file was created in. Finally, if the creator of the file or a high-level user allows access, everyone (world) may be granted access. When a process attempts to open the file, the access check first checks whether the user is the owner of the file, next whether the user is in the group designated for the file, and last the permissions that apply to anyone. An obvious consequence of this system is that it tends to depend on users being in the correct groups, and if the system admin doesn't manage groups correctly, many files end up with excess permissions granted to everyone.

Although there are actually a few different kinds of access control entries (ACEs) on a Windows system, they all have three things in common:

- A user or group identifier
- An access control mask to identify what actions the entry regulates (read, write, etc.)
- A bit to determine whether the entry allows or denies access

The user-group-world approach can be thought of as an ACL with a maximum of three entries, and a limited set of control bits. The way that an ACL is checked is that if the user or group identifier matches the user or a group that's enabled in the process's token, then the entry applies, and the access that's requested is compared with the access controlled by the ACE. If there is a match, and the ACE allows access, then you check to see if all of the access-requested flags match granted-access flags, and if so, then you've passed the access check. If you don't have all of the bits you need, then you continue to process more ACEs until you either get all of the access you requested or run out of ACEs to process. In the event that you hit an access denied ACE that matches both the access you've requested and a user or group in your token (whether enabled or not), then access is denied and no more ACEs are

ACLs and Permissions Redux (continued)

processed. As you can see, ACE order is important, and it's best to use APIs that will correctly order ACEs for you.

Another aspect of ACLs that creates more complexity is the possibility of inheritance. While files on UNIX will sometimes inherit the group from the parent container, any object on Windows that contains other objects—directories, registry keys, Active Directory objects, and a few others—will likely inherit one or more ACEs from the parent object. It isn't always a safe assumption that the inherited access control entries are appropriate for your object.

Sinful Access Controls

Because poor access controls aren't specific to any language, we'll focus on what problem patterns look like. But given a detailed look at the mechanisms used to create correct access controls across multiple operating systems could be a small book in its own right, we're only going to cover the high-level view.

The worst, and one of the most common, problem is creating something that allows full control access to anyone (in Windows, this is the Everyone group; in UNIX, it is world); and a slightly less sinful variant is allowing full access to unprivileged users or groups. The worst of the worst is to create an executable that can be written by ordinary users, and if you're really going to make the biggest mess possible, create a writable executable that's set to run as root or localsystem. There have been many exploits enabled because someone set a script to `suid root` and forgot to remove write permissions for either group or world. The way to create this problem on Windows is to install a service running as a highly privileged account and have the following ACE on the service binary:

```
Everyone (Write)
```

This may seem like a ridiculous thing to do, but antivirus software has been known to commit this sin time and again, and Microsoft shipped a security bulletin because a version of Systems Management Server had this problem in 2000 (MS00-012). Refer to CVE-2000-0100 in the "Example Sins" section that follows for more information about this.

While writable executables are the most direct way to enable an attacker, writable configuration information can also lead to mayhem. In particular, being able to alter either a process path or a library path is effectively the same as being able to write the executable. An example of this problem on Windows would be a service that allows nonprivileged users to change the configuration information. This can amount to a double-whammy because the same access control bit would regulate both the binary path setting and the user account that the service runs under, so a service could get changed from an unprivileged user to localsystem, and execute arbitrary code. To make this attack even more fun, service configuration information can be changed across the network, which is a great thing if you're a system admin, but if there's a bad ACL, it's a great thing for an attacker.

Even if the binary path cannot be changed, being able to alter configuration information can enable a number of attacks. The most obvious attack is that the process could be subverted into doing something it should not. A secondary attack is that many applications make the assumption that configuration information is normally only written by the process itself, and will be well formed. Parsers are hard to write, developers are lazy, and the attackers end up staying in business. Unless you're absolutely positive that configuration information can only be written by privileged users, always treat configuration information as untrusted user input, create a robust and strict parser, and best yet, fuzz test your inputs.

Another incarnation of this problem is when multiple applications have to share memory between them. Shared memory is a high-performance, unsafe, insecure way to do inter-process communications. Unless the application is robust enough to treat a shared memory section as untrusted user input, having writable shared memory sections typically leads to exploits.

The next greatest sin is to make inappropriate information readable by unprivileged users. One example of this was SNMP (Simple Network Management Protocol, also known as the Security Not My Problem service) on early Windows 2000, and earlier, systems. The protocol depends on a shared password known as a *community string* transmitted in what amounts to cleartext on the network, and it regulates whether various parameters can be read or written. Depending on what extensions are installed, lots of interesting information can be written. One amusing example is that you can disable network interfaces and turn "smart" power supply systems off. As if a correctly implemented SNMP service weren't enough of a disaster, many vendors, including Microsoft, made the mistake of storing the community strings in a place in the registry that was locally world-readable. A local user could read a writable community string, and then proceed to administer not only that system, but also quite possibly a large portion of the rest of the network.

All of these mistakes can often be made with databases as well, each of which has its own implementation of access controls. Give careful thought to which users ought to be able to read and write information.

One problem worth noting on systems that do support ACLs is that it is generally a bad idea to use deny ACEs to secure objects. Let's say, for example, that you have an ACL consisting of:

```
Guests: Deny All
Administrators: Allow All
Users: Allow Read
```

Under most scenarios, this works relatively well until someone comes along and places an administrator into the guests group (which is a really stupid thing to do). Now the administrator will be unable to access the resource, because the deny ACE is honored before any allow ACE. On a Windows system, removing the deny ACE accomplishes exactly the same thing without the unintended side effect, because in Windows, if users are not specifically granted access, they will be denied access.

Another Windows-specific problem is that the access token will be built by first including groups from the user's domain, next including groups for the domain that

contains the system being logged onto, and finally including groups provided by the local system. Where you can get into trouble is when you're delegating access to a resource contained by another system. If you just take the ACL from the object (one example might be an Active Directory object) and perform an access check locally, you could end up granting extra access unless you're careful to process the ACL and strip out locally defined groups, such as Administrators. This may sound far-fetched, but it's unfortunately a common problem because few networks have Kerberos delegation enabled. At one time, Microsoft Exchange had a problem with this root cause.

Embedding Secret Data in Code

The next goof up, hardcoding secret data, is really irksome. Let's look at an example. Your application needs to connect to a database server that requires a password, or access a protected network share using a password, or encrypt and decrypt data on the fly using a symmetric key. How are you going to do it? The simplest, and by far the worst (read: most insecure), way is to hardcode the secret data, such as the password or key, in the application code.

There is another reason for not committing this sin, and it has nothing to do with security. What about maintenance? Imagine your application is written in, say, C++, and is used by 1,200 customers. The application is sinful, and has an embedded encryption key used to communicate with your servers. Someone works out the key (it's not difficult, as we'll explain shortly), so you have to update the application for 1,200 users. And all the users must update because the key is now public, which means your servers need updating, which means all customers must update NOW!

Related Sins

Race conditions are a related sin where poor access controls enable some race condition attacks. This sin is covered in detail in Sin 16. Several other related sins involve processing untrusted user input. Another related problem is not using proper cryptography. Sometimes you can mitigate information disclosure problems with encryption, or if you must leave input information in a writable area, sign the information to mitigate tampering attacks.

Another related sin is failure to use the principle of least privilege. If your process is running as root or localsystem, the best access controls won't protect the operating system from your mistakes—the application can do anything, and access controls can't stop it.

SPOTTING THE SIN PATTERN

For the weak access control issue, look for code that:

- Sets access controls
- AND grants write access to low-privileged users

or

- Creates an object without setting access controls
- AND creates the object in a place writable by low-privileged users

or

- Writes configuration information into a shared area

or

- Writes sensitive information into an area readable by low-privileged users

For the embedded data sin, you should evaluate any code using any kind of encryption or creating outbound authenticated connections and determine where the password or key comes from; if it comes from within the code, you have a bug you need to fix (see the following section).

SPOTTING THE SIN DURING CODE REVIEW

For access controls, this is fairly simple: look for code that sets access. Carefully review any code that sets access controls or permissions. Next, look for code that creates files or other objects and does not set access controls. Ask whether the default access controls are correct for the location and the sensitivity of the information.

Language	Key Words to Look For
C/C++ (Windows)	SetFileSecurity, SetKernelObjectSecurity, SetSecurityDescriptorDacl, SetServiceObjectSecurity, SetUserObjectSecurity, SECURITY_DESCRIPTOR, ConvertStringSecurityDescriptorToSecurityDescriptor
C/C++ (*nix and Apple Mac OS X)	chmod, fchmod, chown, lchown, fchown, fcntl, setgroups, acl_*
Java	java.security.acl.Acl interface
.NET code	System.Security.AccessControl namespace Microsoft.Win32.RegistryKey namespace AddFileSecurity, AddDirectorySecurity, DiscretionaryAcl, SetAccessControl
Perl (*nix)	chmod, chown

For the embedded secrets sin, as a first pass, the author of this chapter likes to scan code for certain keywords to help determine if the code could be potentially sinful. Key words include:

- Secret
- Private (of course, you'll get a lot of noise from private classes!)
- Password
- Pwd
- Key
- Passphrase
- Crypt
- Cipher and cypher (sic!)

If you get hits on any of these words, determine if the word relates to embedded secret data, and if it does, make sure the secret is not within the code itself.

TESTING TECHNIQUES TO FIND THE SIN

For access control issues, install the application, and check the access controls set on any objects it creates. Even better still, if you have a way, is to hook the functions that create objects, and log the access controls. This helps you catch nonpersistent objects, like temporary files, events, and shared memory sections.

For embedded secrets, the easiest thing to do is to break apart a binary file using a tool like strings (www.sysinternals.com/ntw2k/source/misc.shtml#strings); this dumps all text strings in an application, and then sees if any of these look "password-ish" or totally random (a key, perhaps?).

Figure 12-1 shows the output from a small binary file. Look at the string right after Welcome to the Foo application. Looks like a bad attempt at hiding a password or key!

For applications written in .NET, such as VB.NET, J#, or C#, you can use `ildasm.exe`, available in the .NET Framework SDK, to perform a more in-depth analysis of an application. The following syntax helps divulge strings in the application, and some of the strings may be embedded secrets.

```
ildasm /adv /metadata /text myapp.exe | findstr ldstr
```

Figure 12-2 shows that we have hit pay dirt!

The second string looks like a random key, but there's more. The third string is a SQL connection string that connects to SQL Server as `sa` and has an embedded password to boot! But it gets even better (or worse, depending on who you are). The last string looks like a SQL statement using string concatenation. You may have just found an application that not only is hardcoding secrets but is also committing Sin 4, SQL injection.

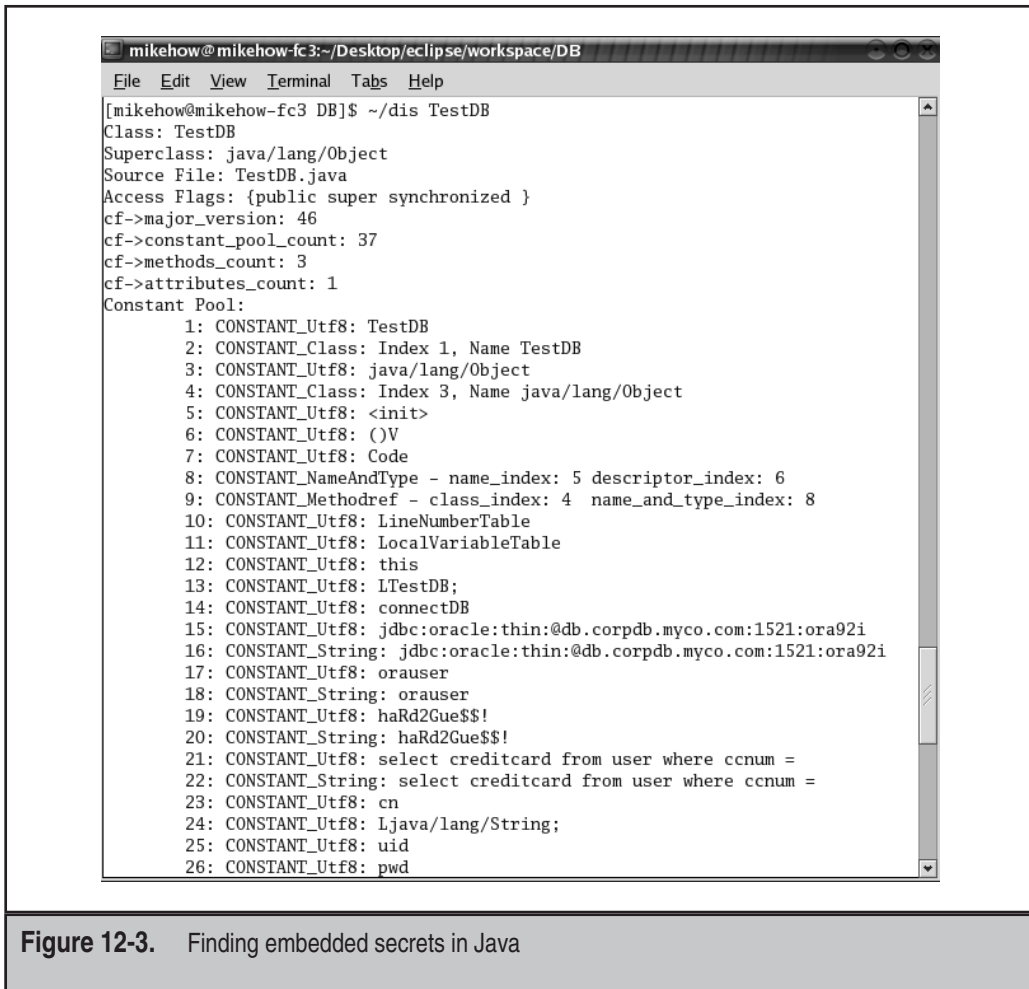


Figure 12-3. Finding embedded secrets in Java

Like the .NET sample, this shows some interesting data. Constant #15 is obviously a database connection string, #17 and #19 look like a username and password, and, finally, #21 looks like a SQL statement built using string concatenation. This is obviously very insecure code just waiting to be exploited. In fact, it looks like an exploit could be catastrophic. Look at the SQL statement; this code is handling credit card information!

Jad is another popular decompiler for Java, with an easy-to-use GUI named FrontEndPlus.

EXAMPLE SINS

The following entries in Common Vulnerabilities and Exposures (CVE), at <http://cve.mitre.org>, are examples of this sin.

CVE-2000-0100

From the CVE description: “The SMS Remote Control program is installed with insecure permissions, which allows local users to gain privileges by modifying or replacing the program.”

The executable run by the Short Message Service (SMS) Remote Control feature was written into a directory writable by any local user. If the remote control feature was enabled, any user on the system could run code of their choice under the localsystem context. (See www.microsoft.com/technet/security/Bulletin/MS00-012.mspcx.)

CAN-2002-1590

From the CVE description:

Web Based Enterprise Management (WBEM) for Solaris 8 with update 1/01 or later installs the SUNWwbdoc, SUNWwbcou, SUNWwbdev, and SUNWmgapp packages with group or world writable permissions, which may allow local users to cause a denial of service or gain privileges.

More information on this problem can be found at <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-1590> and www.securityfocus.com/bid/6061/.

CVE-1999-0886

From the CVE description: “The security descriptor for RASMAN allows users to point to an alternate location via the Windows NT Service Control Manager.”

More information on this problem can be found at www.microsoft.com/technet/security/Bulletin/MS99-041.mspcx. The RAS manager service had an ACL that was intended to allow any user to start and stop the service, but it allowed any user to also change the configuration, including the path to the service binary, which ran as the local system account.

CAN-2004-0311

American Power Conversion’s Web/SNMP Management SmartSlot Card AP9606 AOS versions 3.2.1 and 3.0.3 ship with a default, hardcoded password. A local or remote attacker with the ability to establish a Telnet connection to the device could supply an arbitrary username and the default password “TENmanUFactOryPOWER” to gain unauthorized access to the device.

CAN-2004-0391

According to the Cisco Security Advisory at www.cisco.com/warp/public/707/cisco-sa-20040407-username.shtml:

A default username/password pair is present in all releases of the Wireless LAN Solution Engine (WLSE) and Hosting Solution Engine (HSE) software. A user who

logs in using this username has complete control of the device. This username cannot be disabled. There is no workaround.

REDEMPTION STEPS

Weak access control is, for the most part, a design-level problem. The best approach to solving design-level problems is to use threat modeling. Carefully consider all of the objects your application creates, both at install time and at run time. One of the best code reviewers at Microsoft claims to find most of his bugs “Using notepad.exe and my brain”—use your brain!

A somewhat more difficult redemption step is to educate yourself about the platform(s) that you write code for, and understand how the underlying security subsystems really work. You must, we repeat, *must* understand how access control mechanisms work on your target platforms.

One way to keep out of trouble is to make a distinction between system-wide information and user-level information. If you do this, then setting access controls becomes very simple and you can usually take the defaults.

Now for the big section: remediation of embedded secrets. Remedying this sin is not necessarily easy; if it was easy, we wouldn't have this sin! There are two potential remedies:

- Use the operating system's security technologies
- Move the secret data out of harm's way

Let's look at each in detail, but before we continue, we want to point out a very important maxim:

“Software cannot protect itself.”

A malicious user with unbridled access to a computer system, and enough know-how could access all the secrets on the box, especially if the user is admin or root.

It's imperative that you always consider who you are defending the system against, and then determine if the defense is good enough for the sensitivity of the secret you want to store. Defending a private key used to sign documents that could live 20 years is a harder job than protecting a password used to allow access to the “Membership” section of a web site.

Let's first look at leveraging the OS as a defense.

Use the Operating System's Security Technologies

At the time of this writing, only Windows and Mac OS X support comprehensive system-wide capabilities to store sensitive data where the OS performs the critical (and difficult) key management. In the case of Windows, you can use the Data Protection API (DPAPI); and Mac OS X supports KeyChain.

DPAPI is very easy to use from any language running atop Windows. There's a full explanation of how DPAPI works on <http://msdn.microsoft.com>; a link to this page is in the "Other Resources" section.

The C/C++ redemption shows how to use native DPAPI, and the C# example shows how to use DPAPI from Managed Code and the .NET Framework Version 2.0.

NOTE

There is no class in .NET 1.x to call DPAPI, but there are many wrappers. Refer to *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002) for an example.

On Windows, you can also use Crypto API (CAPI) to access encryption keys, and rather than using the key directly, you pass a handle to a hidden key around the system. This is also explained in *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002).

C/C++ Windows 2000 and Later Redemption

The code that follows shows how to set up and call DPAPI in C/C++ on Windows 2000 or later. There are two functions in this code you must implement yourself; one returns a static BYTE* to the secret data, and the other returns a static BYTE* to some optional, extra entropy. At the very end, the code calls `SecureZeroMemory` to scrub the data from memory. This is used instead of `memset` or `ZeroMemory`, which may be optimized out by an optimizing compiler.

```
// Data to protect
DATA_BLOB blobIn;
blobIn.pbData = GetSecretData();
blobIn.cbData = strlen(reinterpret_cast<char*>(blobIn.pbData))+1;

// Optional entropy via an external function call
DATA_BLOB blobEntropy;
blobEntropy.pbData = GetOptionalEntropy();
blobEntropy.cbData = strlen(reinterpret_cast<char*>(blobEntropy.pbData));

// Encrypt the data.
DATA_BLOB blobOut;
if (CryptProtectData(
    &blobIn,
    L"Sin#13 Example", // optional comment
    &blobEntropy,
    NULL,
    NULL,
    0,
    &blobOut)) {
    printf("Protection worked.\n");
}
```

```

} else {
    printf("Error calling CryptProtectData() -> %x", GetLastError());
    exit(-1);
}

// Decrypt the data.
DATA_BLOB blobVerify;
if (CryptUnprotectData(
    &blobOut,
    NULL,
    &blobEntropy,
    NULL,
    NULL,
    0,
    &blobVerify)) {
    printf("The decrypted data is: %s\n", blobVerify.pbData);
} else {
    printf("Error calling CryptUnprotectData() -> %x",
        GetLastError());
    exit(-1);
}

if (blobOut.pbData)
    LocalFree(blobOut.pbData);

if (blobVerify.pbData) {
    SecureZeroMemory(blobOut.pbData, blobOut.cbData);
    LocalFree(blobVerify.pbData);
}

```

Here's the implementation of `SecureZeroMemory` used in Windows:

```

FORCEINLINE PVOID SecureZeroMemory(
    void *ptr, size_t cnt) {
    volatile char *vptr = (volatile char *)ptr;
    while (cnt) {
        *vptr = 0;
        vptr++;
        cnt--;
    }
    return ptr;
}

```

Or, as suggested by David Wheeler (see the "Other Resources" section):

```
void *guaranteed_memset(void *v, int c, size_t n)
{ volatile char *p=v; while (n--) *p++=c; return v;}
```

ASP.NET 1.1 and Later Redemption

This solution applies to web applications written using ASP.NET 1.1 and later. Because many web applications are database driven, the ASP.NET team made it very easy to securely store sensitive data, such as SQL connection strings, in a web.config file. Refer to Knowledgebase Article Q329290 for more information. (For the link, see the “Other Resources” section.) This tool uses DPAPI under the covers.

You can also use the `HashPasswordForStoringInConfigFile` method to store passwords in a configuration file.

C# .NET Framework 2.0 Redemption

The first example shows how to gather a password, and then write the protected password to a file. Note that DPAPI allows you to protect data so it is accessible only to the current user, or is accessible to all applications on the current machine. Your threat model should dictate which is most appropriate for your application.

```
byte[] sensitiveData = Encoding.UTF8.GetBytes(GetPassword());
byte[] protectedData = ProtectedData.Protect(sensitiveData, null,
                                             DataProtectionScope.CurrentUser);
FileStream fs = new FileStream(filename, FileMode.Truncate);
fs.Write(protectedData, 0, protectedData.Length);
fs.Close();
```

The next example shows the reverse process of opening a file and accessing the secret data inside:

```
FileStream fs = new FileStream(filename, FileMode.Open);
byte[] protectedData = new byte[512];
fs.Read(protectedData, 0, protectedData.Length);
byte[] unprotectedBytes = ProtectedData.Unprotect(protectedData, null,
                                                  DataProtectionScope.CurrentUser);
fs.Close();
```

NOTE

If you are using passwords in .NET Framework *String* classes, you should consider using the *SecureString* class instead. Refer to “Making Strings More Secure” in the “Other Resources” section.

C/C++ Mac OS X v10.2 and Later Redemption

There is sample code at http://darwinsource.opendarwin.org/10.3/SecurityTool-7/keychain_add.c showing how to add a password or key to the Apple Keychain.

The core functions are as follows: `SecKeychainAddGenericPassword` and `SecKeychainFindGenericPassword`:

```
// Set password
SecKeychainRef keychain = NULL; // User's default keychain
OSStatus status= SecKeychainAddGenericPassword(keychain,
        strlen(serviceName), serviceName,
        strlen(accountName), accountName,
        strlen(passwordData), passwordData,
        NULL);

if (status == noErr) {
    // cool!
}

// Get password
char *password = NULL;
u_int_32_t passwordLen = 0;

status = SecKeychainFindGenericPassword(keychain,
        strlen(serviceName), serviceName,
        strlen(accountName), accountName,
        &passwordLen, &password,
        NULL);

if (status == noErr) {
    // Cool! Use pwd
    ...

    // Now Cleanup
    guaranteed_memset(password, 42, passwordLen);
    SecKeychainItemFreeContent(NULL, (void*)password);
}
```

Redemption with No Operating System Help (or Keeping Secrets Out of Harm's Way)

This is a little hard to do, and is certainly not as good as having the operating system perform all the heavy work, but if the operating system you're targeting doesn't support the capability to "hide" secret data for you, then you need to create your own mechanism. The simplest way is to store the secret data out of the line of fire.

Remember we said earlier that you should always consider who you are defending against, and the value of the data being defended? If you have a web app protecting some sensitive data, you should always store the sensitive data outside the "web space." In other words, if the application resides in `c:\inetpub\wwwroot\myapp`, store the sensi-

tive data in c:\webconfig, or, better yet, d:\webconfig, as these directories are not in the line of fire. However, the wwwroot directory (and below) can be accessed by a remote web browser. Sure, your web server may not serve up text-based config files (such as web.config, app.config, and global.asa on IIS; and httpd.conf and .htaccess in Apache), but all it takes is a bug in the web application or the web server and the attacker could potentially read the sensitive data.

In a Windows system, you can also use the registry, which means a remote attacker has to get code running on the box to read the registry value.

On Linux, Mac OS X, or UNIX using Apache, you probably wouldn't want to store sensitive config data in the directory DocumentRoot points to (defined in httpd.conf). For example, on RedHat or Fedora Core, this is /var/www/html. The same applies to cgi-bin.

The following examples show how to read secret data from a resource outside the web line of fire.

Read from the File System Using PHP on Linux

```
<?php
    $filename = "/home/apache/config","r";
    $fh = fopen($filename);
    $data = fread($fh,filesize($filename));
    fclose($fh);
?>
```

Read from the File System Using ASP.NET (C#)

This code loads the filename from the app.config file that points to the file containing a SQL connection string. The app.config file looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="connectFile"
value="c:\\webapps\\config\\sqlconn.config" />
  </appSettings>
</configuration>
```

And the C# code to read this setting, and then get the connection string from the file, is

```
static string GetSQLConnectionString() {
    NameValueCollection settings = ConfigurationSettings.AppSettings;
    string filename = settings.Get("connectFile");
    (filename == null || filename.Length == 0)
        throw new IOException();

    FileStream f = new FileStream(filename, FileMode.Open);
    StreamReader reader = new StreamReader(f, Encoding.ASCII);
    string connection = reader.ReadLine();
}
```

```

        reader.Close();
        f.Close();

        return connection;
    }

```

You can also use the `aspnet_setreg` tool to store and protect configuration information.

Note that in the .NET Framework 2.0, `ConfigurationSettings` is replaced with `ConfigurationManager`.

Read from the File System Using ASP (VBScript)

This is a little less sophisticated than the preceding code because ASP doesn't use a config file. However, you could copy the name of the file into a variable in the `global.asa` file (ASP and IIS will not serve up `global.asa` by default), like so:

```

Sub Application_OnStart
    Application("connectFile") = "c:\webapps\config\sqlconn.txt"
End Sub

```

And then read it into your application when you need the SQL connection string:

```

Dim fso, file, pwd
Set fso = CreateObject("Scripting.FileSystemObject")
Set file = fso.OpenTextFile(Application("connectFile"))
connection = file.ReadLine
file.Close

```

Read from the Registry Using ASP.NET (VB.Net)

Rather than reading from a file, this code reads from the Windows registry:

```

With My.Computer.Registry
    Dim connection As String =
        .GetValue("HKEY_LOCAL_MACHINE\Software\" + _
                "MyCompany\WebApp", "connectString", 0)
End With

```

A Note on Java and the Java KeyStore

The JDK 1.2 and later provides a key management class named `KeyStore` (`java.security.KeyStore`) that can be used to store X.509 certificates, private keys, and, in some derived classes, symmetric keys. `KeyStore` does not, however, provide any key management facility to protect the keystore. So if you want to access a key from code, you should read the key used to encrypt and decrypt the store from somewhere out of the line of sight, such as a file outside the application's domain or outside the web space; and then use that key to decrypt the store, get the private key held within, and then use it.

You can place keys in a KeyStore using the keytool application that comes with the JDK, and then use code like this to extract the key:

```
// Get password used to unlock the keystore
private static char [] getPasswordFromFile()
{
    try
    {
        BufferedReader pwdFile = new BufferedReader
            (new FileReader("c:\\webapps\\config\\pwd.txt"));
        String pwdString = pwdFile.readLine();
        pwdFile.close();

        char [] pwd = new char[pwdString.length()];
        pwdString.getChars(0,pwdString.length(),pwd,0);
        return pwd;
    }
    catch (Exception e) { return null; }
}

private static String getKeyStoreName()
{
    return "<location of keyfile name>";
}

public static void main(String args[])
{
    try {
        KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());

        // get user password and file input stream
        FileInputStream fis = new FileInputStream(getKeyStoreName());
        char[] password = getPasswordFromFile();
        ks.load(fis, password);
        fis.close();
        Key key = ks.getKey("mykey",password);

        // Use key for other cryptographic operations

        ks.close();

    } catch(Exception e) { String s = e.getMessage(); }
}
```

This is by no means a great solution, but at least the key can be managed using `keytool`, and most importantly, the key is not in the code itself. The code also includes a common error noted in Sin 6, catching all exceptions.

EXTRA DEFENSIVE MEASURES

Here are a small number of useful defensive layers to add to your applications:

- Use encryption properly to store sensitive information, and signing to mitigate tampering threats when you cannot set strict ACLs.
- Use ACLs or permissions to restrict who can access (read and write) secret data if it must be persisted.
- Scrub the memory securely once you have finished with the secret data. This is often not possible in languages such as Java, or in Managed Code. However, .NET 2.0 adds the `SecureString` class to alleviate the issue.

OTHER RESOURCES

- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 6, “Determining Appropriate Access Control”
- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 8, “Cryptographic Foibles”
- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 9, “Protecting Secret Data”
- Windows Access Control:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/access_control.asp
- “Windows Data Protection”: <http://msdn.microsoft.com/library/en-us/dnsecure/html/windataprotection-dpapi.asp>
- “How To: Use DPAPI (Machine Store) from ASP.NET”: by J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy: <http://msdn.microsoft.com/library/en-us/dnnetsec/html/SecNetHT08.asp>
- Threat Mitigation Techniques: http://msdn.microsoft.com/library/en-us/secbp/security/threat_mitigation_techniques.asp
- Implementation of `SecureZeroMemory`:
<http://msdn.microsoft.com/library/en-us/dncode/html/secure10102002.asp>
- “Making Strings More Secure”:
<http://weblogs.asp.net/shawnfa/archive/2004/05/27/143254.aspx>
- “Secure Programming for Linux and Unix HOWTO—Creating Secure Software” by David Wheeler: www.dwheeler.com/secure-programs

- *Java Security, Second Edition* by Scott Oaks (O'Reilly, 2001), Chapter 5, "Key Management," pp. 79–91
- Jad Java Decompiler: <http://kpdus.tripod.com/jad.html>
- Class KeyStore (Java 2 Platform 5.0): <http://fl.java.sun.com/j2se/1.5.0/docs/api/java/security/KeyStore.html>
- "Enabling Secure Storage with Keychain Services": <http://developer.apple.com/documentation/Security/Conceptual/keychainServConcepts/keychainServConcepts.pdf>
- Java KeyStore Explorer: <http://www.lazgosoftware.com/kse/>
- "Enabling Secure Storage With Keychain Services": <http://developer.apple.com/documentation/Security/Reference/keychainservices/index.html>
- "Introduction to Enabling Secure Storage With Keychain Services": http://developer.apple.com/documentation/Security/Conceptual/keychainServConcepts/index.html#/apple_ref/doc/uid/TP30000897
- "Adding Simple Keychain Services to Your Application": http://developer.apple.com/documentation/Security/Conceptual/keychainServConcepts/03tasks/chapter_3_section_2.html
- Knowledge Base Article 329290: "How to use the ASP.NET utility to encrypt credentials and session state connection strings": <http://support.microsoft.com/default.aspx?scid=kb;en-us;329290>
- "Safeguard Database Connection Strings and Other Sensitive Settings in Your Code" by Alek Davis: <http://msdn.microsoft.com/msdnmag/issues/03/11/ProtectYourData/default.aspx>
- Reflector for .NET: <http://www.aisto.com/roeder/dotnet/>

SUMMARY

- **Do** think about the access controls your application explicitly places on objects, and the access controls objects inherit by default.
- **Do** realize that some data is so sensitive it should never be stored on a general purpose, production server—for example, long-lived X.509 private keys, which should be locked away in specific hardware designed to perform only signing.
- **Do** leverage the operating system capabilities to secure secret and sensitive data.
- **Do** use appropriate permissions, such as access control lists (ACLs) or Permissions if you must store sensitive data.
- **Do** remove the secret from memory once you have used it.
- **Do** scrub the memory before you free it.
- **Do not** create world-writable objects in Linux, Mac OS X, and UNIX.

- **Do not** create objects with Everyone (Full Control) or Everyone (Write) access control entries (ACEs).
- **Do not** store key material in a demilitarized zone (DMZ). Operations such as signing and encryption should be performed “further back” than the DMZ.
- **Do not** embed secret data of any kind in your application. This includes passwords, keys, and database connection strings.
- **Do not** embed secret data of any kind in sample applications, such as those found in documentation or software development kits.
- **Do not** create your own “secret” encryption algorithms.
- **Consider** using encryption to store information that cannot be properly protected by an ACL, and signing to protect information from tampering.
- **Consider** never storing secrets in the first place—can you get the secret from the user at run time instead?