Building secure applications

Jess Nielsen

Trapeze Group Europe A/S, Søren Frichs Vej 38K, DK-8230 Åbyhøj, Denmark

When developing applications the procedures are "normally" to apply the security aspects at the end of the development lifecycle. Applying the security aspects at the end of the development lifecycle is definitely wrong! Unfortunately this has the consequences that many forgets the security aspects or discard them because it is too expensive to apply them afterwards or because of the risks of reengineering costs. This article tries to highlight some of the security aspects that should be considered when actually implementing the application. By forcing the designer to consider a few rules of thumb doing the implementation phase it will also be a step closer to a development lifecycle where the security aspects are more integrated and thereby closer to the goal of making applications more reliable and secure.

EVERYONE IS EVIL

Crime that is related either directly or indirectly to IT has been changing drastic and it has become more in focus than it has ever been. During the fact that IT has been an everyday-tool it has also become a popular target for criminals and others that might find it interesting to break into foreign computer systems without proper permissions from the owner. Even home users have been popular targets! Due to this increasing crime it should be obvious that it is important to maintain a focus of building secure applications that makes it even harder for the criminals to break into the applications.

USE AUTHENTICATION

Designing applications that should be limited to special group of users, i.e. like an extranet, various kinds of authentication are possible.

Most developers might use a "normal" form-based authentication [1] with two textboxes for logon and password information. By pressing the submit button the two values from the textboxes are compared with the stored credentials. When the entered values match the stored credentials a session is created and the user will be redirected.

```
<%
Dim strUsername, strPwd As String
strUsername = Request.Form("Username")
strPwd = Request.Form("Pwd")
If IsValidCredentials(strUsername,
strPwd) Then
' Allow access to the user! Wee!
Else
' Bad username and password
Reponse.Redirect "401.html"
End If
%>
```

Even this kind of authentication is extremely popular; it will be insecure at all times.

First, instead of trying to implement a bulletproof authentication use the operating system, where it is possible. The simplest way of doing this is to use Windows integrated security or Windows Authentication. There is no reason to implement a second trusted computing base if it is not needed!

When using integrated security, Windows does support two major authentication protocols: NTLM and Kerberos v5 authentication. When using the SSL protocol [4] it further gives the possibility to use the X.509 certificate authentication. [1] [4]. The use of X.509 certificate authentication guarantees that both sides of the connection (meaning the client and the server) actually are the one they are pretending to be.

Secondly, when authentication information is entered there is no guarantee that it is securely transferred to the server unless it is a secure connection i.e. by the use of the SSL protocol.

In the matter of facts all servers, which are using the standard HTTP protocol and basic authentication are vulnerable. Form-based authentication used on a server that uses the HTTP protocol is an example to that kind of server unless the data is encrypted before it is transmitted on the unsecure HTTP channel.

Why use a form-based authentication at all? There is no need to design your own logon dialog. The operation system provides a standard dialog where even the image can be changed for some environments if needed.

- The CredUIPromptForCredentials in the Platform SDK does produce the logon dialog.

First, a common requirement for Windows desktop apps is to make the user log into the app, particularly if the app can't or doesn't use Windows integrated authentication. This is so common a requirement that Windows has what you need built in. Second, it is designed to resist the most well known attacks during the authentication and the dialog is hard to reproduce, which minimize the risk for a potential Trojan horse.

Note: After the dialog box is displayed and until the user or application closes it, no other window in the application can become active.

When implementing an application where neither SSL nor Windows authentication are available you have indeed a server that is vulnerable. What so ever regardless of this the site has to be restricted with some kind of authentication especially if it contains confidential information.

Using other kind of authentication than SSL or Windows authentication requires a second trusted computing base, which has to take care of the validation and storing of user credentials.

CRYPTOGRAPHIC FOIBLES

This introduces some problems because we have to make some explicit requirements to the container, which should hold the credentials.

1. Passwords may not be stored in clear text.
2. Prevention of offline dictionary attacks.
3. Passwords should be verified quickly.

The next step is how to achieve these requirements. The smart guy would properly mention encryption of some kind, but can it really be that simple?

To illustrate it hypothetical, let's try to follow the smart guy's advice and choose encryption of some kind. This will bring us to some issues to be solved before we can start implementing the trusted computing base.

- What kind of cryptosystem should be used when protecting the credentials?

Several kinds of encryption algorithms exist and it should be considered carefully which one to be used, when advantages and disadvantages for each have been found.

- Should symmetric or asymmetric keys be used for the encryption?

Two kinds of key pairs exist. Asymmetric key pairs are definitely the most secure encryption therefore the smart guy would properly say that it has to be asymmetric keys.

- Where and how should the keys be stored?

For both kinds of encryption algorithms a key set must exist. How should this key set be store and where? Remember if the key set is revealed the credentials are revealed as well!

- What is the impact on performance?

Everyone knows that encryption is definitely a time killer. It takes time and asymmetric keys takes several times longer than the symmetric keys. Second, we know that the operation has to be done several times per user; therefore performance is an important issue to our trusted computing base.

By this we can conclude the following by the use of encryption to store the credentials.

1. Passwords are stored encrypted and therefore not in clear text.
2. Even that you do not have the key set it will be possible to perform an offline dictionary attack or even a crypto analysis if you have the stored credentials.
3. Encryption is slow so we might have a problem with the performance.

Finally this means that encryption is not the right way to protect the credentials! First, it causes too many extraordinary issues that will complicate the implementation of the trusted computing base further.

Second, due the mathematical composition of especially the RSA algorithms, they do indeed take time to execute and may result in certain performance issues. To understand the reason for this I will briefly introduce the math behind it.

You might already know that the mathematical theory of RSA is based on large primes. This means that both keys contain a product of two large primes.

- Large primes = $\{p_1, p_2\}$
- Public key = $\{p_1 \times p_2, e\}$
- Private key = $\{p_1 \times p_2, d\}$

Please note the last part of the two keys; e and d. Consider the impact of this when putting them into the following equation, where they are replacing x.

$$t^{x} \bmod (p_1 \times p_2)$$

It should be clear that when x is a large number, the time to evaluate the expression would increase exponential. By definition d is a large number, which is kept private and changes from key set to key set while e is typically three (3), but not one (1).

The next question will properly be something like; do we have any other options? Yes, we do indeed have other options that will fulfill our requirements.

Instead of storing the credentials encrypted, why not store the hash value of them – hashing is much faster!

THE USE OF HASHING

A calculation that checks the hash value is simple and mathematically it can be done as shown with the function V [5] outlined below.

$$V(m, S(m)) = (\text{accept} \mid \text{reject})$$

According to the function V no keys are used and therefore there is (of course) no problem of storing them. What so ever a cryptographic hash function, which does take a key exists in the Windows API if needed.

- What is impact on performance?

A cryptographic hash function is designed to calculate the hash value in a fast way, which means that it will execute several times faster than with the use of encryption.

To satisfy the second requirement using a random salt can do it. Therefore by design each user has a random salt and hash value stored. The salt is changed often i.e. once or twice per day on occasional times.

Use random salts per user and change them regularly to prevent offline dictionary attacks.

In worth case, meaning if an attacker manages to get the hash values for all users then the intruder would not be able to use it. The intruder might be able to find a password, which matches the hash value, but when the salt is changed often, the hash value will be changed as well.

$$V(m + R_{user}, S(m + R_{user}))$$

The function V is now a revised version, where it is using a random salt under the user (R_{user}) and it will look like the one above. The way of creating a salted hash can be done using a very few lines of code as outlined below.

```
using System;
using System.Security.Cryptography;
using System.IO;
using System.Text;

byte[]  pwd, salt;
SHA1 sha1 = SHA1.Create();
UTF8Encoding utf8 = new
UTF8Encoding();
CryptoStream cs = new
CryptoStream(Stream.Null, sha1,
CryptoStreamMode.Write);
cs.Write(pwd,0,pwd.Length);
cs.Write(salt,0,salt.Length);
cs.FlushFinalBlock();
sha1.Hash; // holds the salted hash
```

Please note that the example uses the SHA1 algorithm to generate the hash. By this we can conclude the following by the use of hashing to store the credentials.

1. Passwords are not stored only the hashed values (and the salts) are.
2. It will be extremely hard (or maybe impossible within a short period of time) to make an offline dictionary attach because the hash values are changed regularly together with the random salts.
3. Hashing is fast and it is designed to make several executions within a short period of time.

PROTECTING KEYS

Cryptographic algorithms use keys to encrypt and decrypt data. These keys need to be long and hard to guess, but to make it user-friendlier we protect these keys with a password or pass phrase.

These pass phrases are often no longer than 10 - 12 characters for the human brain to remember. A standard pass phrase has to contain alphanumerical characters (and even special characters). The minimum length of 6 - 8 characters should be mandatory to make them harder to guess.

```
bool IsComplex(String phrase) {
Regex obj =
new Regex("^(?=.*\\d)(?=.*[a-z])
           (?=.*[A-Z]).{7,35}$");
return obj.IsMatch(phrase);
}
```

To ensure that the pass phrases do fulfill these requirements it is recommended to construct an algorithm or regular expression to ensure all pass phrases have this complexity. The one above ensures a minimum length of seven and the use of both small and capitalized letters as well as the use of numbers.

STORING KEYS

According to one of the previous bullets, an issue about storing keys is still remaining. The Crypto API will be helping us with this because it offers some functionality to do this.

First, when storing cryptographic keys most people thinks about using the PKCS#7 structure. The Crypto API in Win32 does not support this structure! Therefore if you have to export the key and store it then you have to use a third-party API (i.e. openSSL) to use a proper structure for that.

The safest way to store a secret is by storing it encrypted in the registry instead of storing it in configuration files.

Second, the operating system provides a machine key, which can be used to encrypt the data rather than using a password. In this way it is the operating system (and its built in trusted computing base) that holds the responsibility of storing the key securely.

The use of CryptProtectData and CryptUnprotectData implies the CRYPTPROTECT_LOCAL_MACHINE flag, which means it uses the machine dependent key.

When using the machine key, it is important to remember to backup the resulting cipher text because if the computer has to be rebuilt both the key used and the data will be lost!

ACCESS TO STORAGES

When storing secrets and user credentials most developers store the information in a database. There is no problem of storing secrets in a database as long as the access is restricted.

Many developers store the connection string in clear text in the WEB.CONFIG file and the connection often includes the password for the database.

Storing secrets in the database with the backdoor open is not secure at all!

When the backdoor is left open a potential attacker will be able to replace all the hashes with some that match passwords known to the attacker. This will give the attacker full access to the system and be a denial of service attack to others!

- Passwords, keys, and database connection strings should be stored out of the sight of the attacker. Placing sensitive data in the registry is more secured than placing it in harm's way. [1]

In the matter of facts the ASP.NET v1.1 does support optional Data Protection API encryption [6] of secrets stored in a protected registry key.

```
<system.web>
<processModel
enable="true"
username="registry:HKLM\Software\SomeK
ey,username"
password="registry:HKLM\Software\SomeK
ey,password"
/>
<system.web>
```

In the WEB.CONFIG file the configuration sections that can take advantages of this are only <processModel/>, <identity/>, and <sessionState/>.

Remember that this technique is not to be used to store arbitrary application data; it is only to usernames and passwords used for ASP.NET process identity and state service connection data.

USER RIGHTS

Whenever a user context has to be used either to a database or any other resource make sure never ever to use the SYSADMIN user or any user with administrative rights.

Always remember to choose a user with least privileges!

Does the database user contain to many privileges, the user might be able to perform a DROP DATABASE or a SHUTDOWN DATABASE. Both commands will result in a catastrophic system breakdown and thereby a mayor denial service attack.

SQL INJECTIONS

This brings me to the next subject, which are SQL Injections. [2] The attacker injects partial SQL expressions that will be replacing yours.

The form-based authentication is obvious to such an attack. An inexperienced developer might take the value from both fields and placing them into a SQL expression.

```
SELECT 1 FROM USER
WHERE USER='John' AND ASSWORD='secret'
```

When the query is executed it will return a row if the credentials are correct. Now lets change the value for the user.

```
HACKED' OR 1=1 --
```

Inserting this into the SQL statement will result in a totally different and much more dangerous SQL expression.

```
SELECT 1 FROM USER
WHERE USER='HACKED' OR 1=1 --' AND
PASSWORD='secret'
```

The expression tries to find a user named HACKED or one where 1 is equal to 1. The last part (1 = 1) is in matter of fact a tautology, which means that it is always true. The query will return as if it had been a "real" login and the attacker has now access to the system! Really, what about the last part of the expression? The last part of the expression has been comment out by standard SQL syntax and that is why it is never evaluated.

The next question is now how to prevent such attacks. It can be divided into two mayor rules of thumb.

1. Never ever connect as sys admin

First of all, when connecting to the database as sys admin you will definitely be violating the principle of running with least privileges. Secondly, if an attacker succeeds to perform a SQL injections attack against your application it will be with full permission to the entire database system.

2. Building secure SQL statements

The simplest way to prevent SQL injection is to leave the completion of the SQL expression to the database. This is done by the use of placeholders, which is often referred to as parameterized commands. [1]

```
SELECT 1 FROM USER
WHERE USER=? AND PASSWORD=?
```

Defining the expression using parameters is done as outlined above. Using parameterized commands are even faster, which gives you a second benefit of using them.

MALICIOUS HTML INPUT

When allowing the user to use HTML tags i.e. to format text inputs it is always extremely difficult to disallow all illegal tags.

It should always be considered carefully before allowing HTML input, because it can lead to compromise unless the solution is bulletproof.

The main idea is only to accept a small subset of HTML tags. This subset will typically contain some safe tags such as <EM>, <PRE>,
, <I>…</I> and <B>…</B>. Common for these tags is that they do not allow any event attributes such as mouse-over or others.

```
bool IsAccepted(String phrase) {
Regex obj =
new Regex("/^(?:[\s\w\?\!\,\.\'\"]*|
(?:\<\/?(?:I|b|p|br|em|pre)\>))*$/I
");
return obj.IsMatch(phrase);
}
```

The above regular expression [1] will allow a limited subset of HTML tags, spaces and alphanumerical characters.

PROTECTING YOUR CODE

In any object-oriented software inheritance exists and it is often used without doubts, but what happens if some code that is not trusted inherits from your code.

```
public class Authentication {
  protected string username;
  protected string password;
  public bool SignOn()
}

public class Evil extends
Authentication {
  public string GetCred() {
    // Information disclosure
    return username + " " + password;
  }
}
```

Notice the subclass that inherits from the Authentication class. It has direct access to both the username and the password! Second, the derived class can even implement its own SignOn() method and store each credentials before letting the users in.

Never trust derived classes!

Consider carefully which features you will allow to be accessed from the subclasses and use the SEALED keyword, where possible to avoid inheritance.

```
public sealed class Authentication {
private string username;
private string password;
public bool SignOn()
}
```

It is possible to protect your code by the use of permissions, but when using method permissions it is important to know that it goes only one level down.

Avoid wrapper methods that access methods protected by permissions.

Finally always use strong-named assemblies. It gives you three significant benefits, which are outlined below.

1. Versioning control is a version number that is assigned to an assembly. Since the version is required in order to assign the strong-name callers of the assembly can insure that the component is in fact the version they are looking to load. This prevents "code breakage", in which a newer (or older) component is loaded in place of the intended component.

2. The existence of satellite assemblies grants us the ability to include support for multiple languages in our application without luttering the primary, neutral assemblies that should execute in a culturally invariant context.

3. The final benefit is arguably also the most important. It is the public key token, and it verifies the identity of the assembly and insures that it was unmodified after compilation.

Strong-named assemblies are made using the Strong Name tool (Sn.exe). It helps you sign assemblies with strong names. [7]

CONCLUSION

This article describes the security concerns that should be considered when designing and implementing secure applications.

Avoid using multiple trusted computing bases. If possible use the one that sticks with the operating system and keep secrets away from the attacker – store them (encrypted) in the registry!

Always run the application with least privileges. A security hole in the application might give the attacker access to the operating system.

When using SQL statements use parameterized commands. They are safer and even faster to execute.

Never trust code that derives from your code. Protect the code in your classes and be sure that no secrets can be revealed.

Always use strong-named assemblies. It verifies the identity of the assembly and insures that it was unmodified after compilation.

During the development lifecycle it is recommended to have both dedicated security reviews and security tests. A bulleted checklist can be found in [1] for this purpose.

Jess Nielsen is a senior developer at Trapeze Group Europe A/S. His research interests include autonomic systems, software architectures and cryptographic. He received his master's degree in IT (Software Development) from Aarhus University, Denmark.
Contact him at jessn@bluebottle.com.

REFERENCES

[1] Write Secure Code, 2nd Edition
Michael Howard and David LeBlanc
ISBN 0-7356-1722-8

[2] SQL Injection - Are Your Web Applications Vulnerable?
SPI Dynamics, Inc. E-Mail: sales@spidynamics.com

[3] AES Proposal: Rijndael
Joan Daemen and Vincent Rijmen

[4] SSL 3.0 Specification
Alan O. Freier, Paul C. Kocher and Philip L. Karlton
http://wp.netscape.com/eng/ssl3

[5] An Introduction to some basic Concepts in IT.
Security and Cryptography.
Ivan Damgård, Ph.D
University of Aarhus

[6] 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

[7] .NET Framework Tools Strong Name Tool
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfstrongnameutilitysnexe.asp

FURTHER READING

- Writing Secure Code using C Sharp
http://www.c-sharpcorner.com/Security/CodesecureUsingCSharp1.asp
- Security Code Guidelines
http://java.sun.com/security/seccodeguide.html
- Security Code Review Guidelines
http://www.homeport.org/~adam/review.html#code-security
- Software Quality Assurance:
Documentation and Reviews
http://hissa.ncsl.nist.gov/publications/nistir4909/
http://msdn2.microsoft.com/ms182020.aspx
- Secure programmer: Developing secure programs
http://www-128.ibm.com/developerworks/library/l-sp1.html
- Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/SecNetch05.asp