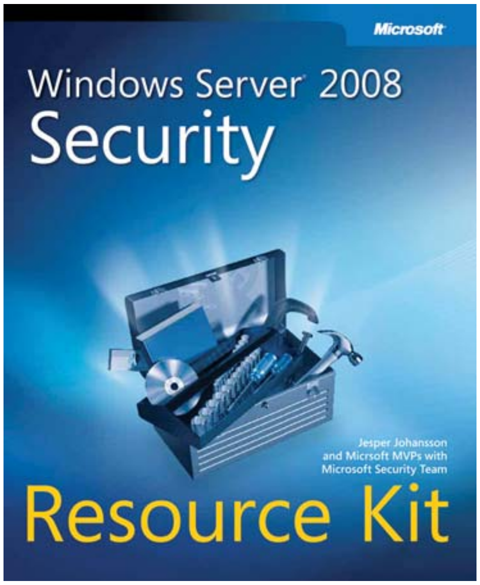


Windows Server® 2008 Security Resource Kit

Jesper M. Johansson and MVPs with the Microsoft Security Team

PREVIEW CONTENT This excerpt contains uncorrected manuscript from an upcoming Microsoft Press title, for early preview, and is subject to change prior to release. This excerpt is from *Windows Server® 2008 Security Resource Kit* from Microsoft Press (ISBN 978-0-7356-2504-4, copyright 2008 Jesper Johansson (Content); Jesper Johansson (Sample Code), all rights reserved), and is provided without any express, statutory, or implied warranties

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/11841.aspx

Microsoft Press

978-0-7356-2504-4

Table of Contents

Chapter 4

Authenticators and Authentication Protocols

Jesper M. Johansson

Once you have a subject, that subject needs some way to prove that it really is who it claims to be. Consider the very real-world case in which you want to purchase something with a credit card in a store where they actually understand security. You have your identity: you. However, the store's personnel do not know who you are so they require some proof—an authentication that you are who you say you are. To provide proof of identity you use an authenticator of some form, such as an identity card or a passport. You present this to the store clerk in a fairly routine fashion, as an authentication protocol.

The virtual world is no different, with the exception that the entity to which you have to authenticate understands that a signature on the back of a credit card is not an authenticator. Therefore, you need a stronger form of authentication. In this chapter, we will discuss how Windows handles authenticators and which authenticators it supports.

Something You Know, Something You Have

Generally speaking, there are three types of authenticators:

1. Something you know
2. Something you have
3. Something you are

Something You Know

A secret that you know, and in many cases share with the system you want to access, is the simplest and most pervasive form of authenticator. A password is a perfect example of something you know.

Something You Have

A token of some kind that you are in possession of is a different kind of authenticator. You authenticate as yourself by proving that you are in possession of this token. An example is a smart card (discussed later in the chapter) or a SecurID one-time password device (*http://www.rsa.com/node.aspx?id=1156*). These types of tokens are almost always combined with something you know, and can greatly strengthen the quality of the authentication claims.

Something You Are

Some systems use something you are as an authenticator. These typically fall in the category of biometric authenticators: tokens that attempt to measure something about you. Examples include retina scans, fingerprints, blood samples, voice recognition, and typing cadence. Biometric systems are inherently imprecise and, unlike the other two types of authenticator, must operate on a range, not an exact value. When you store your authenticator you must record it several times. Based on this, the system develops an acceptable range for your authenticator. To successfully authenticate, subsequent attempts must fall within that range.

Biometric systems suffer from many shortcomings. First, with the exception of typing cadence, they require hardware devices on every client, some of which can be quite intrusive.

Second, biometric systems are imprecise and a close match is all that is needed. If, for some reason, your biometric authenticator has changed, you will fail the authentication. For instance, if you use voice recognition you may not get in if illness or fatigue affects your voice.

Third, many people consider biometric authentication very intrusive. Having extremely personal details such as fingerprints stored on a computer system is not to many people's liking.

Fourth, many security experts consider biometrics oversold. The companies in the business of selling biometric systems often make impossible claims. For example, a company making a software solution that measures typing cadence claims to protect customers against keystroke loggers, making stolen passwords worthless. But this is impossible. The user must still type the password on the client, and a keystroke logger on the client could just be augmented to capture all the same information that the biometric software is capturing. This information could then be easily replayed to successfully authenticate.

Fifth, there is a common perception that biometric systems are secure because they are inherently a part of the user and cannot be left lying around the way passwords written on a sticky note can. However, this ignores the fact that biometric authentication sequences can not only be captured, but the tokens themselves are also most definitely removable. There have already been recorded instances of thieves making off with biometric authenticators.

Finally, there are relatively few choices for biometric authenticators. For example, in a system using fingerprints you only have 10 choices. If one of them is compromised or lost you have nine left to choose from. This makes cycling your authenticators difficult because you will run out relatively soon, and a weekend of ill-conceived do-it-yourself handiwork may very well prevent you from accessing your computer on Monday morning. As capturing and replaying credentials is a real risk this is a threat not to be discounted.

For all these reasons, Windows does not natively support biometric authentication. Third parties do produce add-on software for biometric authentication, and Microsoft also sells a fingerprinting device, although this latter device is clearly labeled as a non-enterprise grade security device. However, for all the reasons stated previously, these are not enterprise-class authenticators and should not be used in enterprises or to protect sensitive personal or corporate information. For enterprise use, smart cards and passwords can be far more secure, flexible, and easily integrated into ordinary business practices. The remainder of this chapter will focus on those two technologies.

Password Storage

Smart cards rely on certificates. (For more information about certificates, see Chapter 10, "Implementing Active Directory Certificate Services.") The smart card itself holds the secret portion of the certificate. The authentication system, in this case an Active Directory domain, holds the public portion. Therefore, when you use smart cards, no are secrets stored on the domain controllers (DCs) that need protection. This makes smart cards simpler in some ways than passwords to manage.

As a practical matter, most systems that use smart cards escrow the secret keys in a central location. Windows includes that functionality as well. By doing so you gain the ability to access any secrets protected with smart card credentials, for example, for forensic purposes. However, it also means that you now have a sensitive secret to protect.

Passwords, in virtually every implementation available today, are shared secrets. The secret the user uses to log on with is the same as the one the authentication server uses to authenticate the user's access. This means that passwords are sensitive secrets and must be protected.

In early computer systems, passwords were simply stored in clear-text in a text file. The passwords in those systems were never really meant to keep people out because only a small group of people had access to the system in the first place. They were mostly used to control which environment you received. Eventually, however, the passwords in the password file were encrypted or hashed.

Encryption and Hashing

Encryption is based on the word cryptography, which, literally, means "hidden writing." Encryption is the process of using cryptography to hide writing, or to convert something from a readable form—typically called clear-text or plaintext—into an obscured form, typically referred to as the ciphertext. Decryption is the reverse operation—converting something from ciphertext to plaintext.

While encryption uses cryptography to convert something into unreadable but reversible form, hashing is a closely related function that converts plaintext into unreadable and irreversible form. A hash can, for example, be used as a checksum to compare to plaintexts. If they both generate the same hash, you have reasonably good assurance that they are identical. A hash is also typically far smaller—proportional to the plaintext—than a ciphertext. Therefore, hashes are very well suited to uses like password storage.

Most Unix-based systems still use this exact form of password storage, with two slight modifications. First, the password file, typically stored in /etc/passwd, now contains no password hashes but just user names and IDs. The actual hashes are stored in the shadow password file—for example, in /etc/passwd.shadow. While the password file itself is world-readable, the shadow file is readable only by super users.

Second, because password hashes were originally world-readable in the /etc/passwd file, they had to be protected against comparison attacks. Imagine a situation in which you and I both have user accounts on the same computer. My password is "pas$word!" and, by sheer coincidence, you create the same password. With a straight hash, we would both have the same password hash stored in the /etc/passwd file. I could search the file for my hash, and then search for any other accounts with the same hash. If I found any, I would know that they had the same password I had. This is an unacceptable situation. The solution is to add a randomly generated salt to the password before hashing it. A *salt* is simply a random value that is added to the password before hashing it and then stored in clear-text in the password database. This way, even if two passwords are identical, they will have different salts and therefore different hashes. The process is shown in Figure 4-1.

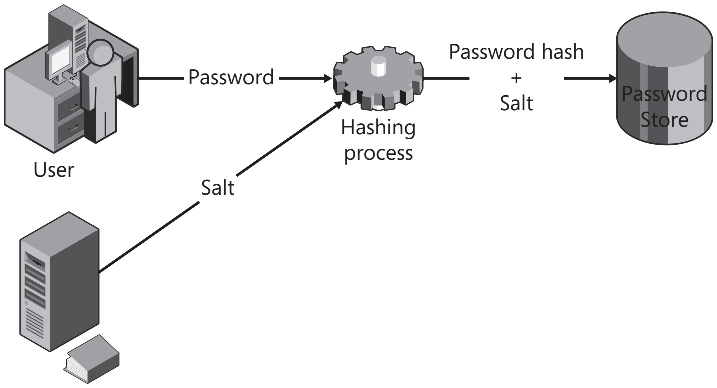


Figure 4-1 By salting the password before storing it the password file is protected against comparison attacks.

Windows uses variants on all these techniques to store its password. In the following sections I will cover the four primary ways Windows stores passwords used to authenticate users to Windows itself.

LM Hash

The LM hash is not actually a hash at all, although it has some of the same properties. It is a one-way function, and is usually referred to internally as the LMOWF (LanManager One-Way Function). In Windows Vista and Windows Server 2008 the LM hash is not stored by default, nor is it used by default during a network authentication. However, on down-level computers it is typically both stored and transmitted by default. Therefore, knowing how the LM hash works is worthwhile. Both Windows Vista and Windows Server 2008 can be configured to store or authenticate with the LM hash, but this is not recommended because of weaknesses in the algorithms.

Direct From the Source: LM Hash History

The LM hash was first used by Microsoft in its LAN Manager network operating system, the last version of which was released in the early 1990s. LAN Manager ran on top of IBM's OS/2 operating system. When Windows NT was first released in 1993 it was imperative that the new operating system interoperated with LAN Manager so that organizations that had invested in LAN Manager did not suddenly find that their investments were useless. Windows NT also provided a smoother upgrade path. However, this also meant that even though Windows NT supported far better security structures than LAN Manager, security concerns in Windows NT were caused by LAN Manager design decisions made in the mid-1980s. In 2006 Microsoft shipped the first operating system that disabled the LAN Manager password hashing mechanism by

> default, although it can still be enabled. It took 13 years to deprecate it.
>
> *Jesper M. Johansson, former Senior Program Manager for Security Policy, Windows Security MVP*

The LM hash is created using a large number of relatively complicated steps, shown in Figure 4-2. The process starts when a user creates a new password. The password is immediately converted to all uppercase. In other words, passwords stored using the LM hash are case-insensitive.

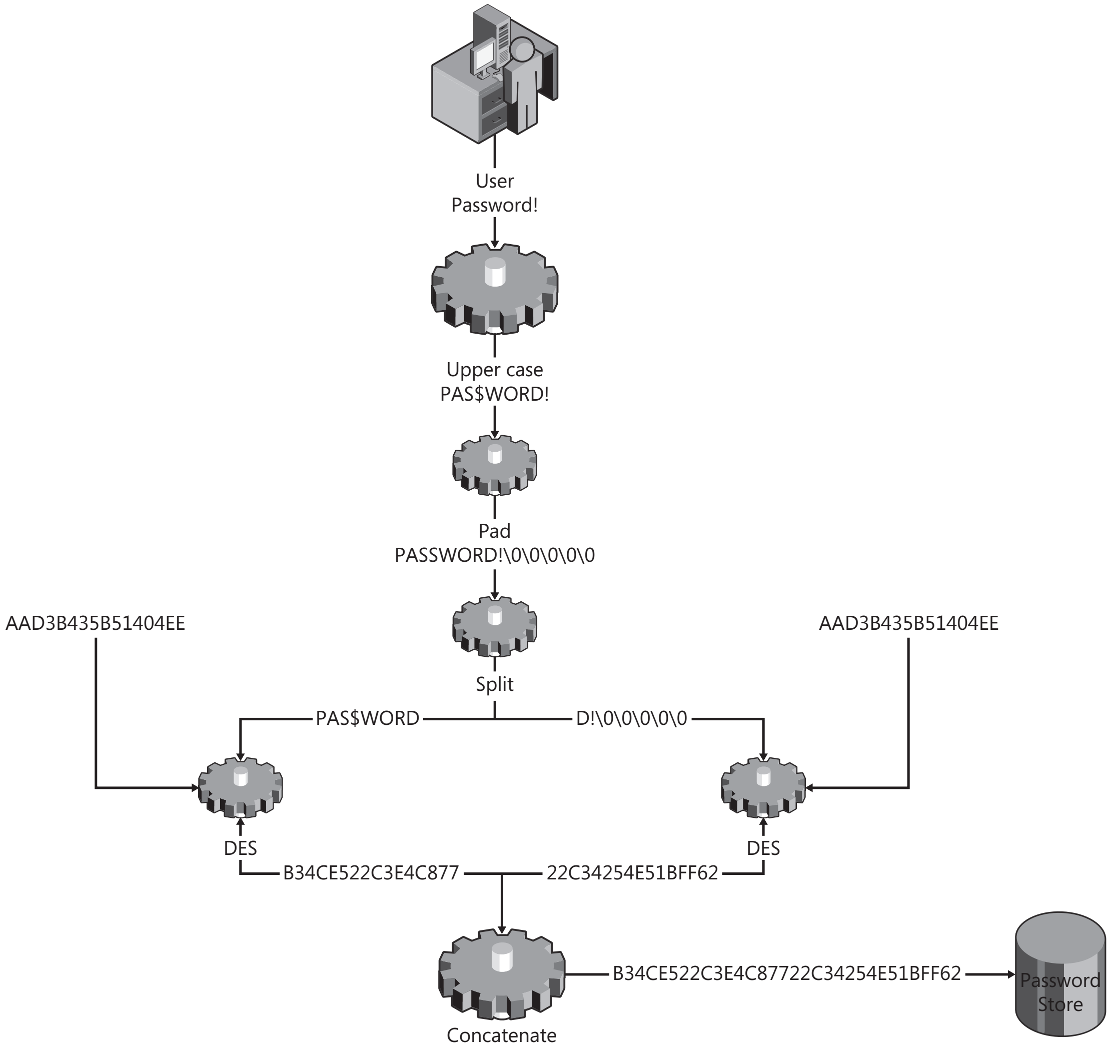


Figure 4-2 The LM hash is created using a series of complicated steps.

After the password is converted to uppercase it is padded out to 14 characters. If the password is already longer than 14 characters, it could theoretically be truncated at this point, but in practice, the process just fails and no LM hash is generated if the password is longer than 14 characters.

Next the password is split into two 7-character chunks. This is because they will now be used as a key in a Data Encryption Standard (DES) encryption, and the Data Encryption

Algorithm (DEA, the algorithm used in DES) operates on 56-bit chunks. These chunks are used as the key to encrypt a fixed value.

Finally, the results of the two DES operations are concatenated and the results are stored as the LM hash. The hash is stored either in the Security Accounts Manager database (if the password is for a local account on a stand-alone computer or a domain member) or in the DBCS-Pwd attribute of the user object in Active Directory.

This explains why an attacker is able to deduce how long a person's user name is just by looking at the hash. If the second half of the LM hash is AAD3B435B51404EE, the second half of the password is blank and the password is no longer than 7 characters. If both halves are AAD3B435B51404EE, the password is entirely blank.

NT Hash

When Windows NT first came out in 1993 a new password storage method was introduced. This mechanism is far simpler, as shown in Figure 4-3.

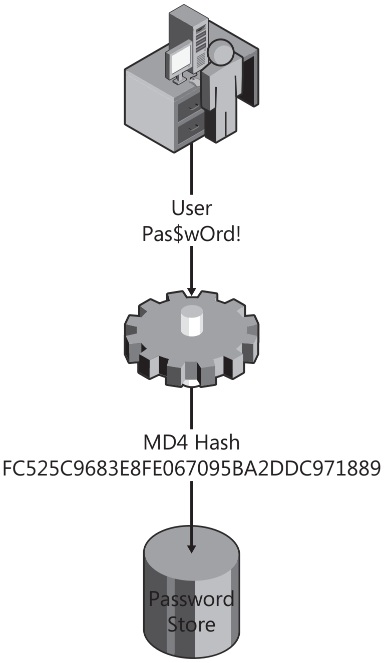


Figure 4-3 The NT hash is a straight MD4 hash.

The NT hash, or NTOWF as it is referred to internally, is stored either in the SAM or in the Unicode-PWD attribute of an AD user.

Note that neither the NTOWF nor the LMOWF are salted. Windows has never salted passwords for the simple reason that the password databases were never readable to others, so the lookup issue was never particularly interesting as an attack vector. To read the databases you have to be an administrator in the first place, meaning you have already fully compromised the computer or domain. Furthermore, shared-secret authentication systems have a very interesting property that we shall discuss shortly.

Password Verifier

If you have worked in a Windows Active Directory environment before you probably noticed that you can carry a domain-joined laptop computer with you and authenticate to it using a domain account even though you are not connected to the domain. This particular bit of magic is thanks to something called the password verifier. The password verifier, often referred to as cached credential outside of Microsoft, is a local copy of your domain password hash that you can use to log on locally. In operating system versions prior to Windows Vista, it was created using the process shown in Figure 4-4.

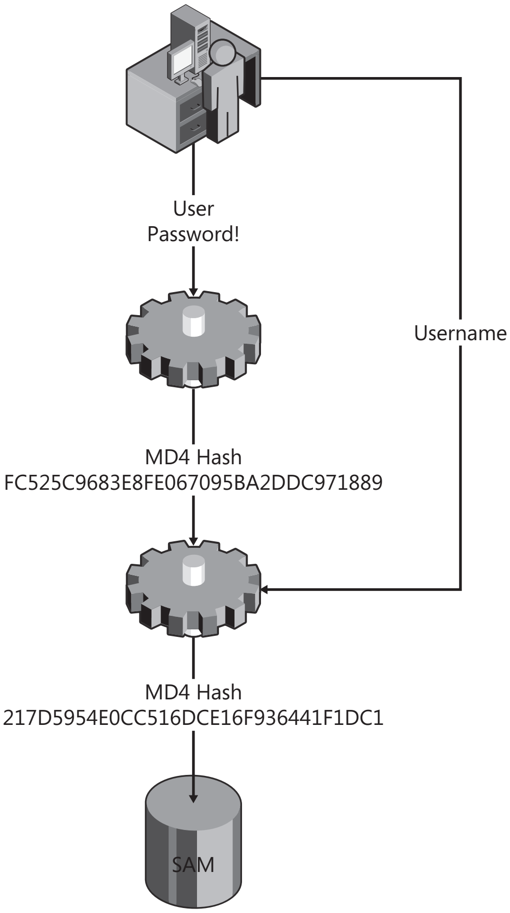

User
Password!
Username
MD4 Hash
FC525C9683E8FE067095BA2DDC971889
MD4 Hash
217D5954E0CC516DCE16F936441F1DC1
SAM

Figure 4-4 In down-level versions the password verifier was simply a hash of a hash, salted with the user name.

In recent years attackers have focused in on the password verifier and started creating tools to crack it. While it is a salted hash of a hash, and therefore quite difficult to crack, cracking it is possible if the password is not very strong. To combat this, in Windows Vista and Windows Server 2008 the calculation for the password verifier was modified, as shown in Figure 4-5.

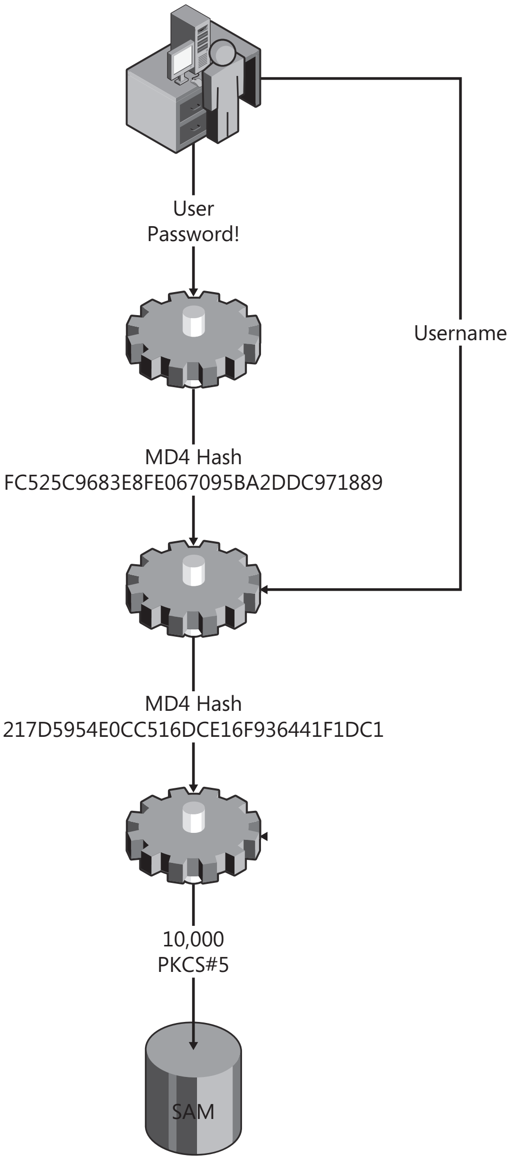

User
Password!
Username
MD4 Hash
FC525C9683E8FE067095BA2DDC971889
MD4 Hash
217D5954E0CC516DCE16F936441F1DC1
10,000
PKCS#5
SAM

Figure 4-5 The password verifier is far stronger in Windows Vista and Windows Server 2008 than in prior versions.

While there is no way to protect weak passwords, the improved password verifier calculation makes for a much stronger verifier. By running the old verifier through 10,000 PKCS #5 operations, a brute-force cracker would only be able to compute about 10 tests per second. This provides adequate protection against all but the very weakest passwords.

In Memory

When a user logs on interactively or using terminal services Windows caches the user's password hash (the NT hash and, if the computer is configured to store it, the LM hash). The hash is held in a memory location available only to the operating system, and of course, any process that can act as the operating system. When a user tries to access a network resource that requires authentication, the operating system uses this cached hash to authenticate with. As soon as the user logs off or locks the workstation the memory location is automatically purged.

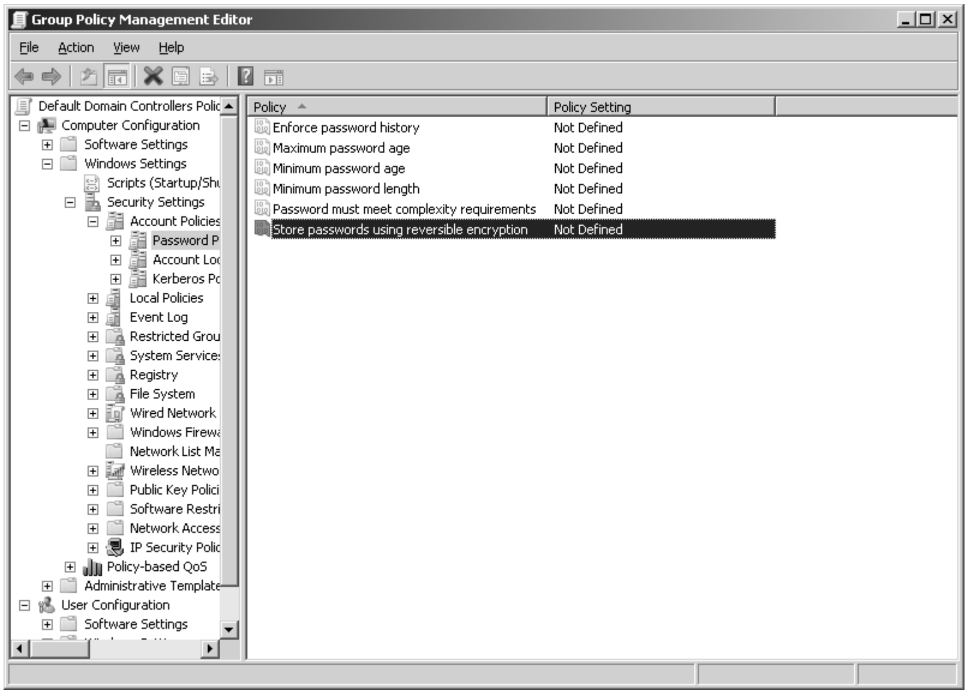
These hashes have been subject to a fair bit of debate after it was shown that if a domain administrator is logged on, any other user that is an administrator can read that domain administrator's password hash and use it to authenticate to a DC as a domain admin. This really should be obvious to any observer, however, and quite frankly, is putting far too much effort into it. If an attacker has compromised a workstation, it would be far easier to simply install a sub-authentication package, which gets the password in clear-text when it is typed during the logon process. These packages are supported to enable pass-through, single sign-on to non-Windows network devices, just like the NT hash is cached to support single sign-on to Windows devices. Although it would be possible to not support that, most users would rebel at having to type their passwords every time they accessed a network resource.

The problem, therefore, is really not with how Windows caches the NT hash, nor with sub-authentication packages, but rather with operational practices. A domain administrator should never log on interactively to a workstation used by a user with local administrative privileges unless that user is as trusted as all the domain administrators. By following this simple principle, you can keep this legitimate functionality from becoming an attack vector. For more information on managing this, see Chapter 12, "Securing Server Roles."

Reversibly Encrypted

Finally, Windows has an option to store passwords reversibly encrypted. When a password is stored reversibly encrypted, it can be reversed to plaintext. Obviously this means that no cracking is needed. Storing passwords reversibly encrypted is disabled by default, and is generally only needed in two circumstances. First, it is required if you need to use certain older authentication protocols for remote access, such as the CHAP protocol. Second, it is required if you want to perform advanced analysis on your passwords after they are set. For instance, some organizations want to go through and analyze whether passwords contain certain words. Those organizations must store the passwords reversibly encrypted.

To enable reversible encryption, or check whether it is still disabled, use the Group Policy editor, as shown in Figure 4-6.



Policy	Policy Setting
Enforce password history	Not Defined
Maximum password age	Not Defined
Minimum password age	Not Defined
Minimum password length	Not Defined
Password must meet complexity requirements	Not Defined
Store passwords using reversible encryption	Not Defined

Figure 4-6 To configure a computer or a domain to store passwords reversibly encrypted use the appropriate Group Policy setting.

The vast majority of organizations do not use reversible encryption, and as clients are upgraded to support more secure authentication protocols, there should be fewer and fewer reasons to do so. However, reversible encryption is another way Windows can store passwords, and it is important to know that it is there.

Many people cringe when they hear that Windows can store passwords reversibly encrypted. After all everyone knows that storing passwords in plaintext is bad. However, this really misses the point. In every password-based system today, *passwords are plaintext-equivalent*! Password-based systems use shared secrets. In the authentication process, the only secret used is the one that is stored on the authentication server. If an attacker gets hold of the authentication server's password database, he has everything he needs to authenticate. The only thing he needs to do now is insert himself at the appropriate step in the authentication process so that he can send the shared secret instead of the password it is derived from. Currently several tools are freely available on the Internet that do this with Windows authentication across the network.

The fact that passwords are plaintext-equivalent is not a security problem by itself. It only becomes a problem when an attacker obtains a password hash. However, as you should realize by now, those are fairly well protected in Windows. If an attacker manages to obtain a password hash, he has already compromised the computer as much or more than he would be able to with that password hash! In other words, that password hash gives him no additional privileges on an already compromised computer.

If passwords are reused across a network, however, it is possible that an attacker can further a compromise using the password hashes. Furthermore, because password hashes are cached in memory, an attacker may be able to obtain domain administrative credentials from a member computer if a domain administrator is logged on. This, however, is largely an operational problem related to how you run your network. If you follow the advice in Chapter 12, you will adequately protect yourself against that vector.

Authentication Protocols

So far we have discussed how passwords are stored on Windows. However, perhaps even more important is how they are used. Passwords are authenticators—they are used to authenticate a user to a computer. If the user is logging on interactively to a local account, the flow is quite simple:

4. User uses the Secure Attention Sequence (SAS, also known as the "three-finger salute," or just Ctrl+Alt+Delete) to bring up the logon dialog box. This causes the Local Security Authority Sub-system (LSASS) to spawn a new session and load WinLogon in that session. WinLogon in turn loads the LogonUI.
5. User types in the user name and password.
6. The WinLogon process takes the password, hashes it to an NT hash, looks up the user name in the local SAM, and compares the NT hash to the one that is stored for the user. If the two match, the logon is successful.
7. If sub-authentication packages are installed on the computer, the logon information is passed to those for additional processing. Otherwise, user32.exe is invoked and the user's environment is loaded.

This process is quite straightforward because there is a secured channel all the way from LogonUI, which takes in plaintext credentials, to the comparison of credentials. However, when authentication is taking place over the network it becomes a bit more complicated because you have to worry about how the authentication claims are transferred between the client where the user is sitting and the authentication server that hosts the accounts database. On Windows, this can take many forms, which I'll discuss in the following sections.

Basic Authentication

Basic authentication is the simplest of all forms of authentication. It just transmits the raw logon information across the network. In other words, the user name and password are sent across the network. This is also sometimes referred to as the Password Authentication Protocol (PAP). Basic authentication is quite common in older network protocols such as Telnet, FTP, POP, IMAP, and even in HTTP. Today it may be used, for example, in the RPC/HTTPS connector mechanism used to connect an Microsoft Office Outlook client to an Exchange server across the Internet. In that case the credentials are traversing inside an encrypted channel up to the Exchange Server or the ISA Server, whichever is terminating the connection. However, other than across an encrypted channel such as this basic authentication should be avoided.

Challenge-Response Protocols

Challenge-response protocols are designed to obviate the need to transmit a password in clear-text across the network. They all essentially operate the same way, shown in Figure 4-7.

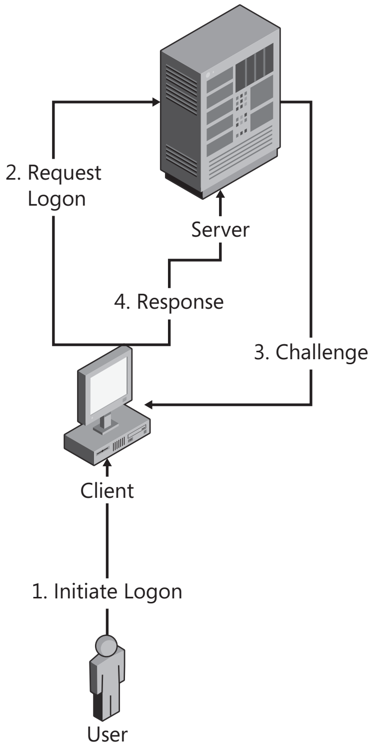


Figure 4-7 All challenge-response protocols are based on the same model.

The basic model for a challenge-response protocol is that a user initiates a logon, upon which the client makes a request to the server. The server creates a challenge, which often is just a random value, and sends this to the client. Meanwhile, the client has collected the user's credentials. The credentials are then combined with the challenge in a cryptographic operation. The result becomes the response. The actual implementation may differ, but the basic structure is always the same.

Digest Authentication

Digest authentication is not a native protocol in Windows. It is used primarily with Internet Information Services (IIS) in accordance with RFC 2617for Web-based authentication, and also with some third-party Lightweight Directory Access Protocol (LDAP) servers. Digest authentication is designed as a replacement to basic authentication. It is considered relatively weak, and makes some security tradeoffs on the authentication server.

The challenge-response sequence in a digest authentication is composed as follows:

1. The server generates a random nonce and sends it to the client.
2. The client computes an MD5 hash of the user name, authentication realm (domain in Windows), and password.
3. The client computes an MD5 hash of the method and the digest URI.
4. The client computes an MD5 hash of the result of operation 2, the server nonce, a request counter, a client nonce, a quality protection code, and the result from operation 3. This is the response value provided by the client.
5. The server computes all the same values.

The main concern with digest authentication happens in step 5. As you can tell, the client response is computed with the actual password, not a hash of the password. This means that to validate the client's response the server must have access to the clear-text password. Hence, if you want to support digest authentication, you must configure your domain to store passwords using reversible encryption.

LM and NTLM

Contrary to digest authentication, both LM and NTLM are considered native protocols in Windows. They are very similar, differing mainly in the hash used to compute the response. LM was first used in the LanManager product mentioned earlier. NTLM was designed as a replacement and released with Windows NT 3.1.

LM and NTLM are used in authentication in workgroups in Windows NT–based operating systems. They are also used in a domain environment if either the client or the server is not a domain member, or if the resource being accessed is specified using an IP address as opposed to a host name. Otherwise, Kerberos is used in Active Directory domains. The reason LM/NTLM must be used when accessing a resource using an IP address is that Kerberos is based on fully qualified domain names (FQDNs) and there is no way to resolve one of those from an IP address because each host can have multiple aliases.

The authentication flow in LM and NTLM is typically conjoined. The aggregate flow is shown in Figure 4-8.

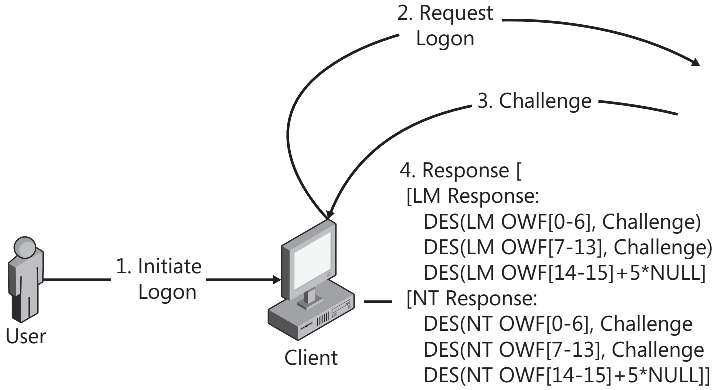

2. Request Logon
3. Challenge
4. Response [
[LM Response:
DES(LM OWF[0-6], Challenge)
DES(LM OWF[7-13], Challenge)
DES(LM OWF[14-15]+5*NULL]
[NT Response:
DES(NT OWF[0-6], Challenge
DES(NT OWF[7-13], Challenge
DES(NT OWF[14-15]+5*NULL]]
1. Initiate Logon
User
Client

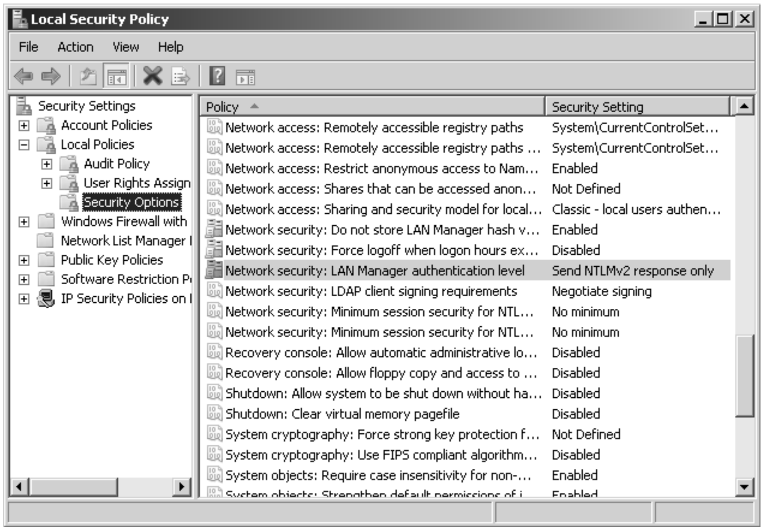
Figure 4-8 The LM and NTLM protocols are typically sent together.

All Windows NT–based operating systems prior to Windows Server 2003 worked as shown in Figure 4-8, sending both the LM and NTLM responses by default. In Windows Server 2003 only the NTLM response was sent by default, but both were accepted inbound. Starting with Windows Vista and Windows Server 2008, this has changed.

NTLM v2

Starting with Windows Vista, and also with Windows Server 2008, both LM and NTLM are deprecated by default. NTLM is still supported for inbound authentication, but for outbound authentication a newer version of NTLM, called NTLMv2, is sent instead. Technically speaking LM is also accepted for inbound authentication but neither Windows Vista nor Windows Server 2008 store the LM hash. Therefore there is no way for them to authenticate an inbound LM response.

The authentication behavior is controlled using the LMCompatibilityLevel registry setting, shown in Group Policy as Network Security: LAN Manager Authentication Level. See Figure 4-9.



Policy	Security Setting
Network access: Remotely accessible registry paths	System\CurrentControlSet...
Network access: Remotely accessible registry paths ...	System\CurrentControlSet...
Network access: Restrict anonymous access to Nam...	Enabled
Network access: Shares that can be accessed anon...	Not Defined
Network access: Sharing and security model for local...	Classic - local users authen...
Network security: Do not store LAN Manager hash v...	Enabled
Network security: Force logoff when logon hours ex...	Disabled
Network security: LAN Manager authentication level	Send NTLMv2 response only
Network security: LDAP client signing requirements	Negotiate signing
Network security: Minimum session security for NTL...	No minimum
Network security: Minimum session security for NTL...	No minimum
Recovery console: Allow automatic administrative lo...	Disabled
Recovery console: Allow floppy copy and access to ...	Disabled
Shutdown: Allow system to be shut down without ha...	Disabled
Shutdown: Clear virtual memory pagefile	Disabled
System cryptography: Force strong key protection f...	Not Defined
System cryptography: Use FIPS compliant algorithm...	Disabled
System objects: Require case insensitivity for non-...	Enabled
System objects: Strengthen default permissions of i...	Enabled

Figure 4-9 The LAN Manager Authentication Level setting governs the authentication behavior in non-domain authentication.

The default value for LMCompatibilityLevel in Windows Vista and Windows Server 2008 is 3, or Send NTLMv2 Response Only. Table 4-1 and Table 4-2 show how the possible values affect a computer when acting as the client and authentication server, respectively. It is important to recognize that the settings in Table 4-2 only relate to the server that performs the authentication, which is the one that contains the user accounts database. Any intermediate servers simply pass on the request to that server.

Table 4-1 Impact of LMCompatibilityLevel on Client Behavior

Level	Group Policy Name	Sends	Accepts	Prohibits Sending
0	Send LM and NTLM Responses	LM, NTLM NTLMv2 Session Security is negotiated	LM, NTLM, NTLMv2	NTLMv2 Session Security (on Windows 2000 below SRP1, Windows NT 4.0, and Windows 9x)
1	Send LM and NTLM—use NTLMv2 session security if negotiated	LM, NTLM NTLMv2 Session Security is negotiated	LM, NTLM, NTLMv2	NTLMv2
2	Send NTLM response only	NTLM NTLMv2 Session Security is negotiated	LM, NTLM, NTLMv2	LM and NTLMv2

Table 4-1 Impact of LMCompatibilityLevel on Client Behavior

Level	Group Policy Name	Sends	Accepts	Prohibits Sending
3	Send NTLMv2 response only	NTLMv2 Session Security is always used	LM, NTLM, NTLMv2	LM and NTLM

Table 4-2 Impact of LMCompatibilityLevel on authentication server behavior

Level	Group Policy Name	Sends	Accepts Inbound	Prohibits Sending
4	Send NTLMv2 response only/refuse LM	NTLMv2 Session Security	NTLM, NTLMv2	LM
5	Send NTLMv2 response only/refuse LM and NTLM	NTLMv2, Session Security	NTLMv2	LM and NTLM

NTLMv2 is a much improved version of NTLM. It also uses the NT hash. However, it also includes a client challenge in the computation. The aggregate flow is shown in Figure 4-10.

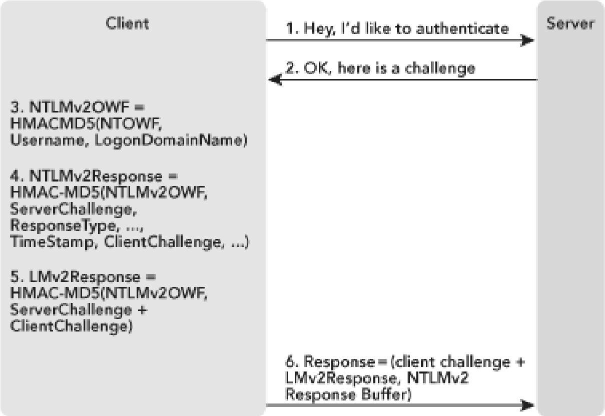

Client
Server
1. Hey, I'd like to authenticate
2. OK, here is a challenge
3. NTLMv2OWF = HMACMD5(NTOWF, Username, LogonDomainName)
4. NTLMv2Response = HMAC-MD5(NTLMv2OWF, ServerChallenge, ResponseType, ..., TimeStamp, ClientChallenge, ...)
5. LMv2Response = HMAC-MD5(NTLMv2OWF, ServerChallenge + ClientChallenge)
6. Response = (client challenge + LMv2Response, NTLMv2 Response Buffer)

Figure 4-10 The NTLMv2 protocol uses HMAC-MD5 and a client challenge.

As Figure 4-10 shows, the NTLMv2 protocol uses not only a client challenge, but also computes two HMAC-MD5 message authentication codes to create the response. It also includes a time stamp that mitigates replay attacks. Figure 4-10 also shows an LMv2 response, which is included in the response. The LMv2 response is a fixed-length response as opposed to the NTLMv2 response. It is included to provide the ability for pass-through authentication with down-level systems, such as Windows 95. Those systems did not support NTLMv2 natively, but did pass through the LM response. When NTLMv2 was first designed, those systems were prevalent, and they would strip pieces of the variable-length NTLMv2 response, breaking the authentication. To prevent this problem the LMv2 response was included in the LM response field. Because it has the same length as the LM response it is passed through to the authentication server unharmed and can be used to complete the authentication. Today it is still passed. However, the authentication server always starts out the authentication process by seeing whether there is an NTLMv2 response that validates successfully. If there is, the authentication succeeds. Therefore, while the LMv2 response still exists, it is rarely used for authentication.

NTLM++

Around the Windows 2000 time-frame, Microsoft added another NTLM-family protocol to Windows. This one does not have an official name. In some places in the implementation it is referred to as NTLM2, to contrast with NTLM3, which is actually NTLMv2. In other places it is called NTLM++. It was never documented, but was discovered externally by several people, including Eric Glass, Christopher R. Hertel, and Hidenobu Seki, and is even picked up by the Ethereal network traffic analyzer, which refers to it as NTLM2 Session Security. This is because it was always observed in conjunction with LMCompatibilityLevel set to 1, which enabled NTLMv2 Session Security. NTLM++ was added to make certain man-in-the-middle attacks more difficult, while retaining the ability to pass through authentication when connecting to down-level clients. In a sense, NTLM++ is an intermediate step between NTLM and LMv2/NTLMv2.

When NTLM++ is used the LM response field is populated with a client challenge instead of the LM response, as shown in Figure 4-11.

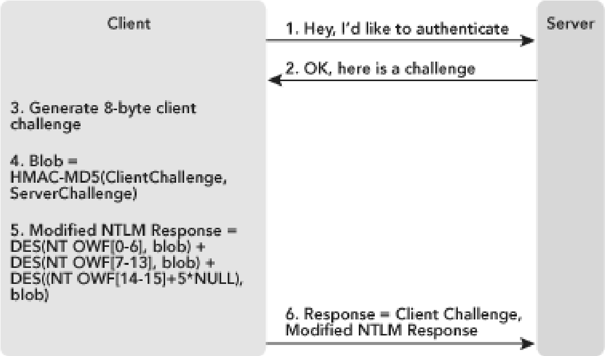

Client
Server
1. Hey, I'd like to authenticate
2. OK, here is a challenge
3. Generate 8-byte client challenge
4. Blob = HMAC-MD5(ClientChallenge, ServerChallenge)
5. Modified NTLM Response = DES(NT OWF[0-6], blob) + DES(NT OWF[7-13], blob) + DES((NT OWF[14-15]+5*NULL), blob)
6. Response = Client Challenge, Modified NTLM Response

Figure 4-11 The NTLM++ protocol includes a modified NTLM response and a client challenge.

The NTLM response field contains a modified NTLM response calculated exactly the same way as the original NTLM response, but using an HMAC-MD5 of the client challenge and the server challenge as the challenge, instead of just the server challenge.

NTLM++ is used whenever NTLMv2 Session Security is enabled. Starting with Windows 2000 Security Rollup Pack 1, all computers will automatically send the NTLM++ response on the first attempt. This means that starting with that release, the effective LMCompatibilityLevel setting is actually 1 on all computers.

For reference, NTLMv2 Session Security also includes stronger computation of session keys that are used by applications that request session security after the connection is set up.

Kerberos

Kerberos is used in domain environments when host names are used to connect. This is most of the time, unless the user specifically requests a connection to an IP address. Like the NTLM family, Kerberos is implemented as a Security Support Provider (SSP) and Kerberos also uses the NT hash for authentication, but any similarities with the other protocols really end there.

Kerberos is designed to provide authentication both for the user who is trying to connect and the authentication between the client and the server. This is quite a departure from NTLM, which does not provide the user with any assurance that the server is the one she thinks it is. Kerberos is also designed with the explicit assumption that the network is hostile; that all traffic is being intercepted by the adversary; and that the adversary has the ability to read, modify, or delete any traffic sent across the network.

To accomplish all this, Kerberos relies on encryption as well as time synchronization. By default in Windows, the synchronization between client and server must be within five minutes of each other. You can modify this setting if you are in an environment with high potential skew. To do so, change the maximum Tolerance For Computer Clock Synchronization value in Computer Configuration\Windows Settings\Security Settings\Account Policies\Kerberos Policies in a GPO that applies to the computers for which you want to change the time skew.

To understand how Kerberos works, let's analyze the exchange shown in Figure 4-12, which shows how a user logs on to a workstation and then requests a file from a file server.

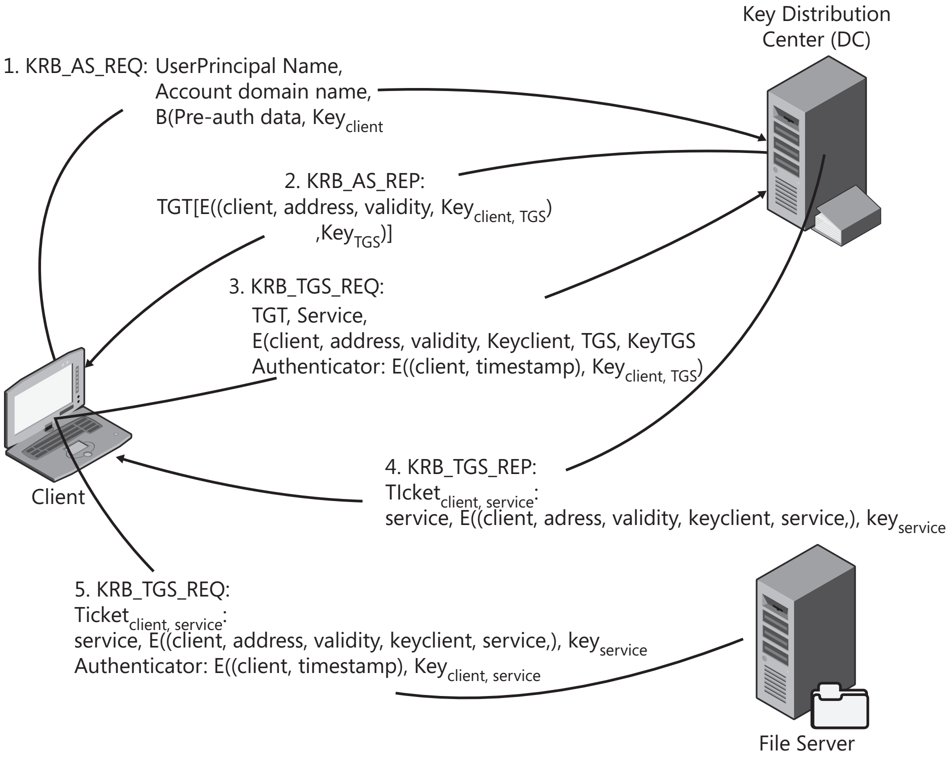

Key Distribution Center (DC)
1. KRB_AS_REQ: UserPrincipal Name,
Account domain name,
B(Pre-auth data, Key$_{client}$
2. KRB_AS_REP:
TGT[E((client, address, validity, Key$_{client, TGS}$)
,Key$_{TGS}$)]
3. KRB_TGS_REQ:
TGT, Service,
E(client, address, validity, Keyclient, TGS, KeyTGS
Authenticator: E((client, timestamp), Key$_{client, TGS}$)
Client
4. KRB_TGS_REP:
TIcket$_{client, service}$:
service, E((client, adress, validity, keyclient, service,), key$_{service}$
5. KRB_TGS_REQ:
Ticket$_{client, service}$:
service, E((client, address, validity, keyclient, service,), key$_{service}$
Authenticator: E((client, timestamp), Key$_{client, service}$
File Server

Figure 4-12 This exchange occurs when a computer starts and requests a file from a file server.

The exchange in Figure 4-12 consists of the following parts:

1. After the computer starts it creates some pre-authentication data, consisting of, among other things, a time stamp. This pre-authentication data is encrypted using a key derived from the computer's password. It is then packaged in a KRB_AS_REQ (Kerberos Authentication Service Request) packet and sent to the Authentication

Service (AS) which resides on the Key Distribution Center (KDC), which, as it turns out, is the DC.

2. The AS constructs a Ticket Granting Ticket (TGT) and creates a session key that the client can use to communicate with the Ticket Granting Service (TGS), which also resides on the DC. This key is denoted with $\text{Key}_{\text{client,TGS}}$ in Figure 4-12. It is transmitted to the client encrypted with the client's own public key. This message is sent back as the KRB_AS_REP.
3. The client now sends a KRB_TGS_REQ message to the Ticket Granting Service (TGS) on the KDC to request a ticket for the file server. This request has the TGT in it, and also includes the service the client wants to access and information on the client encrypted with the TGS public key. The KRB_TGS_REQ includes an authenticator, which is essentially a time stamp encrypted with the session key the client shares with the TGS.
4. The TGS responds with a KRB_TGS_REP message that includes a ticket for the service the client requested. It contains the same information the client sent in the KRB_TGS_REQ, but this time is encrypted using the server's public key. In other words, the client cannot read this data. The TGS also creates a session key that the client can share with the server and encrypts it with the session key the client shares with the TGS.
5. Finally, the client sends its ticket for the service to the server. The client information, along with the client-server session key, is encrypted using the server's public key, and the message also includes the client's authenticator, which is encrypted using the shared session key.

When a user logs on to the client, the same process is repeated, but this time the messages include user information. The Kerberos client sends another KRB_AS_REQ, but encrypts the pre-authentication data with a key derived from the client's password—or rather, the client's NT hash. The KDC validates the authentication based on that information. In the KRB_AS_REP the client receives a TGT that the user can use to contact the TGS. The TGT includes session keys for the KDC along with Security Identifiers (SIDs) for the user and all the groups the user is a member of. From then on, the client will use the user's TGT for requests made on behalf of the user.

Kerberos is clearly a rather complicated protocol, but it has proven remarkably robust in Windows. It also proves to be extensible in that the user's pre-authentication data can just as easily be encrypted with some secret not derived from a password. This happens in smart card–based authentication.

Smart Card Authentication

A smart card is, in most cases, a credit card–sized device that contains a memory chip. These devices have many uses. For example, they are used to provision a phone's identity in the Global System for Mobile communications (GSM) cellular telephone system and its derivatives. Smart cards may also be used to authenticate to Windows. In that case they contain an X.509 certificate. (See Chapter 10 for more information about certificates.) The certificate contains a private key, and the corresponding public key is stored in the user object in Active Directory.

When the user authenticates using a smart card, WinLogon will ask for a PIN code instead of a password. It then contacts the smart card provider and provides it with the PIN code along with the pre-authentication data. The smart card provider uses the PIN code to access the smart card, which will encrypt the pre-authentication data that the Kerberos SSP will use in the KRB_AS_REQ message. From then on, most things happen the same way in a smart card logon as in a normal password-based logon, with one major difference: If the user logs on with a smart card, she never provided a password. This means that if the user tries to access any resources that cannot use the Kerberos system, the computer must prompt her for a password. To avoid that, Windows handles passwords a bit differently in smart card–based logons.

Smart Cards and Passwords

All accounts have a password hash stored on the DC. Even if a user logs on with a smart card, a password hash is still there. In fact, even if the user is required to log on with a smart card there is a password hash. When you configure an account to require smart card logon, the DC will actually create a random password, hash it, and store it in the user object.

When a user logs on with a smart card, the KDC actually provides the client with the user's password hash during the logon process. These credentials are sent encrypted with the client's public key. The Kerberos SSP on the client will decrypt them and cache them in the same way it would cache them if the user had entered them at the logon prompt. These credentials are then used to log on seamlessly to computers that, for whatever reason, cannot be reached using Kerberos. This means that even with smart card logon required, the hashes are still exposed on the client to any rogue software that happens to run as an Administrator. Using smart cards does not protect the password-based credentials any more than password-based logons do. Therefore, all the same cautions apply against the attacks we shall discuss next.

Attacks on Passwords

At this particular juncture, it is worth taking a little detour into attacks, if for no other reason than that so many people are concerned about them. The primary concern with respect to passwords is obviously bad guys getting at them. Once they have them, or some representation thereof, the question is how they use them. Let's start by investigating how a bad guy can obtain a password, or some form of it.

Obtaining Passwords

Bad guys have several ways to get hold of your passwords. The following sections list them in order of ease of attack and prevalence (roughly speaking).

Ask For Them

An astonishing number of people, up to three-quarters in some surveys, are willing to part with their passwords in trade for something they value more, like chocolate.

Capture the Passwords Themselves

The most fruitful, simplest, and possibly most common way to attack passwords today is to use a keystroke logger to capture them in plaintext as they are being entered. There are many different kinds of keystroke loggers. An innocuous option is using a hardware device that mounts between the keyboard and the computer and has onboard memory to hold all keystrokes. It can be surreptitiously installed or removed in a matter of seconds. A software program, commonly found in malware and spyware today, will capture all keystrokes, not just passwords. Some of these include an automatic upload feature to a Web site or an IRC channel. Others include a small Web server that the attacker can use to retrieve the goods. However, the simplest and most direct route for an attacker to capture only passwords is to write an authentication package. Windows, like any other industrial-strength operating system, includes functionality for third parties to extend its authentication subsystem to authenticate to other network devices. An attacker can, with just a few application programming interface calls, write an authentication package that will receive all passwords in plaintext when a user logs on. The package can be augmented with the same features as a more general keystroke logger, but generates far less noise. Both of the software options require administrative privileges to install, meaning that the computer must be completely compromised to get to them. Physical compromise would also be sufficient to install one of these, and it is quite telling that keystroke loggers are now found regularly on public access computers, especially at conferences.

Capture the Challenge-Response Sequence

It is rare that passwords are passed on the network in any form today, and even rarer with plaintext protocols such as FTP, POP, and Telnet. However, the attacker can often capture both the challenge and the response and attack the combination. It requires more calculations than attacking ordinary hashes, but can be very fruitful if the password is weak.

Capture the Hashes

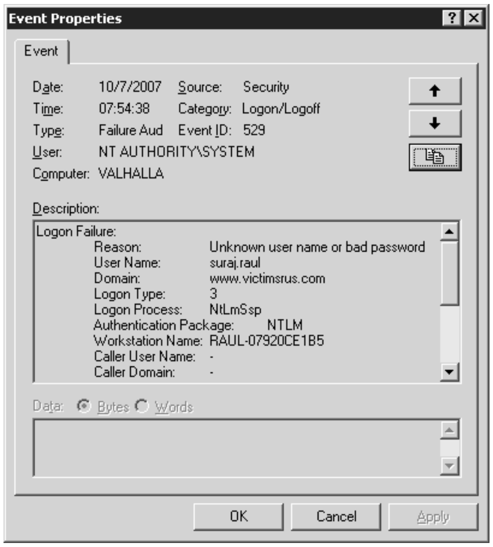
This is the quintessential attack that everyone worries about. If an attacker has access to the password hashes, he can crack them or use them in some other way. There are several ways to crack them, as we shall see shortly. The most common way to capture the hashes is to compromise the authentication server that stores the passwords. As you will see in Chapter 14, "Securing the Network," the more dependencies you have in your network, the easier this attack is to perpetrate.

Another option—less common but equally valid—is to compromise a computer where someone is already logged on. When a user logs on, as I mentioned earlier, Windows caches that user's NT hash in memory. An attacker with complete control over the computer can retrieve that hash and use it in the same way as any other hash. Again, this is a problem largely related to your operational practices. If you do not expose sensitive hashes on computers that are less sensitive (and hence less secure) you will not have this problem. In addition, if a criminal manages to compromise a computer to this extent, she can easily capture the plaintext password as well, as we will see in the next section.

It is important not to lose sight of the fact that in every case that involves compromise of actual hashes, the bad guy has defeated all the security systems and has complete control over at least a system that will provide him with advanced access, and probably to a system that holds all the secrets—the DC. In other words, if a bad guy has hashes to crack, you have already been severely hacked and bad guys with password hashes should be the least of your concerns. Regardless of whether the bad guy manages to use the hashes directly or crack them, your network is beyond repair already. Your only solution is to rebuild any compromised computer—including the entire network if a domain or enterprise admin account could be compromised—from scratch or a backup that is provably not compromised.

Guessing Passwords

Finally, the bad guy can simply try to guess passwords. Anyone who has an Internet-connected Windows computer and actually looks at the log files will see attempts at this. Figure 4-13 shows a failed attempt on one of my computers on the day I was writing this chapter.



```
Date:      10/7/2007    Source:    Security
Time:      07:54:38     Category:  Logon/Logoff
Type:      Failure Aud  Event ID:  529
User:      NT AUTHORITY\SYSTEM
Computer:  VALHALLA

Description:
Logon Failure:
        Reason:                 Unknown user name or bad password
        User Name:              suraj.raul
        Domain:                 www.victimsrus.com
        Logon Type:             3
        Logon Process:          NtLmSsp
        Authentication Package:         NTLM
        Workstation Name:       RAUL-07920CE1B5
        Caller User Name:       -
        Caller Domain:          -

Data:  ● Bytes  ○ Words
```

Figure 4-13 Anyone with an Internet-connected Windows computer will get failed logon attempts in their event logs.

Most the attackers use automated password “grinders” that attempt to log on using either Terminal Services or Windows Networking (Server Message Block, or SMB). The logon attempt in Figure 4-13 is actually an Internet Information Services logon attempt, which I know only because the host does not respond on either Terminal Services or SMB across the Internet.

The automated password grinders will typically try common user names, such as Administrator, with a dictionary of passwords. Shockingly, they must be successful enough with that approach to make it worthwhile to continue. Many people argue that you should rename the Administrator account to fool attackers, and some even say to create a decoy account called Administrator. This has absolutely no effect whatsoever. The error message is the same whether an account does not exist with the name Administrator or whether the attacker gets the password wrong. Therefore, from the attacker’s perspective, he cannot tell whether you have an account called Administrator. He can only tell that he did not get in. You can assure yourself that he will not get in simply by setting a reasonably strong password. For example, if the password is 15 characters long, and seemingly random (meaning that it seems random from the attacker’s point of view) the attacker will have to try 542,086,379,860,909,058,354,552,242,176, or so, times before he succeeds. More than likely he will move on before he succeeds in guessing that password.

Leaving Your Passwords Blank

As with all versions of Windows since Windows XP, user accounts with blank passwords cannot log on from the network in Windows Server 2008. This is actually a genius design, which can be used to great effect for the local Administrator account.

In a typical datacenter the servers are locked inside racks. In many cases, not everyone has access to every rack. Only those personnel who need to get into particular servers can get into those racks. The racks themselves are in locked rooms that require badge and PIN access. In that situation, are your servers physically secured? More than likely you would say yes. If so, why not leave the password blank for the built-in Administrator account? The only people that can use it are the ones that get past the badge scanner, have the PIN code to the right room, and the key to the right rack. More than likely, if someone has all of those, he belongs in there and needs to use that account—and has a way to get at the password should he need to. Obviously, he shouldn't use it on a daily basis, but if everything breaks and he needs to log on as the built-in Administrator, he knows what the password is and can get in very easily. In addition, because accounts with blank passwords are not usable across the network, making the password blank removes what would be a significant security dependency if you used the same password on every server.

Leaving the password blank solves one of the huge problems in network security: how do you keep the admin account from having the same password on every server in the network? It's very difficult to argue that leaving the password blank compromises security in any way at all—when you have adequate physical security. Unfortunately, it is probably going to be far more difficult to convince an ill-informed security auditor that leaving the password blank is more secure than setting the same 8-character password he requires on every single server. If you promise to try though, I'll do my part.

Using the Captured Information

Assuming the bad guy has captured something, how does he go about using it? If he has captured a plaintext password, the answer is relatively straightforward. He just needs to find somewhere to type it in. However, if he has captured a challenge-response sequence, or a password hash, the problem is slightly more complicated.

Cracking Passwords

The most common attack is to crack the password. By "crack" in this case, we generally mean that the attacker creates a password hash or a challenge-response sequence based on some trial password and compares it to the hash or response that he captured. If the test succeeds, the trial password is the right password.

As you have seen earlier in this chapter, several additional computations are involved in computing a challenge-response sequence as opposed to computing a straight hash. It stands to reason, therefore, that cracking a captured challenge response sequence takes significantly longer than simply cracking a password hash. On commonly available hardware today you could compute anywhere from 3 million to 10 million hashes per second to try with, while you could compute only a third as many challenge-response pairs. If the bad guy only has the password verifier, he will be able to compute only 10 per second, rendering them effectively uncrackable unless the password is exceptionally weak.

Several approaches to cracking passwords speed up the process. An attacker can try with a dictionary of common words, or common passwords, such as the lists in (Burnett, 2005). The attacker can also try a brute-force attack using all possible passwords of some given character set. The character set can be greatly trimmed. My own research has shown that 80 percent of the characters used in passwords are chosen from a set of only 32 characters. Finally, the attacker can try a hybrid approach in which the test password is based on some dictionary with characters permuted. For example, the attacker may try common substitutions, such as using "!" or "1" instead of "i", "@" instead of "a" or "at", "3" instead of "e" and so on.

Pre-computed Hash Attacks

Pre-computed hash attacks are very simple in concept. The first common use of them was in Gerald Quakenbush's Password Appraiser tool from the late 1990s. The tool shipped with several CDs full of password hashes. Several years later, Cedric Tissieres and Philippe Oechslin developed Ophcrack, which cracked LM hashes using pre-computed hashes, but used a time-memory tradeoff to reduce the amount of storage space required to hold the hashes. Rather than storing all the hashes, they stored only a portion of them along with all the passwords that created that hash. At run time the cracker would simply look up which set of passwords possibly matched the hash it needed to crack, compute the hashes for all the options, and compare them to the hash. This was significantly slower than Password Appraiser, but many orders of magnitude faster than brute-force cracking. Zhu Shuanglei implemented the same technique in the immensely popular Rainbow Crack tool, which can crack almost any hash out there. Pre-computed hash attacks are often referred to as Rainbow Cracks or Rainbow Table Attacks after that tool.

Pre-computed hash attacks have created immense media buzz, and many, many people, and many security "experts" have opined about how bad they are and how they work only because Windows is flawed and how Windows should be fixed to prevent them. Typically these claims are accompanied by statements about how (of course) other operating systems had the foresight to protect against these attacks. These characterizations are gross simplifications that fail to account properly for either history or reality.

First, Windows is not flawed in that it does not take into account pre-computed hash attacks in its design. It is true that use of a salt in the password-hashing mechanism would combat pre-computed hash attacks. However, it simply was not (and still is not) an interesting threat to protect against. Furthermore, nobody should be lured into thinking that the designers of competing operating systems had the foresight to protect against these attacks. Salts were added to protect against the fact that the password file was world-readable. Pre-computed hash attacks were not relevant when those platforms were designed. Keeping gigabytes, or even terabytes, of password hashes was not particularly interesting when the computer had 16KB of core memory and a tape drive.

Second, it makes no sense whatsoever to start salting Windows password hashes to protect against pre-computed hash attacks. Consider how the authentication protocols work. If you change the hashing mechanisms, you must also introduce a new authentication protocol because the old ones rely on the old hashes. The last time a new authentication protocol was actually retired was in Windows Vista, when LM was retired. That took 13 years from the introduction of its replacement. Changing the hashing mechanism to only add a salt would certainly stop pre-computed hash attacks. However, it would take 13 years or so before the old NT hashes were gone. Furthermore, because password hashes are plaintext equivalent, with or without a salt, it would not solve the real problem.

Pass-the-Hash Attacks

Password hashes are plaintext equivalent. This should be eminently clear by now. The secret used by the server to verify the client's identity is the same secret the client uses to prove its identity. If a criminal manages to capture that secret, he can simply use it to prove his identity, *without any knowledge of the password used to create that secret.*

This is a crucial point. If we can accept the fact that password hashes are plaintext equivalent the way we think about things changes. First, we can immediately see why replacing the current NT hashes with salted ones is meaningless, because the salted ones are also plaintext equivalent. Second, we can also see that the core problem is not password hashes, but bad guys with access to them when standard challenge-response protocols are used. The only real technical solution is to move away from challenge-response protocols to public key protocols. However, this requires a substantial change to all platforms and is unlikely to happen any time soon.

Therefore, the real solution is to stop bad guys from getting at password hashes. To do that we need to minimize the exposure of password hashes and we need to ensure that we adequately protect our authentication servers. Chapter 14 discusses these topics in depth.

Protecting Your Passwords

Every one of the attacks we have discussed so far can be mitigated by either using better passwords, or managing and operating your network more securely. Chapter 14 goes into depth about how to manage and operate the network more securely. Obviously, because password hashes are plaintext equivalent, using strong passwords will not mitigate all the attacks outlined so far. However, it will have a significant impact on many of them.

What constitutes a strong password? The answer is: a long password! No single factor is more important than length when it comes to password strength. Table 4-3 shows how long a password composed from *n* characters chosen randomly from a set of 32 characters resists both guessing and cracking attacks.

The password resilience data presented in Tables 4-3 and 4-4 are based on a theoretical attacker than can guess 600 passwords per second or crack 7.5 million passwords per second. These numbers are significantly greater than what can be achieved today both with respect to password guessing and cracking captured challenge-response pairs.

Table 4-3 Password Attack Resilience for 32-Character Character Sets

Length	Guessing Resilience in Days	Cracking Resilience In Days
6	10	0
7	331	0
8	10,605	1
9	339,355	27
10	10,859,374	869
11	347,499,971	27,800
12	11,119,999,080	889,600
13	355,839,970,558	28,467,198
14	11,386,879,057,845	910,950,325

As you can tell from Table 4-3, the strength of the password goes up dramatically the longer it gets. A 14-character password composed of randomly chosen symbols from a known 32-character character set resists guessing for more than 31 billion (!) years. Even an 8-character password would be impossible to guess in a reasonable time frame as long as the attacker cannot rely on heuristics. The 14-character password resists a cracking attack for 2.5 million years, but of course it is still plaintext equivalent, so resistance to cracking is really only relevant in the case of a captured challenge-response sequence.

The question, however, that many want answered is how important the character set is in password strength. Obviously, the larger the character set the attacker has to contend with, the stronger the password is. However, the effect is nowhere near as drastic as length. Table 4-4 shows the same data as Table 4-3, but for passwords composed using a character set consisting of 95 characters.

Table 4-4 Password Attack Resilience for 95-Character Character Sets

Length	Guessing Resilience in Days	Cracking Resilience In Days
6	7,090	1
7	673,551	54
8	63,987,310	5,119
9	6,078,794,461	486,304
10	577,485,473,802	46,198,838
11	54,861,120,011,233	4,388,889,601
12	5,211,806,401,067,100	416,944,512,085
13	495,121,608,101,375,000	39,609,728,648,110
14	47,036,552,769,630,600,000	3,762,924,221,570,450

As Table 4-4 shows, the passwords based on the set of 95 characters are certainly stronger than ones based only on 32. However, a 6-character password from the 95-character set is weaker than an 8-character password from the 32-character set. Realizing that difficulty dealing with complexity is often a human weakness, many people would probably have a simpler time remembering an 8-character password composed of a small set of commonly used characters than a 6-character password composed of characters they hardly ever use. These numbers can be used to develop an appropriate strategy for different people, depending on how they think. However, it is clear from this data that if we can simply get users to use longer passwords, we can solve a lot of password-related problems.

Fundamentally, passwords are a perfectly acceptable, very convenient, comprehensible, and simple-to-implement authentication mechanism. The only flaw in the equation is that people are not good at remembering passwords. If we could only remove the people that use them from the problem, passwords are probably the best way to authenticate to a system. Fortunately, we can.

Managing Passwords

Left to their own devices, people will not pick very good passwords. Yet we need them to pick longer ones to protect us. To reconcile that dilemma, we need to rethink some old concepts that many hold as truth.

Use Other Authenticators

First, a password that the user does not know is better than one the user does know. If you use smart cards and configure the system to require smart card logon, every account will still have a password, but it will be a long and random password. Its hash can still be stolen from any computer that the user logs on to, providing that malware running as the operating system is present on that computer, but the password, for all practical purposes, can never be guessed.

Record Passwords, Safely

For those of us who cannot require smart cards, however, we need to live with the fact that the user must know the password. To help them remember their passwords, the Chinese invented this marvelous technology, called paper, in the second century CE. Users should record their passwords. Currently most organizations have a password policy that requires 8-character passwords, and they must have three different character sets in them. The result? Users pick passwords like "Seattle1", which, if you check it, complies with the policy. "Test1234" complies as well, as does "Password1", "Passw0rd" and "Pa$$word". If you were given the choice, wouldn't you rather have a user carry a little piece of paper in her wallet with the words "Get a skinny tall latte before work!" on it? If a bad guy got hold of that note, the user would know pretty quickly and could take appropriate steps to reset the password (assuming you have told her how to do that), and what exactly would the bad guy do with the note? Which system does password belong to? Is it even a password, or is it a shopping list? A password the user can write down is far easier to manage than one she has to memorize after typing twice. And, for all the other passwords we use every day, you can use an electronic password management tool, such as Password Safe (*http://passwordsafe.sourceforge.net*). Which is really worse: a weak password that the user can remember after typing it twice, or a very strong one that is securely recorded? What exact exposures are we worried about here?

Now imagine that you told your users that they could keep the password on a note until they remembered them, and after that they had to put the notes in the secret disposal bin, or eat them, whichever they preferred. If you do that, your users may even let you set the password policy to require 10 characters and live to tell the tale.

Stop Thinking About Words

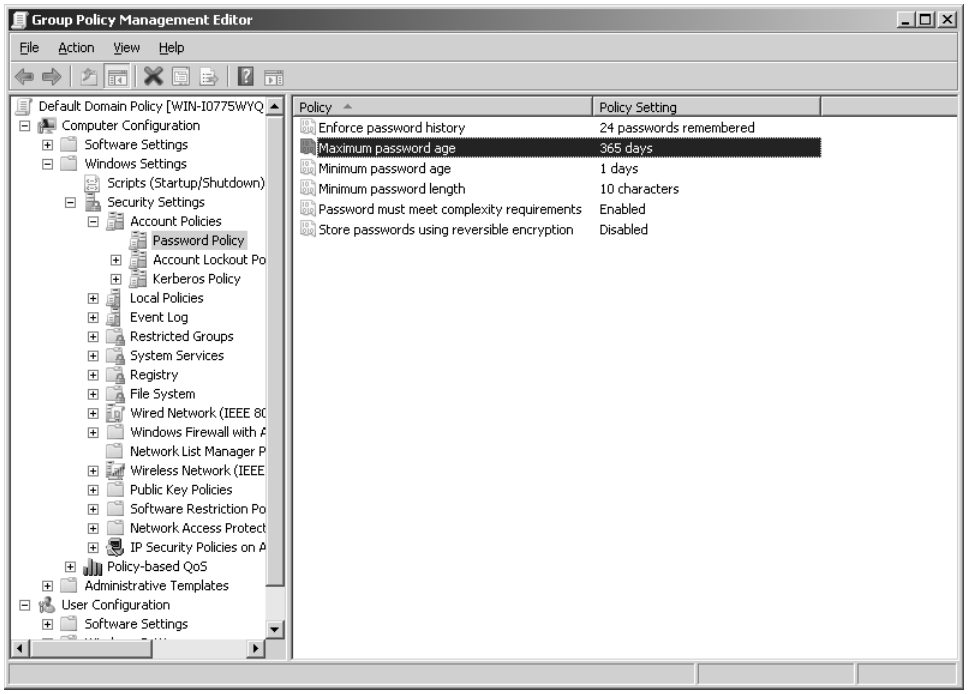
Notice that in the preceding discussion the imaginary password was “Get a skinny tall latte before work!” That is not a password. It is a passphrase. Nothing says that passwords have to be words any more. The very term—password—is wrong. Windows will happily accept up to 127 characters, chosen from the entire keyboard (including the space bar) in a password. Recall also that we concluded earlier in the chapter that the longer the password is the stronger it is. Using a passphrase is the perfect way to add length to your password. Passphrases are long and therefore strong. They are simpler to type and easier to remember than contorted strong passwords, such as hG%'3m.^. Simply put, passphrases just work the way people are used to working already. People are used to thinking about words. I have seen seven-year-old children use passphrases successfully. In addition, a phrase such as the latte one is far, far longer and many orders of magnitude stronger than the contorted strong password. If we assume a worst-case scenario, in which the attacker knows that we use passphrases, knows that this one is seven words long, and even knows the dictionary of words it was composed from, it could still take millions of years to guess—even if the attacker uses an attack tool that permutes words as opposed to characters. The set of possibilities is so many times larger than the set of characters on a keyboard. If you wanted to improve the strength a little, do one of the common substitutions somewhere. For example, replace an “a” with “@”, or an “l” with a “1”, or an “e” with a “3”, or an “o” with a “0”. In our 8-character password we’ll be lucky to get one of those substitutions, merely doubling the possibilities. In the case of the passphrase, we get 12 possible substitutions just with those 4 substitution options, increasing the total search space 4,096 times! Passphrases are immensely powerful as an authenticator.

Set Password Policies

Finally, you should of course have password policies. You need both written organizational policies and technically enforced policies. The written policies are beyond the scope of this book, but should include policies that are realistic—in other words, don’t ban writing passwords down. You should also have an implementation guideline that helps people understand how to pick passwords.

Technical policies should be enforced domain-wide, and also on member computers if you use local accounts on member computers. They should require complexity, long passwords (10 or more characters are highly preferable) and should cycle the passwords regularly. However, tie the policies together logically. If you require 10-character passwords, it is almost certainly acceptable to keep them for 6 months or a year. With 8 characters you should change them every 3-6 months. With anything fewer than 8 characters, you should consider changing passwords monthly.

Policies can be managed with Group Policy (GP). Figure 4-14 shows where in GP the settings are.



Policy	Policy Setting
Enforce password history	24 passwords remembered
Maximum password age	365 days
Minimum password age	1 days
Minimum password length	10 characters
Password must meet complexity requirements	Enabled
Store passwords using reversible encryption	Disabled

Figure 4-14 Password policies can be managed with Group Policy.

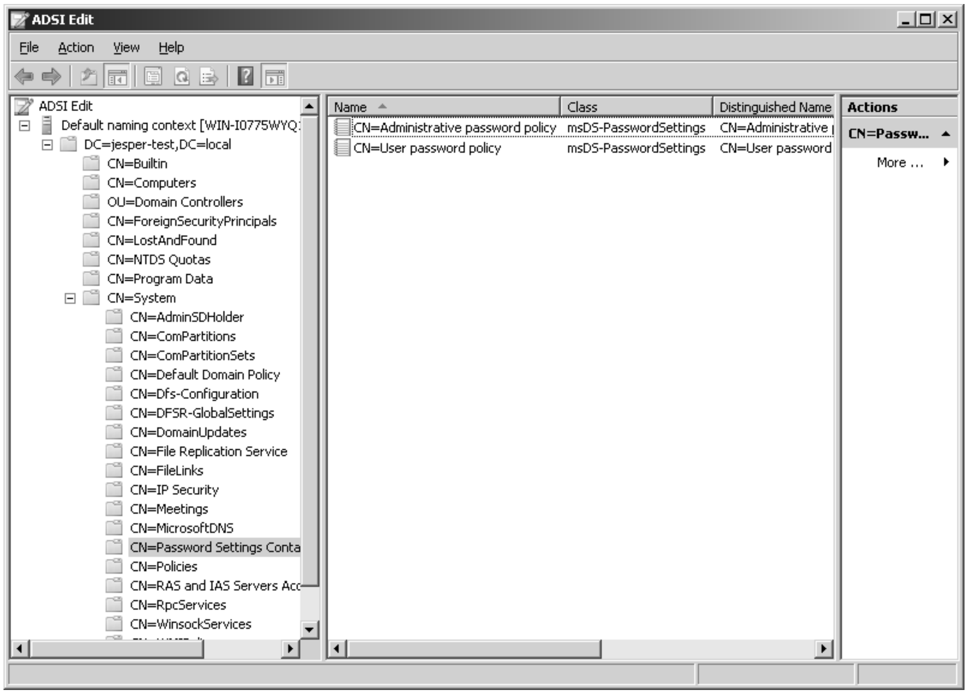
Password policies applied with domain scope apply to domain accounts when the password is set from a workstation. They do not apply to accounts whose passwords are reset in Active Directory Users And Computers (ADUC). This is done so that administrators can reset a user password there without having to create incredibly complex passwords that will be changed in a few minutes anyway. Password policies applied to an organizational unit scope apply to local accounts on all member computers in that OU.

Fine-Grained Password Policies

A persistent request from customers has been the ability to manage password policies so that different users in the domain have different password policies. In Windows Server 2008 this is finally possible, with *fine-grained* password policies. Fine-grained password policies are available in all editions of Windows Server 2008, but only if the domain functional level is Windows Server 2008. In other words, you must first upgrade all your domain controllers to Windows Server 2008 before you can use it.

The primary purpose of fine-grained password policies is to apply stricter settings to privileged accounts and less strict settings to the accounts of the normal users. In other cases you might want to apply a special password policy for accounts whose passwords are synchronized with other data sources.

Fine-grained password policies apply only to user objects, or inetOrgPerson objects if they are used instead of user objects and global security groups. Fine-grained password policies are implemented using a password settings container (PSC) under the System container of the domain. (See Chapter 9, “Designing Active Directory Domain Services for Security,” for more details on Active Directory.) The PSC stores one or more password settings objects PSOs that hold the actual policies. Figure 4-15 shows a PSC with two PSOs.



Name	Class	Distinguished Name
CN=Administrative password policy	msDS-PasswordSettings	CN=Administrative
CN=User password policy	msDS-PasswordSettings	CN=User password

Figure 4-15 This domain uses one password policy for administrators and another for users.

Unfortunately, Microsoft did not provide a very good user experience for managing fine-grained password policies in Windows Server 2008. There is a step-by-step walkthrough available, but the steps are somewhat complicated. To configure a separate password policy for administrators, follow these steps:

1. Run the ADSI Edit tool by running adsiedit.msc.
2. Connect to your domain by right-clicking the ADSI Edit node in the left-hand pane and selecting Connect To. Type in the name of the domain.
3. Expand the domain, expand the DC node, navigate down to CN=System, and select CN=Password Settings Container.
4. Right-click CN=Password Settings Container, select New, and then select Object.
5. Select the *msDS-Password Settings* object, as shown in Figure 4-16.

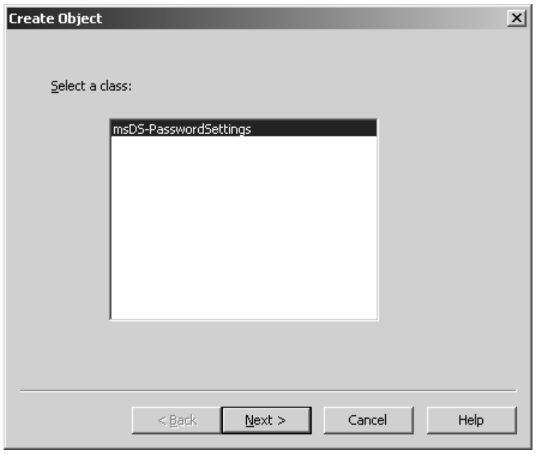


Figure 4-16 To create a fine-grained password policy, you need to create a new *msDS-PasswordSettings* object.

6. Name the new object something memorable, such as **Administrative password policy**.
7. Click Next, and set the precedence value for this object. This value governs which policy takes precedence if two policies apply to the same user. The lowest precedence wins.
8. Walk through the rest of the wizard and set values for all the items. The possible values are listed in Table 4-5.

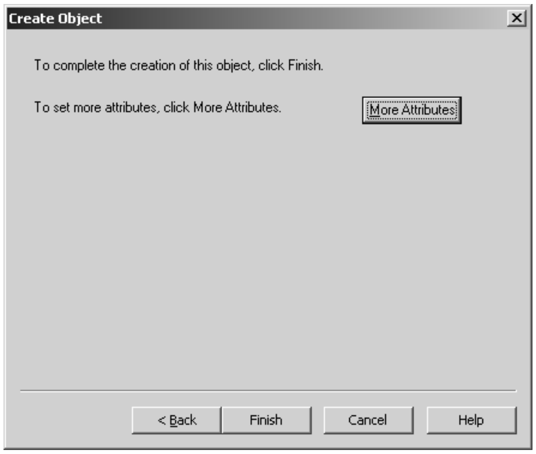
Table 4-5 Fine-Grained Password Policy Values

Attribute name	Description	Acceptable value range
msDS-PasswordSettingsPrecedence	Defines which policy takes precedence if more than one policy applies to a given user. The policy with the lowest precedence wins.	Greater than 0
msDS-PasswordReversibleEncryptionEnabled	Whether passwords are stored with reversible encryption. False means they are not.	FALSE / TRUE
msDS-PasswordHistoryLength	How many passwords the system remembers for a user. Practically speaking, this means the user cannot reuse a password until he has chosen at least this many different ones.	0 through 1024
msDS-PasswordComplexityEnabled	False if password complexity is not required for these user accounts. True if password complexity is required.	FALSE / TRUE
msDS-MinimumPasswordLength	The minimum length for a password. Note that passwords can be up to 255 characters long, but older systems support entering only 127-character passwords.	0 through 255

Table 4-5 Fine-Grained Password Policy Values

Attribute name	Description	Acceptable value range
msDS-MinimumPasswordAge	The minimum age that a password must be before it can be changed again. Setting this to some reasonable value, such as a day or two, ensures that a user cannot cycle through the password history automatically and change the password back to the one that she just had. This value, (and all other time values) is entered in DAYS:HOURS:MINUTES:SECONDS format. Hence 02:00:00:00 is two days.	(None) 00:00:00:00 through msDS-MaximumPasswordAge value
msDS-MaximumPasswordAge	How old a password can be before it must be changed. To have passwords that never expire, use the value *(Never)*. Otherwise, set a date in the standard time value format, such as 180:00:00:00.	(Never) msDS-MinimumPasswordAge value through (Never) msDS-MaximumPasswordAge cannot be set to zero
msDS-LockoutThreshold	How many tries a user gets at the password before it is locked out. To disable account lockout, set this to 0.	0 through 65535
msDS-LockoutObservationWindow	The time interval used to calculate the number of incorrect password tries. If this value is set to 00:00:30:00, for example, the user gets *msDS-LockoutThreshold* tries in 30 minutes and then the counter is reset.	(None) 00:00:00:01 through msDS-LockoutDuration value
msDS-LockoutDuration	How long the account remains locked out before it is automatically unlocked. To require administrative unlock set it to *(Never)*.	(None) (Never) msDS-LockoutObservationWindow value through (Never)

9. After you configure the lockout duration you will see the screen shown in Figure 4-17. At this point you need to configure which users this PSO applies to. To start that process, click More Attributes.

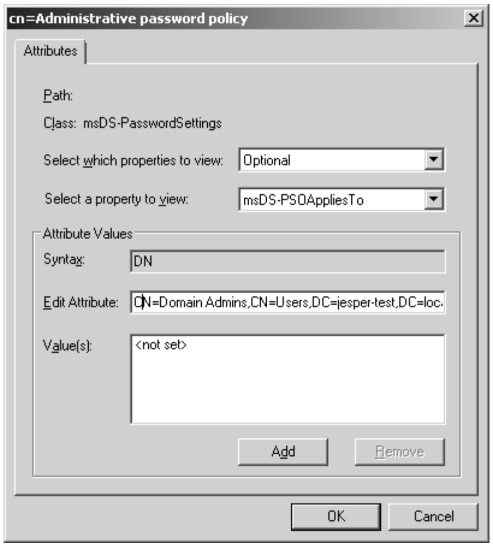


To complete the creation of this object, click Finish.

To set more attributes, click More Attributes.

Figure 4-17 When you get to this screen, configure who the object applies to.

10. From the Select A Property To View drop-down list, select msDS-PSOAppliesTo.
11. Type in the distinguished name (DN) of the user or the global security group you want this policy to apply to. For example, to apply it to the Domain Admins group, use the following syntax, replacing the DC attributes with your domain information:
    CN=Domain Admins,CN=Users,DC=jesper-test,DC=local
    The net result is shown in Figure 4-18.



Path:

Class: msDS-PasswordSettings

Select which properties to view: Optional

Select a property to view: msDS-PSOAppliesTo

Syntax: DN

Edit Attribute: CN=Domain Admins,CN=Users,DC=jesper-test,DC=loc

Value(s): <not set>

Figure 4-18 You use the *msDS-PSOAppliesTo* attribute to apply the policy to a group or a user.

By now you have probably already figured out that Microsoft kind of ran out of time to build good tools to manage fine-grained password polices. Fortunately, there are some options out there. Joeware has a command-line tool available at: *http://www.joeware.net/freetools/tools/psomgr/index.htm*

A GUI tool is available for PowerGUI, based on the PowerShell in Windows Server 2008:

http://powergui.org/entry.jspa?externalID=882&categoryID=46

Another free GUI tool is available from Specopssoft:

http://www.specopssoft.com/wiki/index.php/SpecopsPasswordPolicybasic/SpecopsPasswordPolicybasic/

Precedence and Fine-Grained Password Policies

I mentioned earlier that a precedence value is associated with fine-grained password policies. This value is for resolving conflicts when two PSOs apply to a single user. The policies are not merged, so there must be some way to resolve the conflict. The resolution works as follows:

1. If only one PSO is linked to the user object, that PSO is the resultant PSO. If more than one PSO is linked to the user object, a warning message is logged to the event log and the one with the lowest precedence value is the resultant PSO.
2. If no PSOs are linked to the user object, the system compares the precedence values of all the PSOs linked to groups the user is a member of. The PSO with the lowest precedence value is the resultant PSO.
3. If neither of these methods results in a PSO being the most preferred, the default domain policy applies.

Summary

Additional Resources

1. Burnett, M. Perfect Passwords: Selection, Protection, Authentication. (Syngress, 2005)
2. Johansson, J. M. *Protect Your Windows Network.* (Addison-Wesley, 2005)
3. Johansson, J. M. "The Most Misunderstood Security Setting of All Time." *TechNet Magazine*.
4. Kent, J. "Malaysia car thieves steal finger." http://news.bbc.co.uk/2/hi/asia-pacific/4396831.stm
5. Microsoft Corporation. "Server Core Installation Option of Windows Server 2008 Step-By-Step Guide." http://technet2.microsoft.com/windowsserver2008/en/library/47a23a74-e13c-46de-8d30-ad0afb1eaffc1033.mspx?mfr=true
6. Microsoft Corporation. "Step-by-Step Guide for Fine-Grained Password and Account Lockout Policy Configuration." *http://go.microsoft.com/fwlink/?LinkID=91477*
7. Wagner, M. "The Password Is: Chocolate" http://informationweek.com/story/showArticle.jhtml?articleID=18902123
