Essential .NET Security

Security on the new platform



Introductions

- What is your first name?
- What sort of job do you do?
- What does security mean to you?
- What programming languages are you fluent in?
- Do you have any particular expectations?



Goals of the class

- Learn what threats are out there
- Learn what it takes to design secure systems
- Examine security features of the .NET platform
- Learn how to use them correctly



Module Outline

- Threats and Mitigation
- Conventional Cryptography and Kerberos
- Public Key Cryptography and SSL
- Windows Security 101: Basics
- Windows Security 102: Impersonation and Delegation
- Code Access Security Part 1, Policy
- Code Access Security Part 2, Enforcement
- Securing Web Applications
- Securing Web Services
- Securing System.Runtime.Remoting
- Securing COM+
- Dumb Code: avoid writing code with silly security holes



Logistics

- Hours
- Food
- Facilities
- Materials

Threats and Mitigation



Objectives

- What types of threats are out there?
- Ways of mitigating those threats
- A process for designing secure code
- Some guiding principals for writing secure code
- Authentication
- Authorization
- Other security techniques and technologies



The STRIDE Threat Model

- **STRIDE**
  - **S**poofing Identity
  - **T**ampering with data
  - **R**epudiation
  - **I**nformation Disclosure
  - **D**enial of Service
  - **E**levation of Privilege



Spoofing identity

- **Attacker pretends to be someone he is not**
  - there are two flavors of this attack
- **Spoofing client identity**
  - access a server and pretend to be a legitimate user
  - gain access to sensitive data
  - run potentially dangerous queries/processes on the server
  - gain administrative access to the server
- **Spoofing server identity**
  - pretend to be a legitimate server to unsuspecting clients
  - collect sensitive data from clients
  - provide false data to clients
  - often opens the door for other attacks

Mitigating the spoofing threat

- **Strong authentication is the best defense**
  - authentication is a secure process for validating identity
  - clients can prove their identities to servers
  - servers can prove their identities to clients
- **Identity can be proved in several ways**
  - something you have
  - something you know
  - something you are
- **Authenticating over a network requires cryptography**
  - more on this later…



Tampering with data

- **Attackers often gain advantage by tampering with data**
- **Tampering with persistent data**
  - change prices for products they want to buy online
  - modify audit logs to cover their tracks
  - modify password data to gain access to other user accounts
  - corrupt data files to crash, or even take over a server
  - deface web pages
  - add viruses or Trojan horses to files
  - tamper with network topology (routing tables, DNS, etc.)
- **Tampering with network packets**
  - tampering with packets on the wire

Mitigating the tampering threat

- **Protect persistent data**
  - hash codes
  - digital signatures
  - encryption
  - example: NTFS Encrypting File System (EFS)
- **Protect network packets**
  - network authentication protocols usually offer integrity and confidentiality protections via cryptography
  - Kerberos
  - SSL/TLS
  - IPSec

Repudiation

- **Attacker denies an action, and victim cannot prove otherwise**
  - an attacker might:
    - claim he didn't delete a file
    - claim he didn't make a purchase or return
    - claim he didn't receive goods/services



Mitigating the repudiation threat

- **Mitigation techniques are called nonrepudiation**
  - audit actions in the OS and protect the audit logs
  - require receipts as acknowledgement
  - use timestamps
  - digital signatures can help with electronic transactions

Information disclosure

- **Attacker sees data he shouldn't be seeing**
  - local files
  - data traveling between computers
- **Attacker sees information about the system architecture**
  - banners that display software type and version
  - helps the enemy narrow down potential attacks



Mitigating the information disclosure threat

- **Use strong authentication and consistent access control**
- **Encryption might help**
  - NTFS EFS, for example
- **Turn off banners on publicly exposed services**
  - or expose purposely misleading banners
  - obscurity is not security but sometimes it helps
- **Disable tracing and debugging features in production apps**
- **Avoid sending verbose error information to clients**
  - Pipe this information to internal logs instead



Denial of service (DoS)

- **Attacker causes your service to become unavailable**
  - usually associated with services provided over a network
    - syn flood attacks
    - distributed denial of service attacks (DDoS)
    - amplification attacks such as smurf
    - all designed to consume precious bandwidth!
  - the anonymity of TCP/IP doesn't help
    - DoS attacks are quite troublesome because the attacker can spoof his source ip address randomly

Mitigating the denial of service attack

- **Increase availability and reliability**
  - make sure your system doesn't melt under high loads
  - have a strategy for throttling requests
  - consider clustering
  - buy more bandwidth
- **Filtering and throttling**
  - block incoming ICMP broadcasts (for example)
  - throttle anonymous requests
- **Be a good neighbor**
  - egress filtering (verify source IP addr on outgoing packets)
  - automate virus checking to avoid DDoS zombies

Elevation of privilege

- **Attacker finds a way to gain more privileges on the system**
  - the ultimate goal is to gain administrative privileges
  - most common exploit is the buffer overflow (more on this later)
  - bugs in the operating system itself can allow this



Mitigating the elevation of privilege threat

- **Produce and consume only quality, robust code**
  - avoid common security errors like buffer overflows
  - fear user input
    - more on this later
  - run code with only the privileges it really needs
    - known as *Principal of Least Privilege*
  - eliminate dead code
    - code paths that are never used
    - features of third party software that you don't need
  - keep up to date with the latest operating system patches
    - HFNETCHK.EXE + Baseline Security Analyzer
    - http://www.microsoft.com/security

Summary of STRIDE threats and mitigation

- **STRIDE**
  - **S**poofing Identity
    - strong authentication
  - **T**ampering with data
    - hash codes, digital signatures, encryption
  - **R**epudiation
    - audit logs, receipts, digital signatures, timestamps
  - **I**nformation Disclosure
    - strong authentication, access control, encryption, obscurity
  - **D**enial of Service
    - increase availability, reliability, and be a good neighbor
  - **E**levation of Privilege
    - robust code, least privilege, OS patches

The three components of a secure system

- **Just as with physical security, we need all three**
  - protection
  - detection
  - reaction
- **You don't need unbreakable protection**
  - you really can't achieve this anyway
  - many developers throw up their hands if they can't design a perfect solution (it feels frustrating)
- **Design detection and reaction into your systems**
  - protection then becomes a way to slow down the attacker
  - once detected, an attack can be halted by a sysadmin
  - diagnose and patch the problem quickly

A process for developing secure apps

- **Security should be an integral part of the design process**
  - write down your security goals
  - examine the system architecture
  - determine the threats using STRIDE
  - prioritize threats
    - risk = (potential damage) **x** (likelihood of success)
  - choose a response
    - accept the risk as is
    - warn the user (transfer the risk)
    - remove the feature (remove the risk)
    - fix the problem (mitigate the risk)
  - revisit your security strategy with each iteration!

General principals to live by

- Security is a feature
- Use least privilege
- Layer your defenses
- Pay attention to failure modes
- Prefer secure defaults
- Cryptography doesn’t ensure security
- Firewalls don’t ensure security



Security is a feature

- **Security is a crosscutting feature**
  - Similar to performance
- **Impossible to bolt on security at the end of a project**
  - Requires constant attention and iteration
- **Be sure you have a security feature team**
- **Need to convince management you need security?**
  - It’s amazing what a demonstration can do



Use least privilege

- **Run your code with only the privileges it requires**
  - most services don't need to run as SYSTEM
  - most desktop apps don't need admin privileges
- **Use WinXP and .NET Server's built in low-privilege accounts**
  - NT Authority \ LocalService
  - NT Authority \ NetworkService
- **Don't be lazy**
  - open kernel objects for only the permissions you really need
  - test your code in a non-administrative environment
  - or go one step further and WRITE your code in a non-administrative environment!

Layer your defenses

- **Don’t assume someone else will save you**
  - Consider your code the last bastion of defense
  - Validate input data



Pay attention to failure modes

- **Developers focus on normal paths of execution**
- **Attackers focus on failure modes**

devote at least as much time to design, code, and test error handling paths as you do for normal paths of execution

Prefer secure defaults

- **Don’t ship code with every possible feature enabled**
  - This just broadens the attack surface area
- **Case in point**
  - IIS 5 installed on every workstation by default in Win2k
  - IIS 5 enabled several features that few people needed
  - Those unused features harbored bugs that went unnoticed until hackers found and exploited them



Cryptography doesn't ensure security

- **So what if you're using 128-bit encryption?**
  - Are the keys generated from low entropy passwords?
  - Where are the keys stored?
  - How are the keys exchanged?
  - How much data is encrypted with a single key?
  - Did you realize that encryption does not ensure integrity?



Firewalls don’t ensure security

- **Any application exposed through the firewall must be robust**
  - Dumb code in exposed applications leads to compromise
- **How do you manage trust across a firewall?**
  - Do you trust the authority outside the firewall to authenticate external clients?



Books every Windows programmer should own

- **Secrets and Lies**
  - Schneier
- **Hacking Exposed (latest edition)**
  - McClure, et al.
- **Writing Secure Code**
  - Howard & LeBlanc
- **.NET Framework Security**
  - LaMacchia, et al.
- **Programming Windows Security**
  - Brown

Summary

- **Security is a feature, design it as such!**
- **STRIDE helps us categorize threats**
- **Design protection, detection, and reaction into your systems**



Conventional Cryptography and Authentication

Cryptography, passwords, and Kerberos



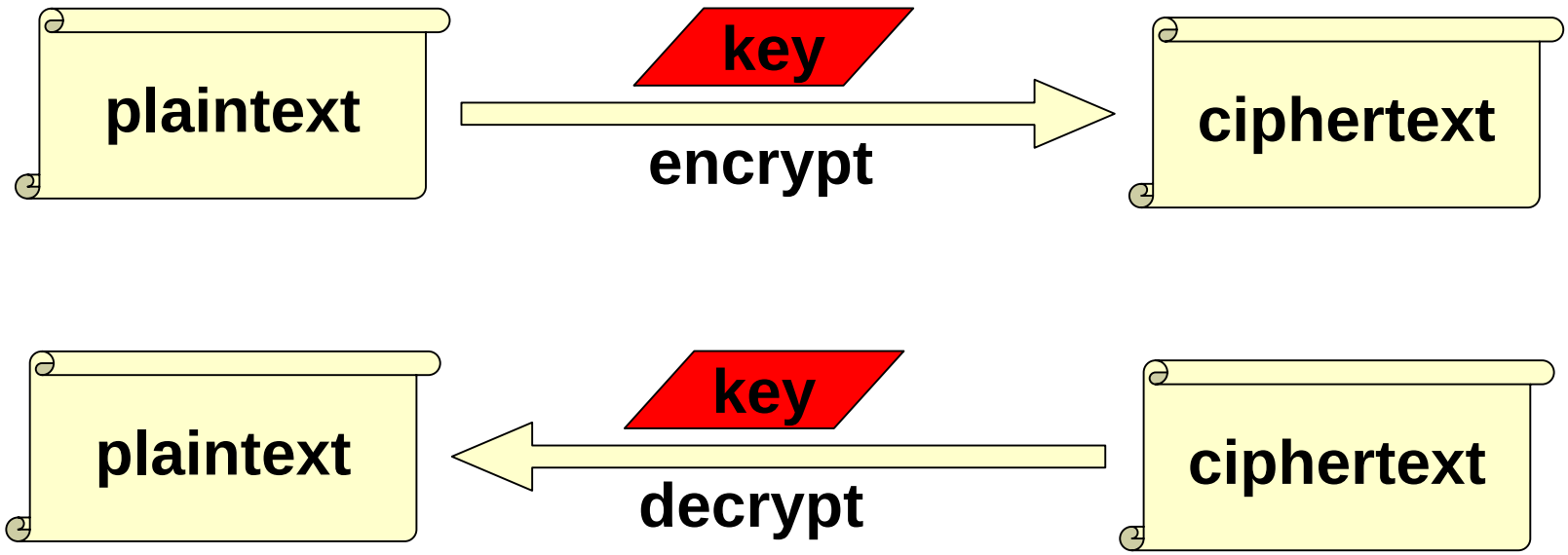
Outline

- Conventional cryptography
- Using passwords as keys
- Conventional crypto from .NET
- Network authentication using passwords
- The Kerberos authentication protocol
- SSPI, the unmanaged interface to Kerberos
- Using Kerberos from managed code



Conventional cryptography

- **Conventional cryptography uses symmetrical algorithms**
  - same key material used to encrypt and decrypt



Uses of cryptography

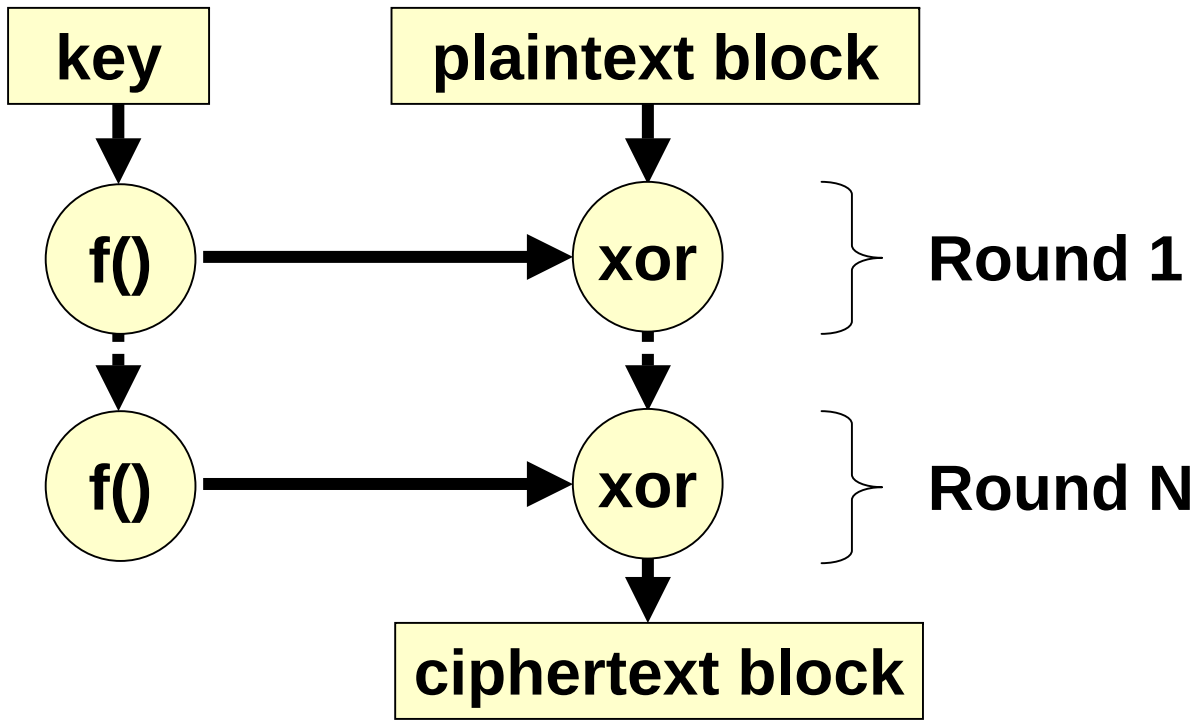
- **Encrypting static data**
  - generate a random key
  - encrypt the data
  - store the key somewhere safe, offline for instance
  - provide key at later date to allow decryption
- **Encrypting network traffic**
  - generate a random key
  - share it secretly with a peer on the network
  - use the shared key to encrypt message before sending
  - use the shared key to decrypt message upon reception
- **Network authentication**
  - Can use cryptography to prove knowledge of a password

Stream Ciphers

- **How a stream cipher works**
  - key is used as a seed for a pseudo-random-number generator
  - run the PRNG continuously to get a key stream
  - XOR each bit of plaintext with corresponding bit in key stream
  - Most common example is RC4
- **Benefits**
  - easy to implement, blazingly fast
  - ciphertext is always exactly the same length as plaintext
- **Drawbacks**
  - incredibly easy to misuse
  - most software that uses stream ciphers has at one time been buggy and insecure
- **System.Security.Cryptography doesn't expose them at all**

Block Ciphers

- **How a block cipher works**
  - input is broken up into fixed size blocks (typically 8 or 16 bytes)
  - transformation f() applied to key, result xor'd into block
  - this is known as a "round" – 16 to 32 rounds is typical



Block Ciphers (cont.)

- **Problem: redundancy**
  - Two blocks of plaintext with the same content produce two blocks of ciphertext that are equivalent
- **Feedback modes were introduced to hide redundant blocks**
  - Electronic Code Book (ECB) means no feedback
  - Cipher Block Chaining (CBC) xors ciphertext from previous block into plaintext for next block
  - other modes available, but ECB and CBC most common
- **Using a feedback mode requires an initialization vector (IV)**
  - random block of data to get the feedback loop started
  - don’t need to keep the IV secret, can send with ciphertext

Block cipher padding

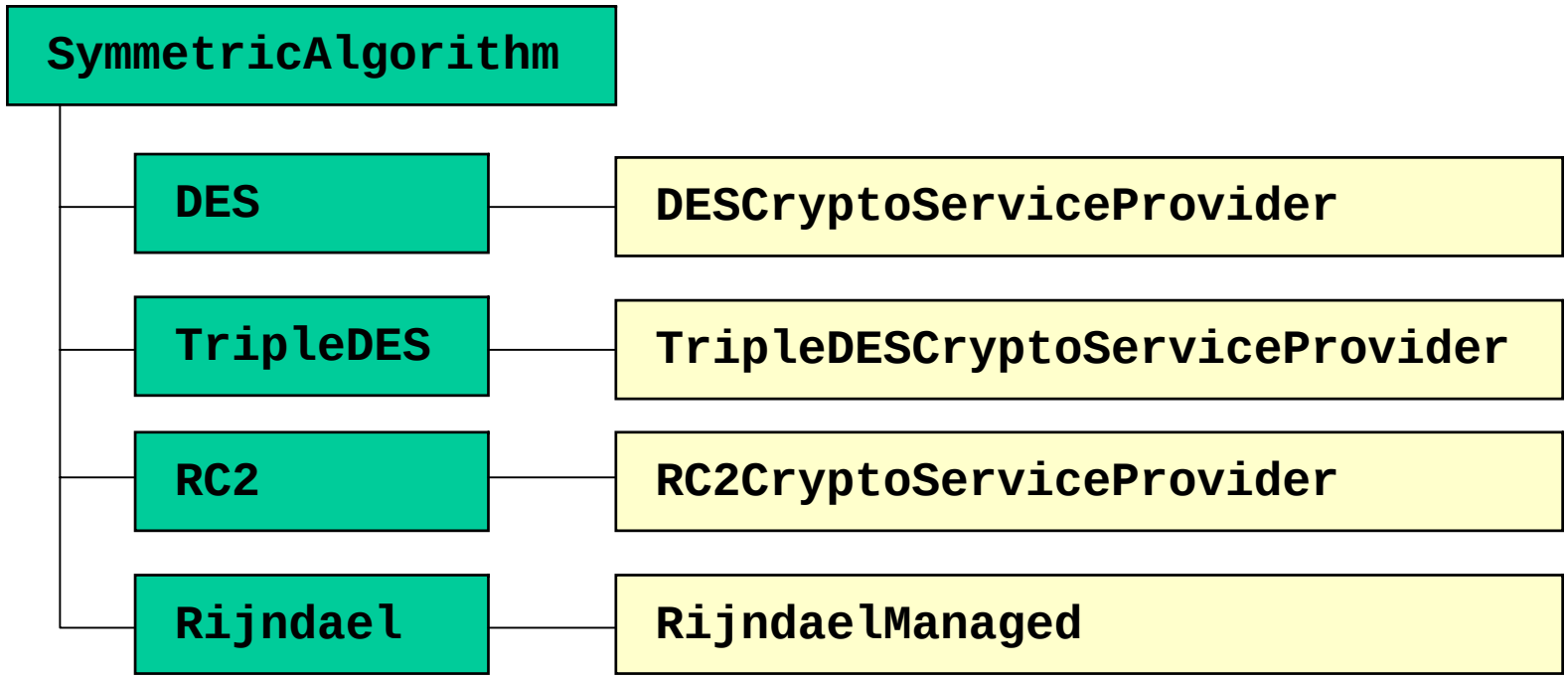
- **Block ciphers need to operate on fixed size blocks**
  - what if plaintext length isn't a multiple of the block size?
  - need to pad the plaintext
- **PKCS deals with these sorts of issues**
  - PKCS = Public Key Cryptography Standards, by RSA Security
- **PKCS#7 specifies a simple way to pad plaintext**
  - if 1 byte of padding is required, pad with the byte 0x01
  - if 2 bytes of padding are required, pad with the byte 0x02
  - and so on…
  - all messages must be padded for this to work
- **The .NET Framework classes automate this**
  - add padding before encryption, remove it after decryption
  - calculate the ciphertext length with padding in mind!
  - C = P + BS – (P mod BS)

Encrypting data in .NET

- **Setting up**
  - choose an algorithm and implementation
  - choose a feedback mode
  - choose a padding mode
  - generate an initialization vector (IV)
  - choose a key
- **Encrypting**
  - record the initialization vector for use during decryption
  - create a CryptoStream object based on your key
  - pump data through the stream to encrypt it



Algorithms and implementations in .NET



note that these are all block ciphers

Example: encrypting a file

```
static void _encrypt(FileStream s, FileStream d) {
  SymmetricAlgorithm alg = new RijndaelManaged();
  alg.Mode    = CipherMode.CBC;
  alg.Padding = PaddingMode.PKCS7;
  alg.GenerateIV();
  _writeIV(alg, d); // writes alg.IV to the stream

  // this example uses a password as a key
  // more on this later...
  alg.Key = _keyFromPassword(_getPassword());

  Stream cryptOutput = new CryptoStream(d,
                                        alg.CreateEncryptor(),
                                        CryptoStreamMode.Write);
  _pump(s, cryptOutput);
  cryptOutput.Close();
}
```

Decrypting data in .NET

- **Setting up**
  - choose the same algorithm you used to encrypt (duh!)
  - choose the same feedback mode
  - choose the same padding mode
  - retrieve the initialization vector (IV) used during encryption
  - retrieve the key
- **Decrypting**
  - create a CryptoStream object based on your key
  - pump data through the stream to decrypt it
  - close the CryptoStream immediately when done decrypting
  - this causes it to eat any leftover padding from the input stream

Example: decrypting a file

```
static void _decrypt(FileStream s, FileStream d) {
  SymmetricAlgorithm alg = new RijndaelManaged();
  alg.Mode    = CipherMode.CBC;
  alg.Padding = PaddingMode.PKCS7;
  _readIV(alg, s);

  alg.Key = _keyFromPassword(_getPassword());

  Stream cryptOutput = new CryptoStream(d,
                                        alg.CreateDecryptor(),
                                        CryptoStreamMode.Write);
  _pump(s, cryptOutput);
  cryptOutput.Close();
}
```




Key Length

- **Some algorithms have hardcoded key lengths**
  - DES is limited to a 56-bit key, TripleDES is 56 * 3
- **Other algorithms offer a variety of key lengths**
  - RC2, Rijndael (AES), for example
- **The longer the key, the harder it is to break the encryption**
  - assuming the key comes from a good random source
- **The chart on the next slide is from 1995, but gives you a visceral look at how key length affects security**

Brute force encryption breaking

Attacker	Budget	Hardware	Time & Cost		Secure Key Length	
			40 bit	56 bit	1995	2015
Pedestrian	Tiny	PC	1 week	Infeasible	45	59
Hacker	$400	FPGA	5 hours $0.08	38 years $5,000	50	64
Small Business	$10K	FPGA	12 min $0.08	556 days $5,000	55	69
Corporate Department	$300K	FPGA	24s $0.08	19 days $5,000	60	74
		ASIC	0.18s $0.001	3 hr $38		
Big Company	$10M	FPGA	0.7s $0.08	13 hr $5,000	70	84
		ASIC	0.005s $0.001	6 min $38		
Intelligence Agency	$300M	ASIC	0.0002s $0.001	12s $38	75	89

Passwords as keys

- **Passwords or phrases can be turned into conventional keys**
  - variable length passphrase converted into a fixed length key
  - hash of password produces fixed length key
- **Passwords are often long term secrets**
  - we limit their use as much as possible to avoid compromise
  - shorter term secrets known as session keys are often used
  - long term secrets such as passwords are usually used to help exchange short term secrets such as session keys
- **Use short term keys to encrypt data whenever possible**
  - attacker has less ciphertext to work with to break the key

Turning a password into a key

```
static byte[] _keyFromPassword(string s) {
  // encode string into a byte array
  MemoryStream media  = new MemoryStream();
  BinaryWriter writer = new BinaryWriter(media);
  writer.Write(s);
  writer.Flush();
  media.Seek(0, SeekOrigin.Begin);

  // compute the 20 byte SHA hash
  byte[] shaHash = SHA1.Create().ComputeHash(media);

  // take the first 16 bytes for use as a 128 bit key
  byte[] key = new byte[16];
  Array.Copy(shaHash, 0, key, 0, 16);

  return key;
}
```

Reality check – password entropy

- **The code you just saw has a huge flaw**
  - it accepts passwords of arbitrary quality
  - it always produces a 128-bit key
  - it gives the illusion that you have the protection of a 128-bit key
- **A password that truly has 128-bits of entropy is rare indeed**
  - this requires 20 random characters from the following:
    - upper and lower case alpha (A-Z, a-z)
    - digits (0-9)
    - punctuation
- **Be sure to give the user feedback on password quality!**
  - consider rejecting passwords with too little entropy

Calculating the entropy of a password

```
static double _passwordEntropy(string s) {
  if (0 == s.Length) return 0;

  // first determine the type of characters used
  int permutations = 0;

  // psuedo-code for brevity
  if (usesLowerCaseAlpha) permutations += 26;
  if (usesUpperCaseAlpha) permutations += 26;
  if (usesNumerics)       permutations += 10;
  if (usesPunctuation)    permutations += 32;

  double passwordEntropy =
     Math.Log10(Math.Pow(permutations, s.Length)) /
     Math.Log10(2);

  return passwordEntropy;
}
```

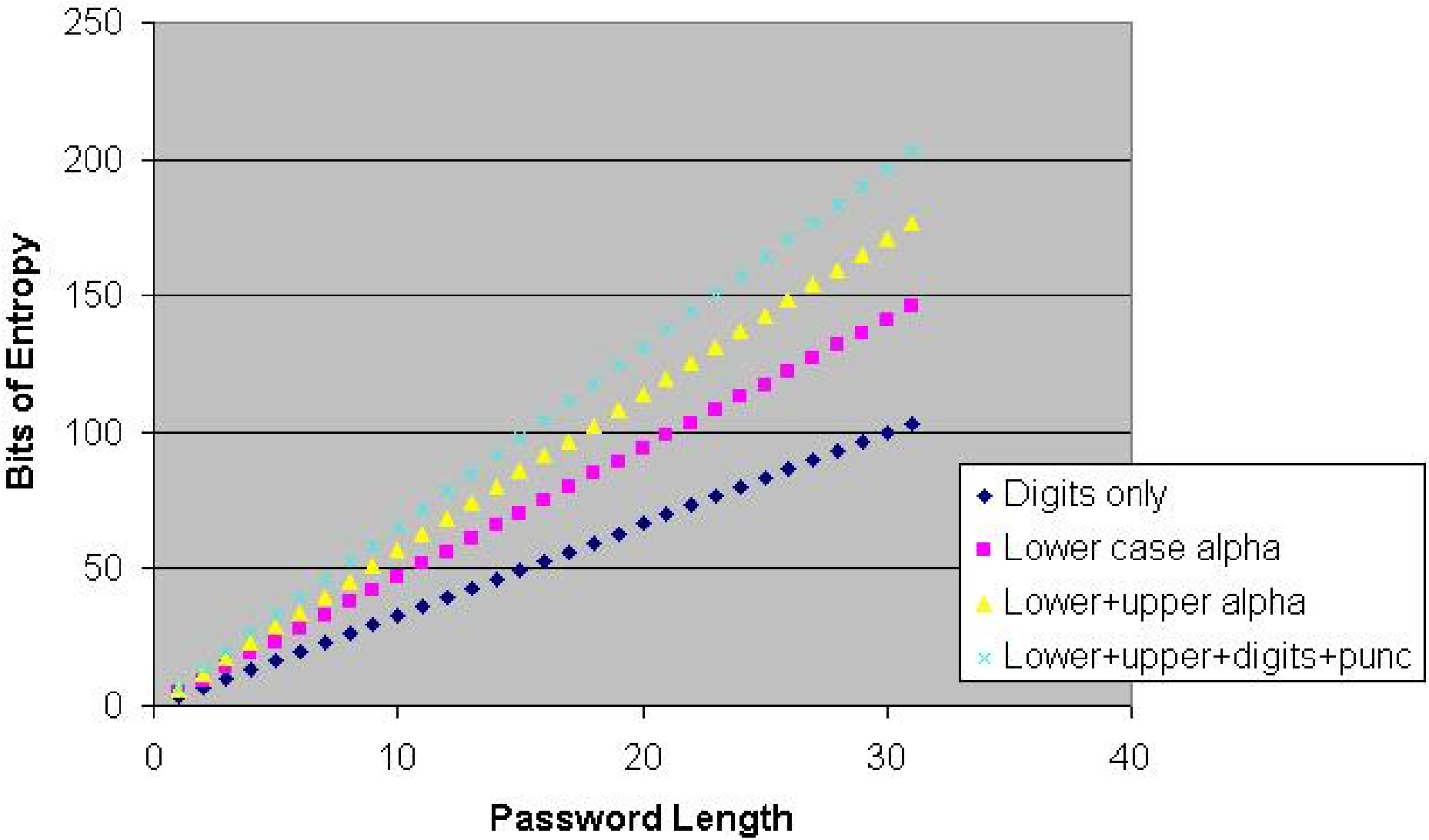
Note this calculation is totally bogus if password contains words found in a dictionary!

Estimating the required length of a password

```
static double _minPasswordLength(double permutations,
                                 double minEntropy) {
  return Math.Log(Math.Pow(10, requiredEntropy *
                               Math.Log10(2)),
                  permutations);
}
```

char set	permutations
lower case alpha	26
+ upper case alpha	52
+ digits	62
+ punctuation	94

Password lengths



Choosing an algorithm

- **Narrow down your choices**
  - 1) use well-known algorithms. Avoid obscure ones.
  - 2) use an algorithm that supports your required key length
  - 3) prefer a block cipher to a stream cipher
  - 4) pick an algorithm that performs well on your platform
    - some algorithms perform better in hardware (DES)
    - some perform well in software (RC2, IDEA)
- **Sources for information on Crypto Algorithms**
  - Applied Cryptography, 2cnd Ed, by Bruce Schneier
  - Peter Gutmann's crypto tutorial
    - http://www.cs.auckland.ac.nz/~pgut001/tutorial/index.html



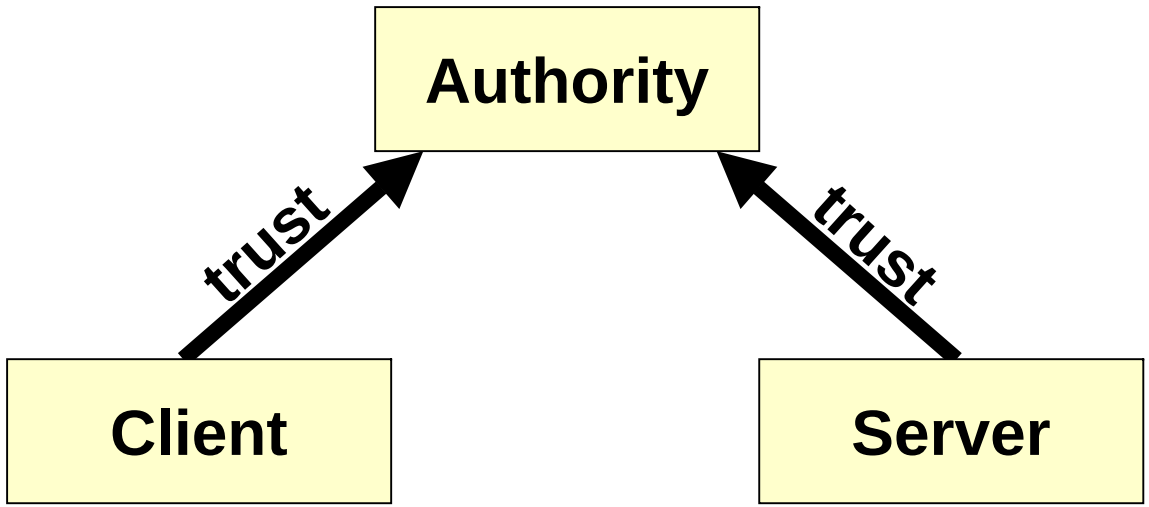
Network authentication using passwords

- **Password only known by two parties**
  - the principal who owns the account
  - the authority who issued the account (i.e., domain controller)
- **To authenticate with the authority**
  - principal must prove knowledge of password to authority
  - this is pretty easy to accomplish
- **To authenticate with some other server**
  - server doesn't know client's password
  - this is more complex
- **Kerberos is a client-server authentication protocol**
  - Helps client and server develop trust in each other's identity
  - Helps client and server discover a session key



The Kerberos authentication protocol

- **Kerberos is based on conventional cryptography**
  - three parties are involved: client, server, authority
  - client and server must have accounts with authority
  - authority “introduces” client and server, helping them develop trust in each other’s identity
  - Only has meaning if client and server trust the authority



Kerberos tickets

- **Authority generates a session key**
  - this session key is for the client and server to share
- **Authority whispers session key to client**
  - session key is sent to client encrypted with client's master key
- **Authority gives the client a "ticket" to send to the server**
- **Ticket contains (among other things)**
  - client's name
  - copy of the session key
- **Ticket is encrypted with server's master key**
- **If server decrypts ticket successfully**
  - server trusts that contents were generated by the authority
  - server obtains a copy of the session key



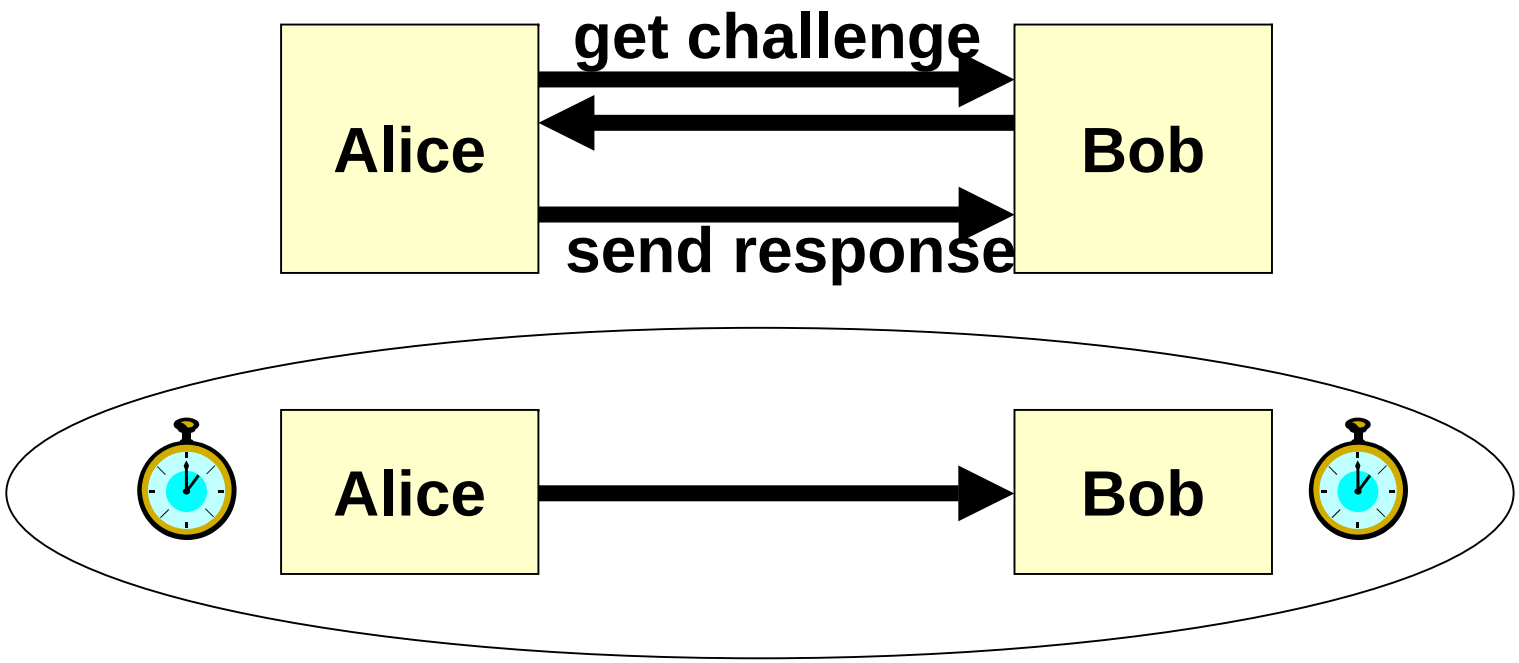
Using a ticket to authenticate the client

- **Is a ticket enough to prove the client's identity?**
  - ticket was sent to the client in the clear
  - ticket is sent to the server in the clear
  - thus anyone could have a copy of the ticket
- **Client needs to prove knowledge of the session key**
  - remember, the authority "whispered" this to the client
  - only someone with the client's master key could have decrypted it
- **How do we prove knowledge of a key without disclosing it?**



Proving knowledge of a key

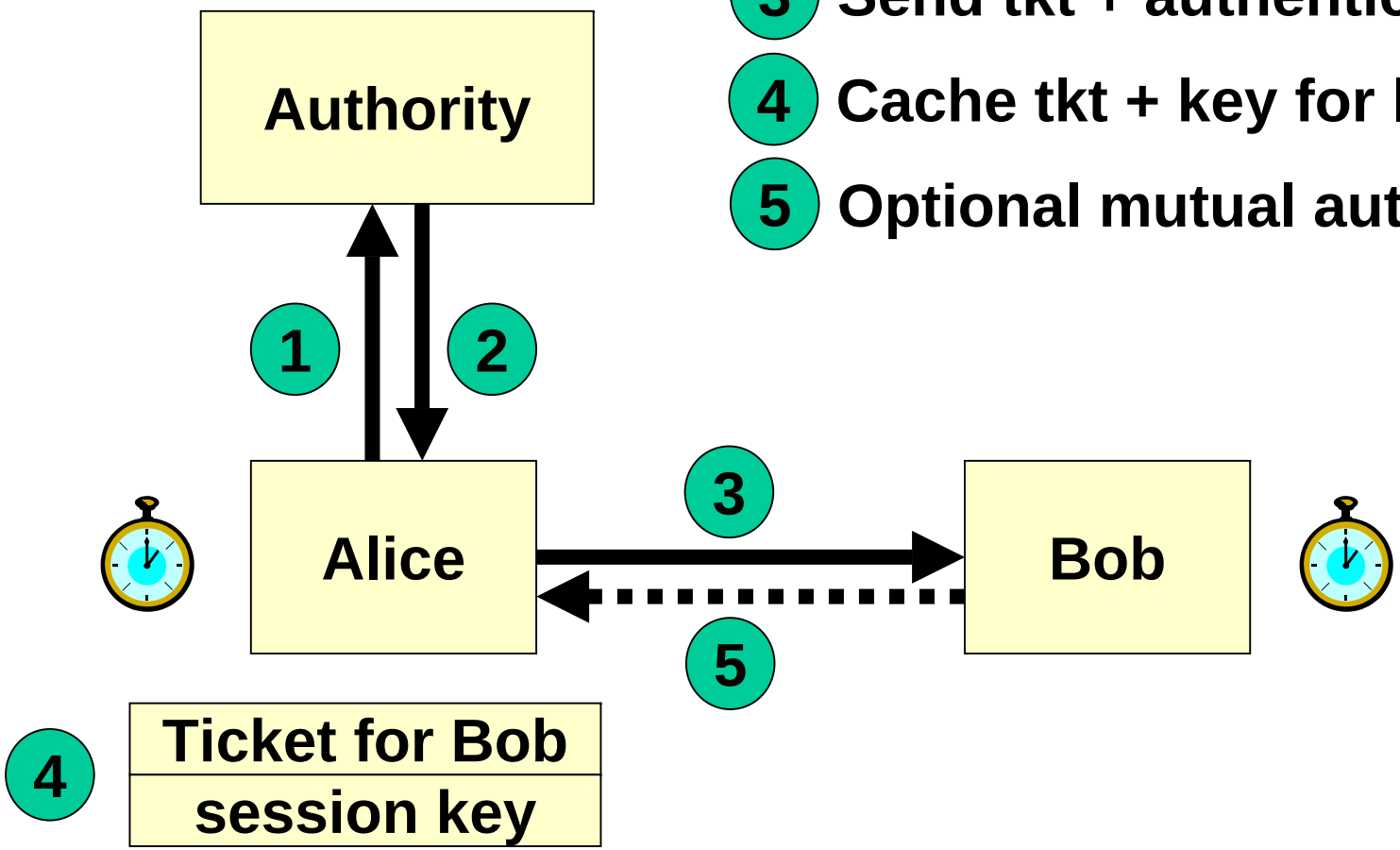
- **Two common ways to prove knowledge of a key**
  - encrypt a random challenge obtained from the server
  - encrypt a timestamp
  - in both cases, the proof is called an “authenticator”



Kerberos uses this method

Ticket usage

1. Request tkt for Bob
2. Issue tkt, whisper key
3. Send tkt + authenticator
4. Cache tkt + key for later use
5. Optional mutual authn step



Kerberos

- **Clients shouldn't need to keep their master key in memory**
  - required to decrypt session keys coming from authority
- **Kerberos introduced the "Ticket Granting Ticket" to fix this**
  - at login, when client password is available, client uses it to request a ticket + session key for the authority itself
  - this ticket is known as the TGT, or Ticket Granting Ticket
  - new session keys are "whispered" to client using session key instead of master key
- **Practically speaking**
  - this means one extra round trip to authority at login time
  - Windows still caches master key to allow access to NT4 boxes



Kerberos credentials

- **Windows caches credentials for each client logon session**
  - credentials are a ticket plus the session key for that ticket
- **Windows gathers group SIDs during authentication**
  - hence group membership information is cached inside tickets
- **Logging out clears your ticket cache**
  - good idea to log out and log back in if groups change, or
  - LsaCallAuthenticationPackage can clear the cache manually



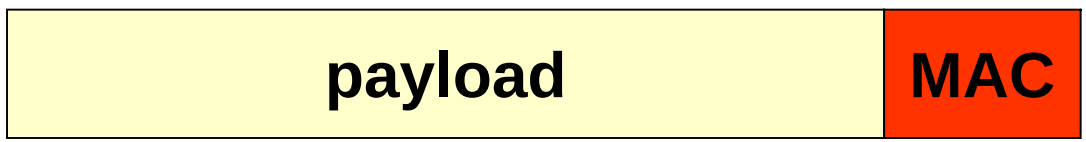
After authentication

- **Client and server have developed trust in the other's identity**
- **Session key should now be used to protect data sent between client and server**
  - can integrity protect and authenticate the data
  - can encrypt the data as well
  - if you don't protect the data stream, an attacker could be modifying or replaying packets, or simply injecting his own!
- **Encryption != Integrity Protection**



Integrity protection and the MAC

- **MAC == Message Authentication Code**
  - simple idea
  - hash that can be computed only if you know the session key
- **Forming a MAC**
  - include the session key in the payload when hashing*, or
  - encrypt a hash of the payload with the session key
- **MAC protecting messages accomplishes two things**
  - integrity protects the data
  - authenticates the data
- **Still need to protect against replays**
  - sequence numbers can help with this



Where is Kerberos used?

- **Domain controller + Active Directory == Kerberos authority**
- **Lots of subsystems use Kerberos in Windows**
- **Using Kerberos over the Internet is tricky**
  - clients usually can't reach the Kerberos authority on port 88
  - next generation of MS Passport will use Kerberos, likely tunneling requests to the authority over port 80
- **Using Kerberos on an intranet is much more common**
  - authenticated RPC and DCOM calls
  - file and printer sharing, remote administration tools
  - IPSEC
- **On Windows, SSPI can be used to "Kerberize" an application**
  - SSPI = Security Support Provider Interface

Security Support Provider Interface (SSPI)

- **SSPI is a generic interface to authentication providers**
  - Kerberos
  - NTLM (for local accounts and NT4 boxes)
  - SSL / TLS
- **Designed for use by C / C++ programmers**
  - generic support for any authentication protocol
  - being generic makes it tough to use for most people
- **.NET Framework currently has no official wrappers for SSPI**
  - but a sample written by the .NET remoting team provides this
  - http://msdn.microsoft.com/library/en-us/dndotnet/html/remsspi.asp



Using SSPI from .NET (initial authentication)

```
// client code to obtain ticket for server
ClientCredential cred =
     new ClientCredential(Credential.Package.Kerberos);

ClientContext ctx = new ClientContext(cred,
     @"MyDomain\MyServerPrincipal",
     ClientContext.ContextAttributeFlags.None);

byte[] authnData = ctx.Token; // ticket + authenticator
```

```
// server code to process incoming ticket
ServerCredential cred =
     new ServerCredential(Credential.Package.Kerberos);
ServerContext ctx = new ServerContext(cred, authnData);

// server code to look at client's name
WindowsIdentity clientIdentity =
    new WindowsIdentity(ctx.ClientSecurityToken);

string clientName = clientIdentity.Name;
```

Using SSPI from .NET (protecting the channel)

```
// client code to MAC protect and encrypt a message
byte[] ciphertext = ctx.EncryptMessage(plaintext);
```

```
// server code to decrypt and verify the MAC
try {
    byte[] plaintext = ctx.DecryptMessage(ciphertext);
}
catch {
    // message integrity violation!
}
```

Summary

- Conventional cryptography is all about secret keys
- Passwords are commonly hashed to form keys
- The .NET Framework has great support for cryptography
- Kerberos is an authentication protocol
- Authn protocols help exchange keys and validate identity
- SSPI is an unmanaged interface to Kerberos
- Currently no support for raw Kerberos in .NET Framework



Public Key Cryptography and Authentication

Cryptography, certificates, and SSL



Outline

- Public key cryptography
- Encryption
- Signatures
- Certificates
- Using public keys from the .NET Framework
- CAPICOM examples
- SSL and TLS



Origins of public key cryptography

- **In 1976, Whitfield Diffie and Martin Hellman made a discovery**
  - public key cryptography was born

In real-world terms, it allows you and a friend to shout numbers at each other across a crowded coffeehouse filled with mathematicians so that when you are done, both you and your friend know the same random number, and everyone else in the coffeehouse is completely clueless.

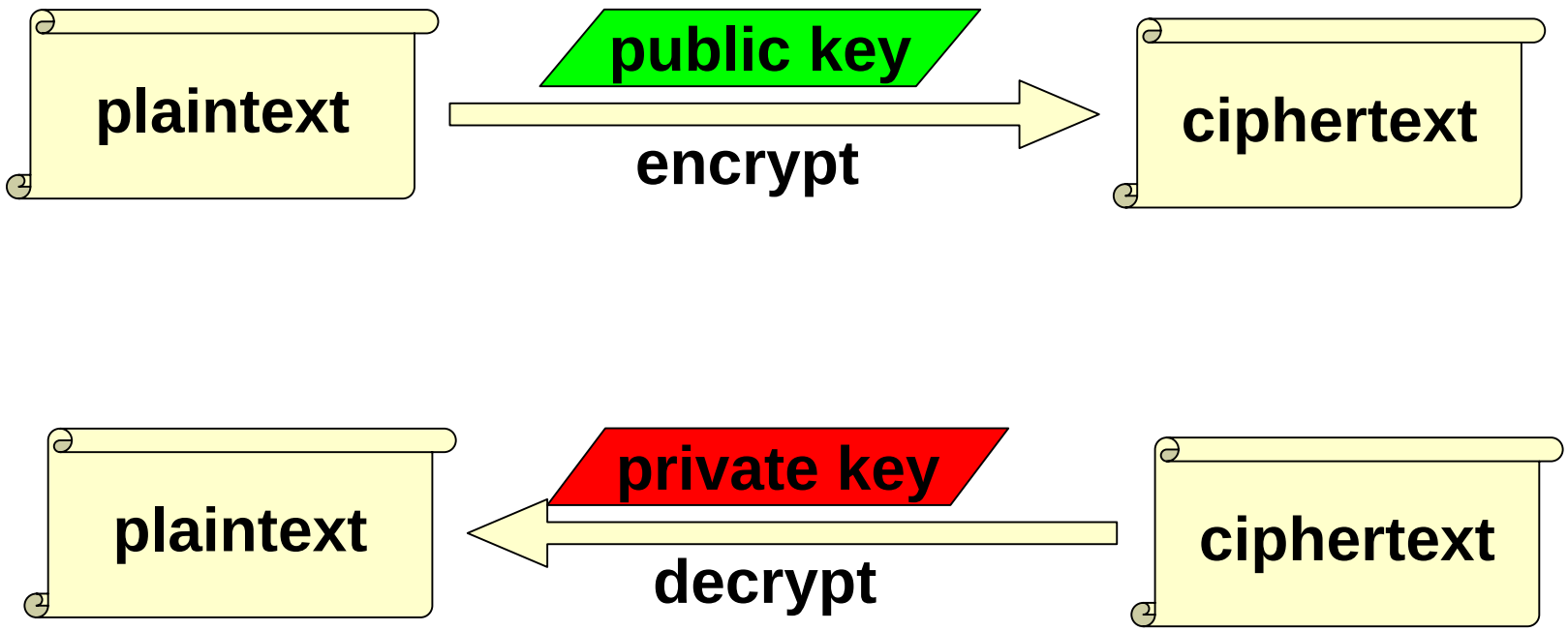
- Bruce Schneier, Secrets and Lies

Public and private keys

- **Keys are generated in pairs**
  - public key
  - private key
- **How sensitive are these keys?**
  - public key not sensitive, anyone can know it
  - private key very sensitive, only one party knows it
  - not possible to calculate the private key given the public key
- **Encryption**
  - plaintext encrypted with the public key
  - ciphertext decrypted with the corresponding private key
  - much, much slower than conventional cryptography!
- **RSA is the most well-known public key algorithm**



Public key cryptography



Signatures

- **Public key cryptography can also be used for signatures**
  - say we want to “sign” some piece of data
  - private key used to encrypt the data
  - if you can decrypt the data with Bob’s public key, then you develop trust that Bob encrypted it originally
- **Goals are twofold**
  - verify that we know who created the signature
  - verify that the data hasn’t been tampered with since signed

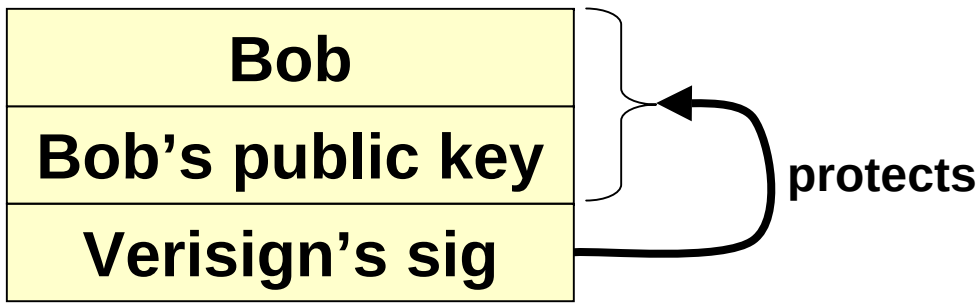


Reality checks

- **Nobody encrypts or signs bulk data with public keys**
  - it's way too slow!
  - we encrypt data with a conventional key, the "session key"
  - we use a public key to encrypt the session key
  - when signing, we encrypt a hash of the data, not the data itself
- **Public key cryptography is not a silver bullet!**
  - the public key isn't secret but it does need some protection
  - imagine a central database of public keys, and you want Bob's
  - if an attacker can swap his public key for Bob's public key:
    - he can decrypt any ciphertext you send to Bob
    - he can fake Bob's signature to you
  - digital certificates can help…

Digital certificates

- **Certificate couples a name with a public key**
  - contents of certificate are signed by some trusted authority, Verisign being the most notable nowadays
  - authority's public key is well-known
    - ships with browsers and other software
  - by verifying the authority's signature on the certificate, you develop trust in the contents of the certificate
- **Certificates are the most common way to communicate public keys over insecure channels**



Public key encryption in the .NET Framework

- **The current story is incomplete**
  - there is a class representing an X.509 certificate
  - there is a class representing an RSA key
  - there isn't any clear path between them
- **If you want to use standards (PKCS), use CAPICOM**
  - can easily use CAPICOM via COM interop
  - although this means that mobile code won't be doing this stuff
- **If you want to work with raw public and private keys, can use RSACryptoServiceProvider, but be careful**
  - how will you securely transmit the public key?
  - how will you securely store the private key?
  - how will you interop with other systems?



Using CAPICOM from .NET

- **Step 1: Get the redistributable**
  - capicom.dll doesn't ship with Windows
  - http://www.microsoft.com/msdownload/platformsdk/sdkupdate
  - downloads : redistributables
- **Step 2: Install it**
  - log in as a local administrator
  - install it somewhere secure (\windows\system32 works)
  - regsvr32 capicom.dll
- **Step 3: Run TLBIMP**
  - tlbimp capicom.dll
  - alternatively, add a reference to it from your Visual Studio project



Encrypting data using certs and CAPICOM

- **Setting up**
  - serialize the data you want to send into a string
  - choose the recipient(s) for the message
  - locate their certificates in the various certificate stores
  - choose a symmetric encryption algorithm
  - choose an encoding for the resulting ciphertext
- **Encrypting**
  - create an EnvelopedData object
  - add certificates for each recipient
  - set the content string
  - set the encryption algorithm
  - call Encrypt



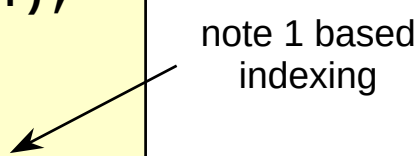
Example: encryption using certs and CAPICOM

```
Store s = new Store();
s.Open(CAPICOM_STORE_LOCATION.CAPICOM_CURRENT_USER_STORE,
       "AddressBook",
       CAPICOM_STORE_OPEN_MODE.CAPICOM_STORE_OPEN_READ_ONLY);

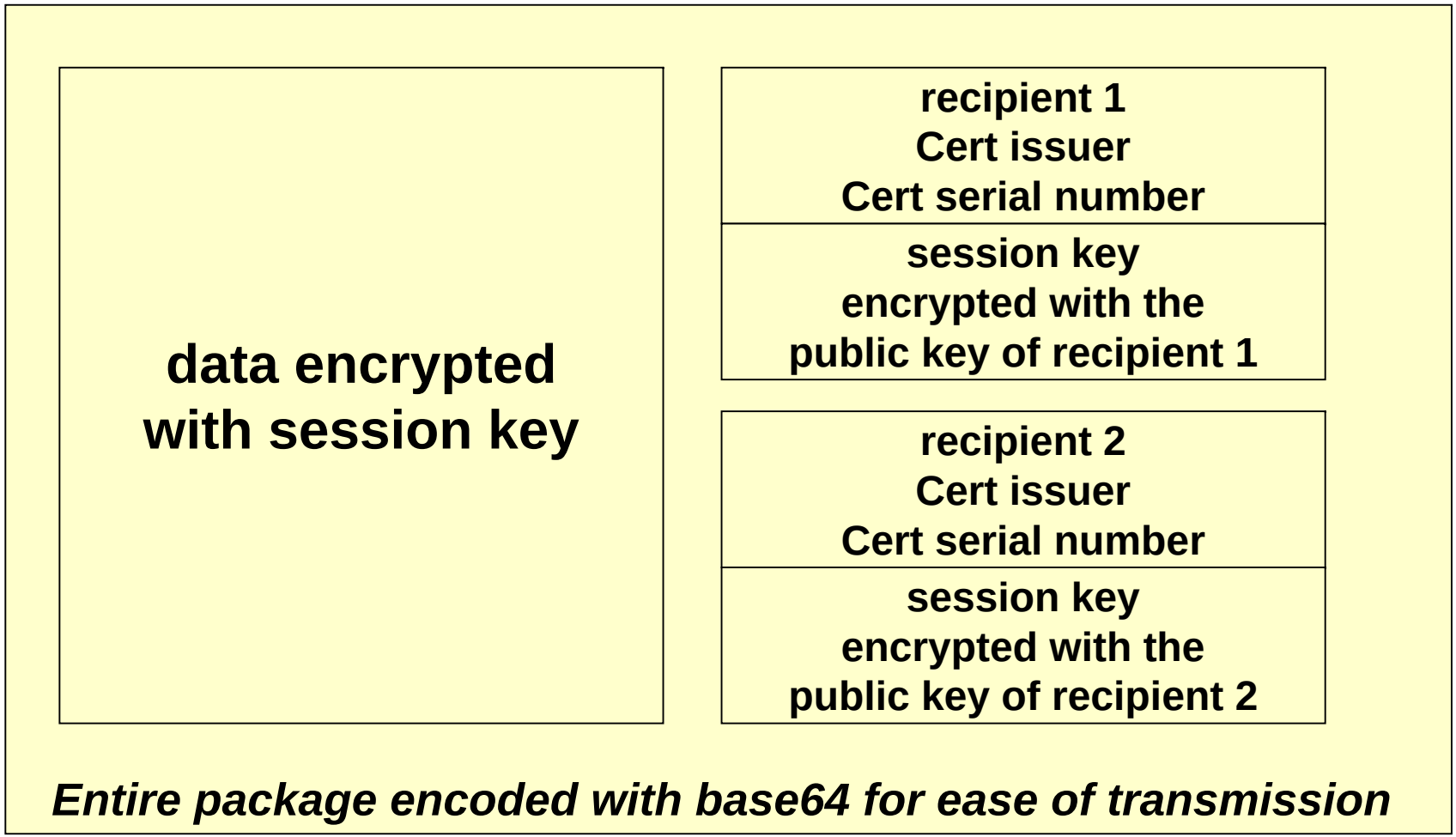
EnvelopedData env = new EnvelopedData();
env.Recipients.Add((Certificate)s.Certificates[1]);
env.Recipients.Add((Certificate)s.Certificates[2]);
env.Algorithm.Name =
CAPICOM_ENCRYPTION_ALGORITHM.CAPICOM_ENCRYPTION_ALGORITHM_3DES;

env.Content = "Here is the plaintext.";

string ciphertext =
   env.Encrypt(CAPICOM_ENCODING_TYPE.CAPICOM_ENCODE_BASE64);
```



What we just produced



Decrypting an enveloped message

```
EnvelopedData env = new EnvelopedData();

env.Decrypt(ciphertext);

string plaintext = env.Content;
```

looks in personal certificate store for a cert that matches one of the recipients in the envelope

Signing data using CAPICOM

- **Setting up**
  - serialize the data you want to send into a string
  - choose a certificate to use for the signature
  - remember you must have the private key for this certificate
  - choose an encoding for the resulting signed message
- **Signing**
  - create a Signer object
  - load the Signer with the certificate you'll be using
  - create a SignedData object and add the Signer to it
  - set the content property of SignedData
  - call SignedData.Sign to create the signed message
  - can optionally CoSign to add more signatures



Example: signing data

```
Store my = new Store();
my.Open(CAPICOM_STORE_LOCATION.CAPICOM_CURRENT_USER_STORE,
        "MY",
        CAPICOM_STORE_OPEN_MODE.CAPICOM_STORE_OPEN_READ_ONLY);

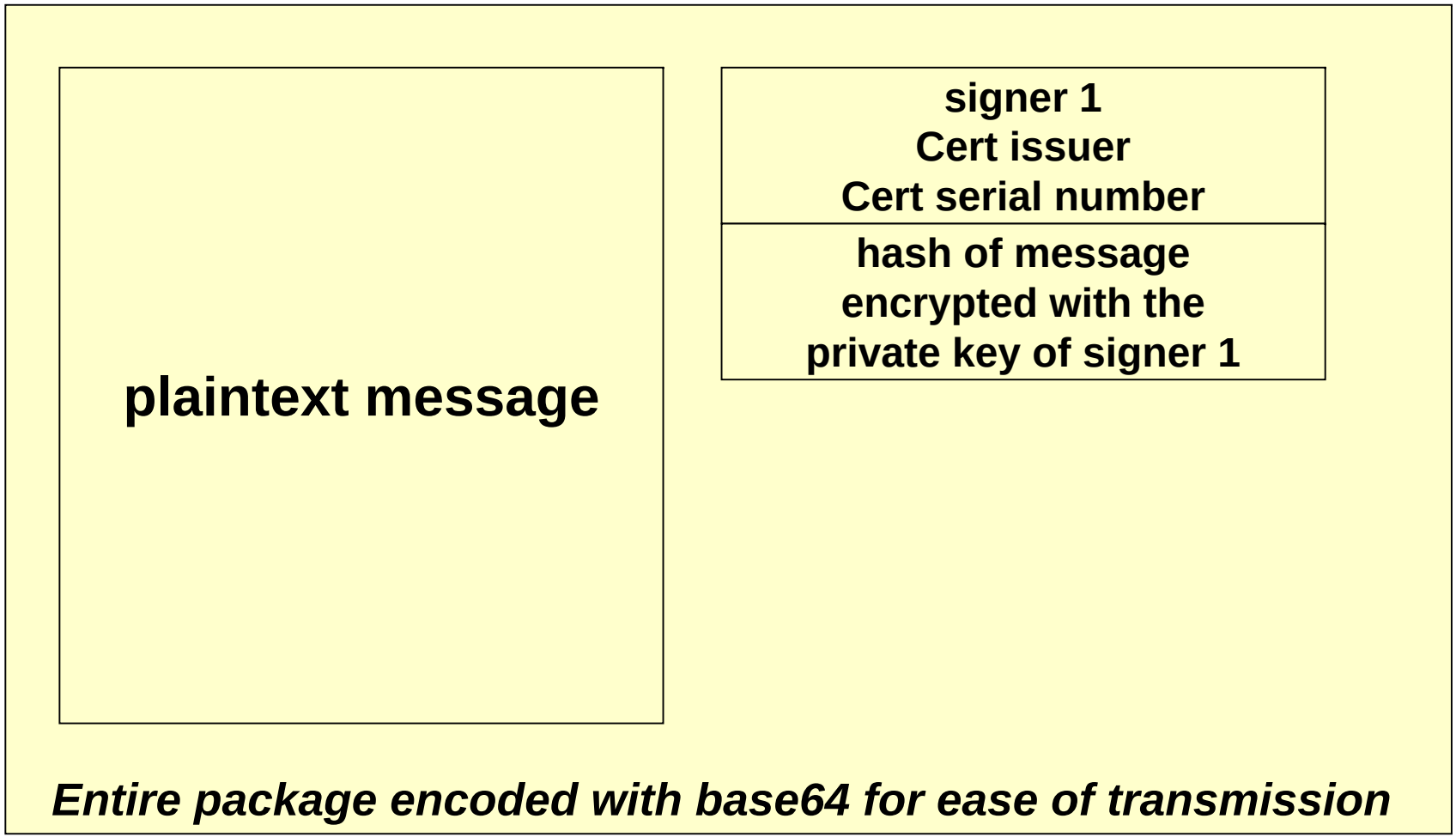
Signer me = new Signer();
me.Certificate = (Certificate)my.Certificates[1];

SignedData sd = new SignedData();
sd.Content = "Here is the message to be signed";

string signedMessage = sd.Sign(me, false,
  CAPICOM_ENCODING_TYPE.CAPICOM_ENCODE_BASE64);
```




What we just produced



Verifying signed data

- **Verification is a two step process**
  - Step 1: call SignedData.Verify
  - Step 2: look at each signer and see if the signature is valid
  - Note that this does not check cert revocation lists

```
SignedData sd = new SignedData();
sd.Verify(Console.In.ReadToEnd(), false,
  CAPICOM_SIGNED_DATA_VERIFY_FLAG.CAPICOM_VERIFY_SIGNATURE_AND_CERTIFICATE);

Console.WriteLine(sd.Content);
Console.WriteLine("Signature validity:");
foreach (Signer s in sd.Signers) {
    Console.Write(s.Certificate.IsValid().Result ?
      "Valid: " : "Invalid: ");
    Console.WriteLine(s.Certificate.SubjectName);
}
```

Verifying certificates

- **Just because the signature matches the public key doesn't mean it's a valid signature**
  - you need to check the validity of the certificate itself
  - this means looking all the way up the chain of trust
  - this means looking for revoked certificates
- **Certificate Revocation Lists (CRL) are important**
  - authorities that issue certificates also publish CRLs regularly
  - CAPICOM allows you to check against CRLs, but it's not automatic



Example: verifying a certificate

```
static void validateCert(Certificate certToValidate) {
    certToValidate.IsValid().CheckFlag =
        CAPICOM_CHECK_FLAG.CAPICOM_CHECK_SIGNATURE_VALIDITY         |
        CAPICOM_CHECK_FLAG.CAPICOM_CHECK_TIME_VALIDITY              |
        CAPICOM_CHECK_FLAG.CAPICOM_CHECK_TRUSTED_ROOT               |
        CAPICOM_CHECK_FLAG.CAPICOM_CHECK_OFFLINE_REVOCATION_STATUS  |
        CAPICOM_CHECK_FLAG.CAPICOM_CHECK_ONLINE_REVOCATION_STATUS;
    if (certToValidate.IsValid().Result) {
        return; // cert is fine
    }

    // something is wrong, find out the details
    Chain chain = new Chain();
    chain.Build(certToValidate);

    // fail and provide a description of the reason
    _throwBadCertException(chain.get_Status(0));
}
```

Authentication using certificates

- **A certificate simply makes a statement**
  - authority X asserts that this is subject Y's public key
- **Certificates are not secret**
  - so when someone presents you a cert over the wire, can you use that to figure out the identity of the sender?
  - sender needs to prove that she knows the corresponding private key!
- **This is what SSL is all about**

SSL and TLS

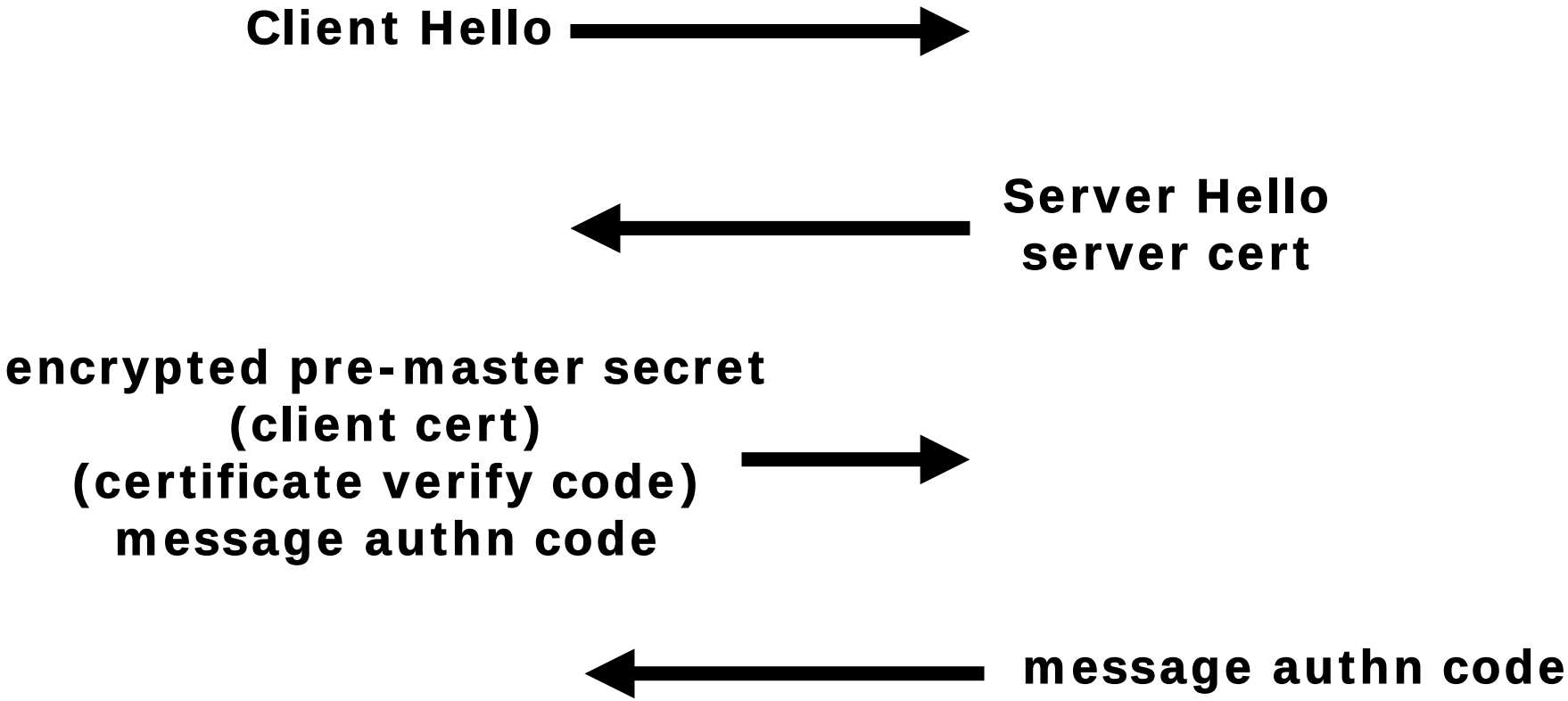
- **SSL == Secure Sockets Layer**
  - developed by Netscape in early ’90s
  - became a de-facto standard on the web
- **TLS == Transport Layer Security**
  - really just SSL 3.1 put into an IETF standard (RFC 2246)
- **Authentication protocol based on certificates**
- **Remember that authentication does two things**
  - helps you develop trust in the identity of your peer
  - helps you and your peer exchange a session key

Who does SSL authenticate?

- **SSL always authenticates the server**
- **SSL can also authenticate the client, if desired**
  - this means the client has to have a certificate
  - not common for the average computer user
  - very common in B2B scenarios



The SSL handshake



Using SSL

- **SSL can be used directly via SSPI**
  - very tricky to implement correctly
- **Most people use IIS to get SSL support**
  - ASP.NET web applications have IIS as a front end
  - web services have IIS as a front end
  - secure .NET remoting uses IIS as a front end



Summary

- Public key cryptography is used to create signatures and encrypt keys
- Certificates help authenticate public key material
- CAPICOM can be used from the .NET Framework
- CAPICOM helps you encrypt/sign using certificates
- SSL is an authentication protocol based on certificates



Windows Security 101

A whirlwind tour of operating system security concepts on the .NET Platform

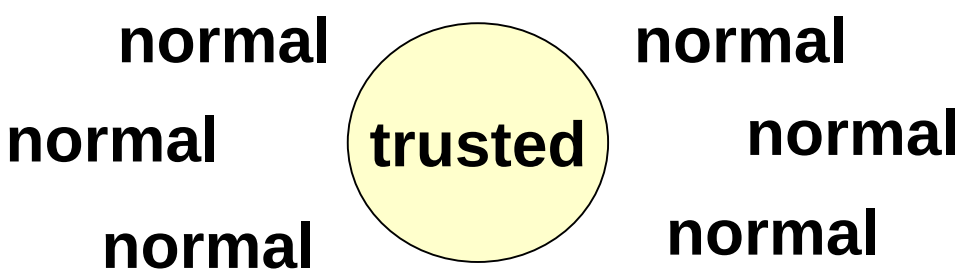


Outline

- Windows security primer
- Processes and tokens
- Role based security
- Tracking client identity
- Logon sessions and window stations

Trusted Computing Base (TCB)

- **The TCB is a theoretical boundary**
  - the TCB must enforce security policy
  - normal software is subject to the security policy of the machine
- **TCB consists of most hardware and some software**
  - “Act as part of the operating system” privilege puts you in TCB
  - SYSTEM (a.k.a. “Local System”) has this privilege
- **The administrator’s role**
  - define and defend the TCB
  - define security policy (groups, privileges, etc.)

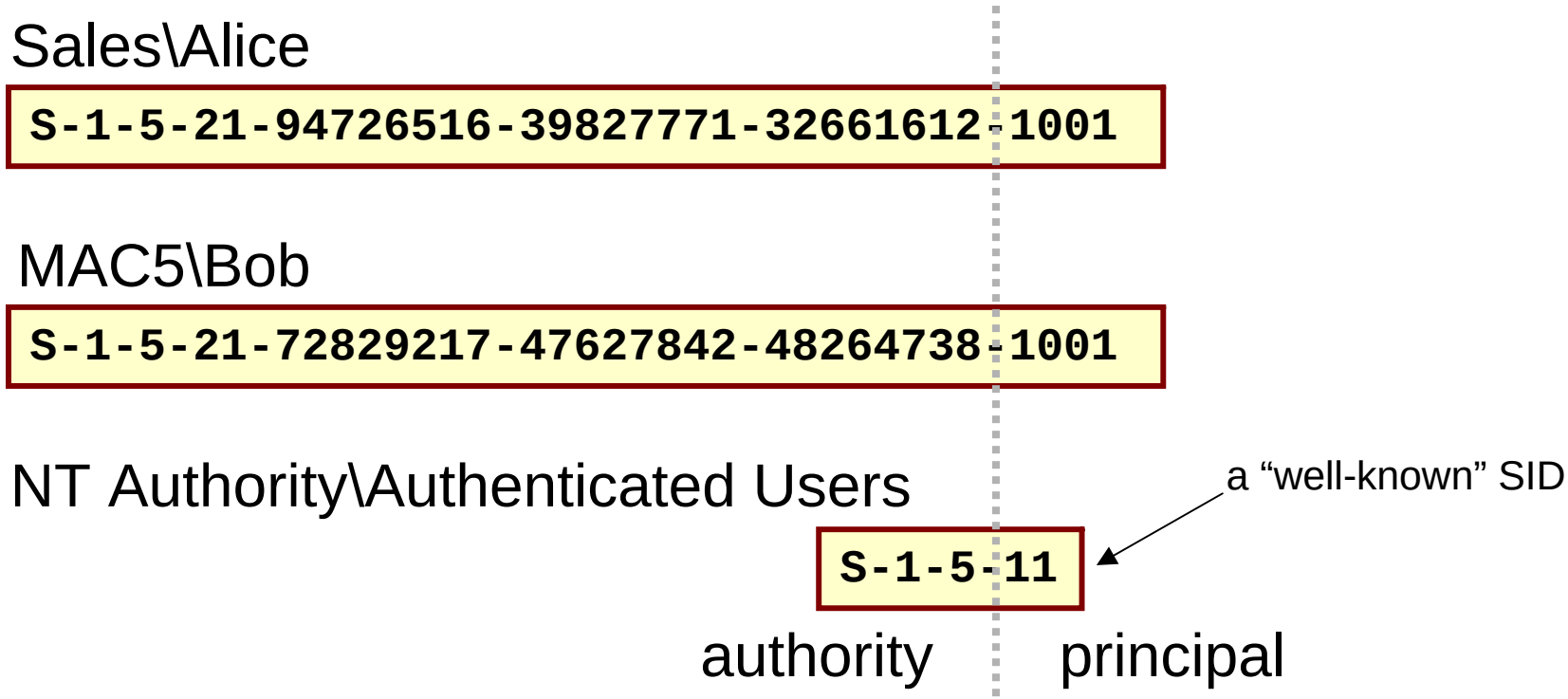


Principals, accounts, and authorities

- **An *authority* issues an *account* for each *principal***
  - local authority issues local accounts
  - domain authority issues domain accounts
- **Fully qualified name includes two parts: authority\principal**
  - MAC5\Alice
  - SalesDomain\Bob
- **Domain accounts rely on Kerberos authentication**
  - can be authenticated anywhere in the domain
  - can be authenticated in any other trusting domain
- **Local accounts use an older authentication protocol (NTLM)**
  - two reasons to use a local account:
    - no network credentials are needed, or
    - no domain is present

Security Identifiers (SIDs)

- **Used to identify both users and groups**
  - Unique in space and time
  - Similar to a GUID, but has a structure you can parse



Groups

- **There are four types of groups**
  - Some follow you all over the network (unlimited scope)
  - Others only show up in the scope where they are defined (machine or domain scope)

type of group	stored in	nest?	scope
Universal	active directory	yes	unlimited
Global	active directory	yes	unlimited
Domain Local	active directory	yes	single domain
Local	registry	no	single machine

Privileges

- **Groups typically used to grant permissions on objects**
  - via access control lists (ACLs)
- **Privileges are broader rights granted to users**
  - privileges are bestowed via security policy
    - secpol.msc for local machine policy
    - gpedit.msc for group policy
- **Some interesting privileges**
  - bypass traverse checking
  - backup, restore
  - debug programs
  - shut down the system
  - generate security audits
  - act as part of the operating system (part of the TCB, that is)

Processes and tokens

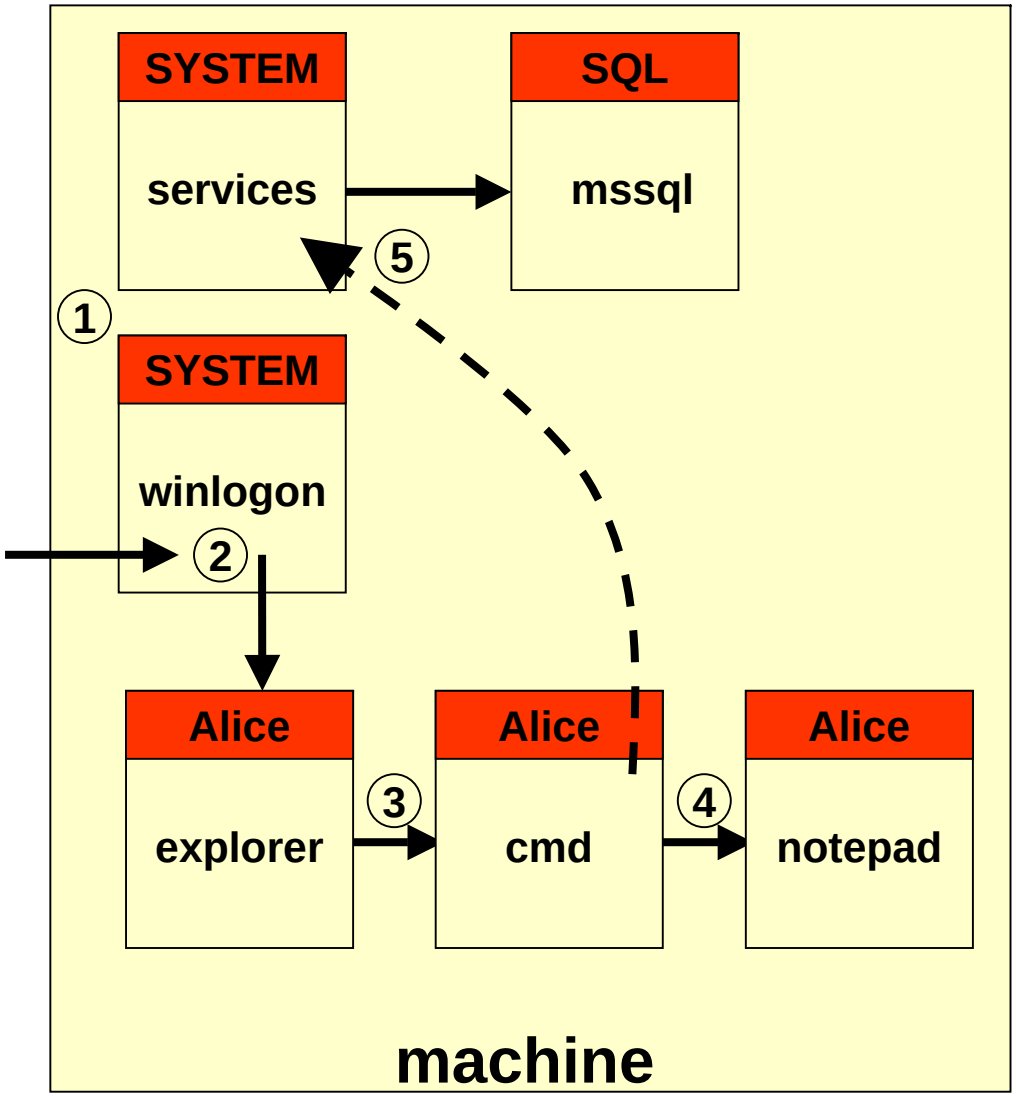
- **Each process has a security identity**
  - this associates the process with a principal
  - required to enforce access control policies
  - required to audit actions performed by each principal
- **This info is stored in a kernel object known as a token**
  - user SID
  - group SIDs
  - privileges
- **Even a daemon must have a token**
  - NT services
  - COM+ applications
  - Web application worker processes

Token propagation

- **When a new process is created, it needs a token**
  - created by copying token from parent process
- **When a daemon is started, it also needs a token**
  - created from user name and password configured for daemon



Token propagation



1. machine boots up
2. Alice logs in interactively
3. Alice starts a command shell
4. cmd: start notepad.exe
5. cmd: net start mssql

Programming with tokens

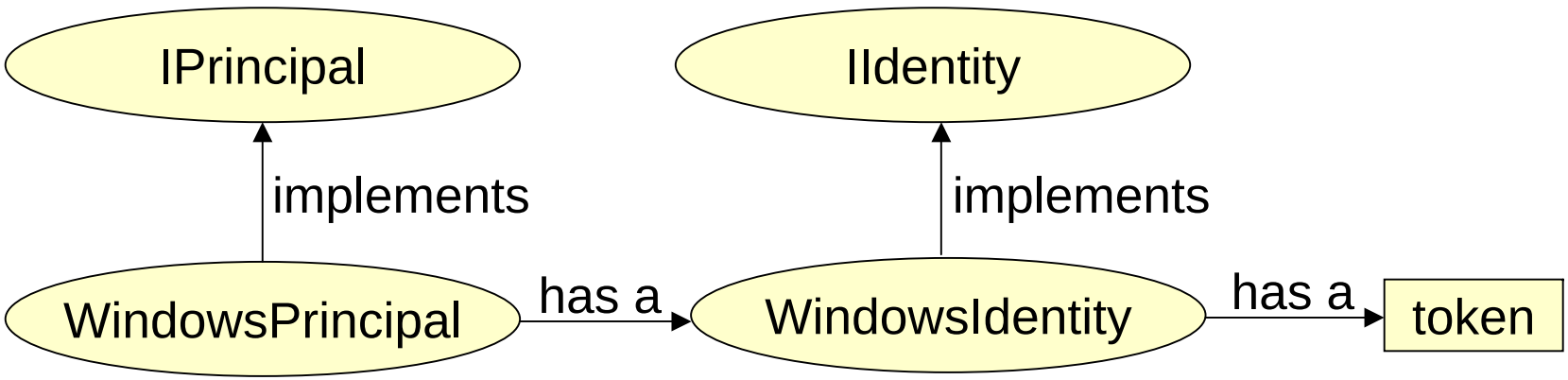
- **Usually we care about two things in a token**
  - identity of the user
  - groups in which that user is a member
- **The .NET Framework abstracts this with two interfaces**

```
interface IIdentity {
  bool   IsAuthenticated    { get; }
  string AuthenticationType { get; }
  string Name               { get; }
}
```

```
interface IPrincipal {
  IIdentity Identity { get; }
  bool      IsInRole(string roleName);
}
```

WindowsIdentity and WindowsPrincipal

- **These are simply wrappers for tokens**
- **WindowsPrincipal.IsInRole queries groups in token**
  - group names must be fully qualified
- **WindowsIdentity has some useful features:**
  - GetCurrent() retrieves process token*
  - GetAnonymous() retrieves a null session token
  - Impersonate() is also important, but more on this later…



Example: a simple desktop application

- **Runs in a single process**
- **Has a process token for the launching user**
  - it'll be able to do anything that user can do

```
using System;
using System.Security.Principal;

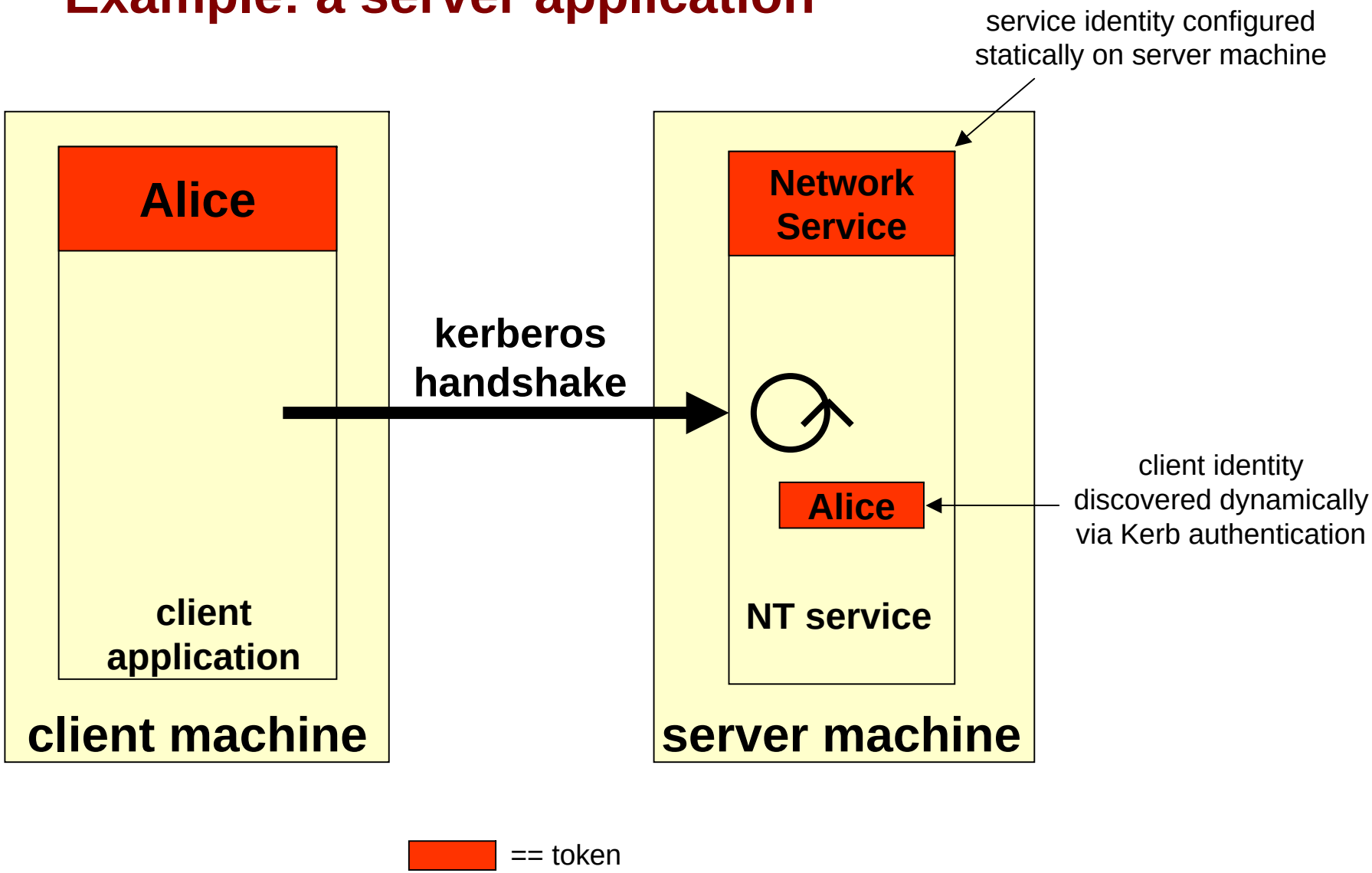
class WhoAmI {
  static void Main() {     // look at process token
    IIdentity   identity = WindowsIdentity.GetCurrent();
    IPrincipal principal = new WindowsPrincipal(
                                (WindowsIdentity)identity);
    if (identity.IsAuthenticated) {
      Console.WriteLine("My name is {0}", identity.Name);
      if (principal.IsInRole(@"BUILTIN\Backup Operators")) {
        Console.WriteLine("I am a Backup Operator");
      }
    }
    else Console.WriteLine("I am anonymous.");
  }
}
```




Server applications and tokens

- **Server applications are usually packaged as daemons**
  - Web apps (w3wp.exe)
  - COM+ applications (dllhost.exe)
  - NT services (*.exe)
- **Server application identity is configurable**
  - Web apps configured via “application pools”
  - COM+ apps configured via Component Services catalog
  - NT services configured via control panel applet
- **For each authenticated client, server gets another token**
  - Kerberos ticket sent from client is turned into a token
  - server looks at client token to make security decisions
  - can check client’s groups via IsInRole, for example

Example: a server application



Keeping track of your clients

- **FCL provides a variable to help servers track client identity**
  - Thread.CurrentPrincipal
  - as it's name implies, this variable is kept on a per-thread basis
- **Pattern for usage**
  - plumbing sets it (ASP.NET, for example, or your own code)
  - application gets it (business logic) to make security decisions
- **This is key to "Role Based Security" infrastructure in .NET**
- **This is *not* impersonation**
  - Thread.CurrentPrincipal is simply for application bookkeeping
  - the *operating system* doesn't ever see or use this information
  - we'll talk about impersonation in the next module…

Tracking client identity

```
using System.Threading;
using System.Security.Principal;

class Plumbing {
  public static void DoHeavyLifting() {
    IPrincipal client = AuthenticateUserSomehow();
    Thread.CurrentPrincipal = client;
    Application.ImplementBusinessLogic();
  }
}

class Application {
  public static void ImplementBusinessLogic() {
    if (Thread.CurrentPrincipal.IsInRole("Managers")) {
      // do something privileged
    }
  }
}
```

More fun with Thread.CurrentPrincipal

- **PrincipalPermissionAttribute uses Thread.CurrentPrincipal**
  - allows you to add declarative checks against client identity
  - JIT compiler inserts the checks at the start of the method

```
using System.Security.Permissions;

[PrincipalPermission(SecurityAction.Demand, Authenticated=true)]
class PetStore {
  public void PetAnimals() { ... }
  public void BuyAnimals() { ... }

  [PrincipalPermission(SecurityAction.Demand, Role="Staff")]
  public void FeedAnimals() { ... }

  [PrincipalPermission(SecurityAction.Demand, Role="Managers")]
  public void GiveRaise() { ... }
}
```

Thread.CurrentPrincipal and asynchrony

- **What if I make an asynchronous call?**
  - BeginInvoke copies Thread.CurrentPrincipal to worker thread
  - Thread.Start copies Thread.CurrentPrincipal to new thread
- **Watch out for these special cases**
  - ThreadPool.QueueUserWorkerItem doesn't copy it
  - System.Threading.Timer doesn't copy it



Defaults for Thread.CurrentPrincipal

- What if no plumbing has set Thread.CurrentPrincipal?
- An AppDomain property controls this…

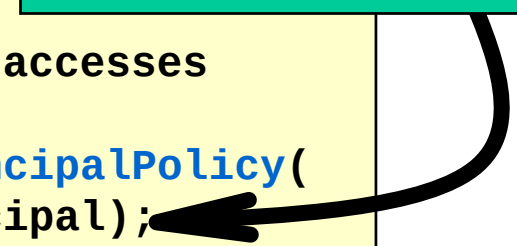
```
using System;
using System.Security.Principal;

class SimpleConsoleApp {
    static void Main() {
        // must set this before anyone accesses
        // Thread.CurrentPrincipal
        AppDomain.CurrentDomain.SetPrincipalPolicy(
           PrincipalPolicy.WindowsPrincipal);

        DoSomeWork();
    }

    // ...
}
```

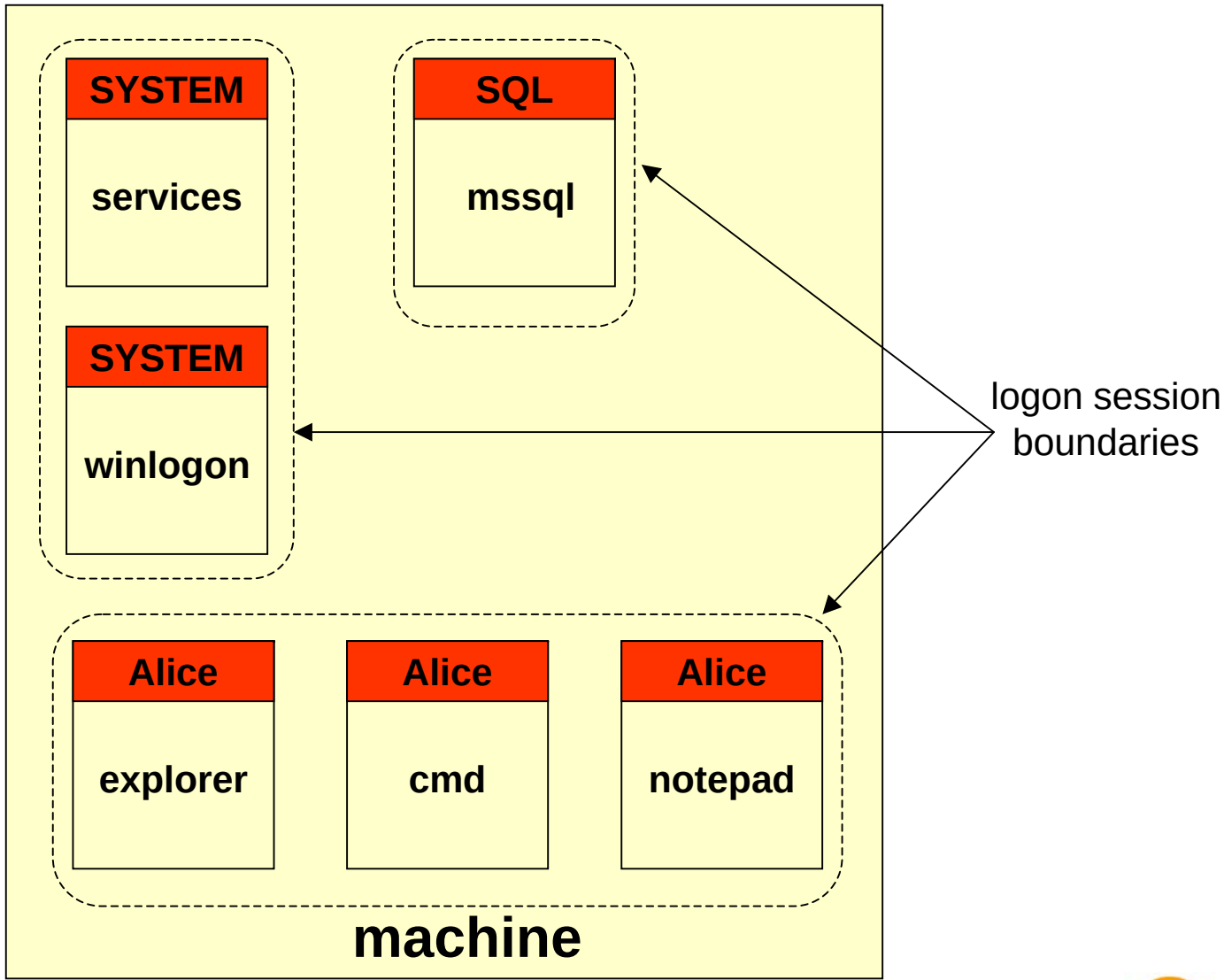
```
NoPrincipal
UnauthenticatedPrincipal
WindowsPrincipal
```



Logon sessions

- **A logon session is a data structure in the kernel**
  - represents an instance of a principal on a machine
  - holds network credentials (kerb tickets, pwd hash, etc.)
  - a token (and thus a process) is tied to a single logon session
  - “logging off” tears down processes tied to the logon session
- **You can have more than one logon session for one principal**
  - interactive user could be Alice
  - service could be running as Alice
  - if the interactive Alice logs off, this has no effect on the service
- **There are several built-in logon sessions**
  - #999: SYSTEM
  - #998: Null Session (the anonymous logon)
  - #997: Local Service
  - #996: Network Service

Logon sessions, illustrated

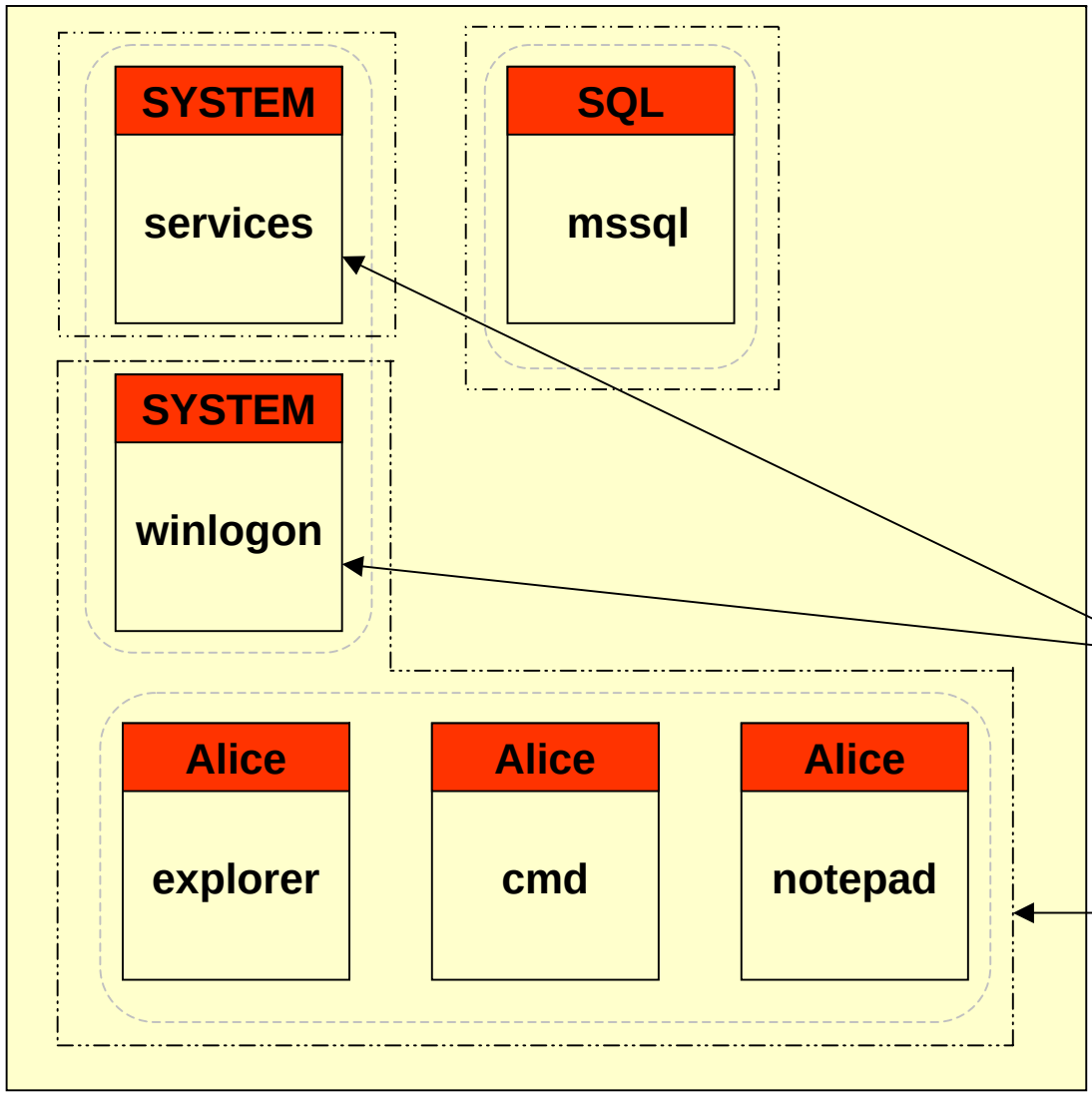


Window stations

- **Window and GDI objects have no individual security settings**
  - compare this to files, registry keys, processes, each of which has a security policy saying who can manipulate them
- **The entire windowing environment is sandboxed**
  - this sandbox is known as a Window Station
- **Window stations have:**
  - a set of one or more desktops on which windows are created
  - a clipboard and atom table
- **Each process has an attribute indicating which window station it belongs to**



Window stations, illustrated



window station boundary

logon session boundary

note that services running as SYSTEM can choose to run in the interactive WS ("interact with the desktop" checkbox)

interactive window station (winsta0)

Why care about window stations?

- **Two common reasons to care about window stations**
  - cannot send window messages across winstation boundaries
  - only interactive window station (winsta0) bound to hardware
- **Rules to live by**
  - don't use window messages for interprocess communication
  - don't put up dialog boxes unless you're running interactively
    - dialogs displayed in noninteractive winstations will hang!
    - server code should beware of ASSERT macro in C/C++
- **MessageBox can be used with a special flag**
  - MB_SERVICE_NOTIFICATION (C++)
  - MessageBoxOptions.ServiceNotification (CLR)
  - causes the message box to show up on the interactive desktop, even if initiated from a noninteractive window station



Running applications in alternate logon sessions

- **Secondary Logon service makes this easy**
  - handles details such as granting permission to enter winsta0
- **What you provide**
  - path to the app you want to run, plus user name and password
- **What the service does**
  - authenticates user name and password getting a token
  - starts new process with that token in winsta0
  - loads user profile if you desire
  - logs off any secondary logons when interactive user logs off*
- **CreateProcessWithLogonW is the API that makes it happen**
  - runas utility or shift-right-clicking via explorer

Example: starting a command prompt as Alice

```
// See labs/extraSamples/secondaryLogon.zip for full sample
StartupInfo si = new StartupInfo();
si.cb          = Marshal.SizeOf(typeof(StartupInfo));
si.title       = "This command prompt is running as Alice";

ProcessInfo pi = new ProcessInfo();
string app = Path.Combine(Environment.SystemDirectory,
                          "cmd.exe");

bool result = CreateProcessWithLogonW(
    "alice", ".", promptUserForPassword(),
    LogonFlags.LOGON_WITH_PROFILE,
    app, null,
    0, IntPtr.Zero, null,
    ref si, out pi);

if (result) {
  result = CloseHandle(pi.hProcess);
  result = CloseHandle(pi.hThread);
}
```

Summary

- Authorities vouch for principals at runtime
- Groups and privileges are part of security policy
- Administrator specifies security policy, TCB enforces it
- Local and Domain accounts are both possible
- Every process has a token, even daemons
- Servers track client identity using Thread.CurrentPrincipal
- Logon sessions cache credentials and help determine process lifetime
- Window stations are used to sandbox the interactive GUI
- For further info, consult *Programming Windows Security*

Windows Security 102

Continuing our whirlwind tour of operating system security concepts on the .NET Platform



Outline

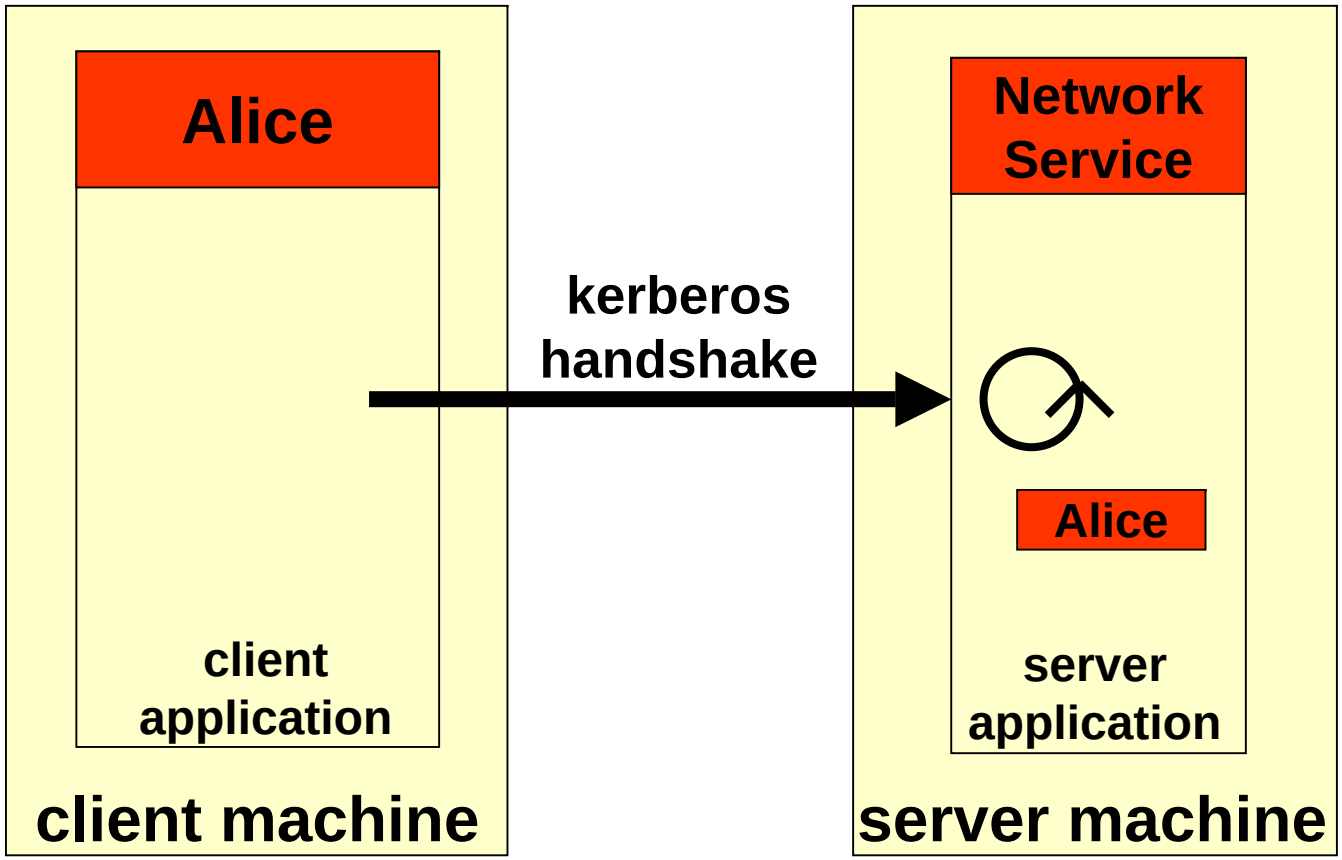
- Servers and client identity
- Impersonation
- Delegation in Windows 2000
- Delegation in Windows Server 2003

Network authentication using Kerberos

- **Secure servers require clients to be authenticated**
  - Kerberos most typical on today's intranets
- **Successful authentication results in a logon session and token for the client on the server machine**
  - almost as if the client had logged on locally



Network authentication, illustrated

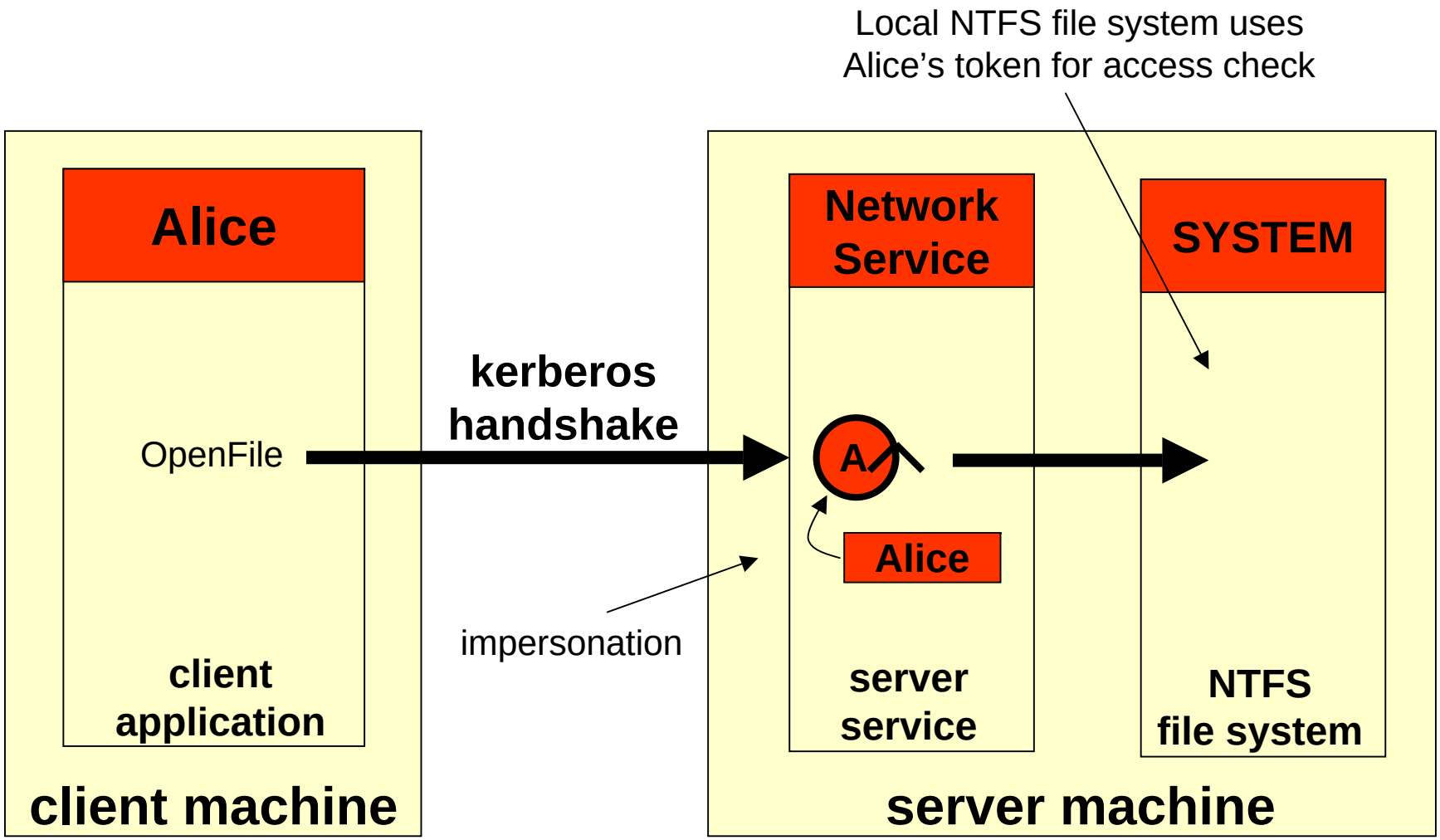


Impersonation

- **After authentication, the server can do two things with the client's token**
- **Look at it and make access control decisions**
  - perhaps to verify client is in a required role
  - perhaps to perform an access check against some resource
- **Use it to impersonate the client**
  - place the client's token on the server's thread and call out to some other secure resource manager
    - file system
    - SQL Server
    - etc.



Impersonation example: the file server



C# impersonation example

```
void impersonationFun(WindowsIdentity client) {

    // id is basically a wrapper around a token
    Console.WriteLine("Client name is {0}", client.Name);

    // slap the client's token on this thread temporarily
    WindowsImpersonationContext ctx = client.Impersonate();
    try {
        // this request to the file system will only succeed
        // if the client has access to the file
        new FileStream("someFile.txt", FileMode.Open);
    }
    finally {
        ctx.Undo(); // remove client's token from this thread
    }

    // this request to the file system will only succeed
    // if the server process has access to the file
    new FileStream("someFile.txt", FileMode.Open);
}
```

Impersonation is now a privileged operation

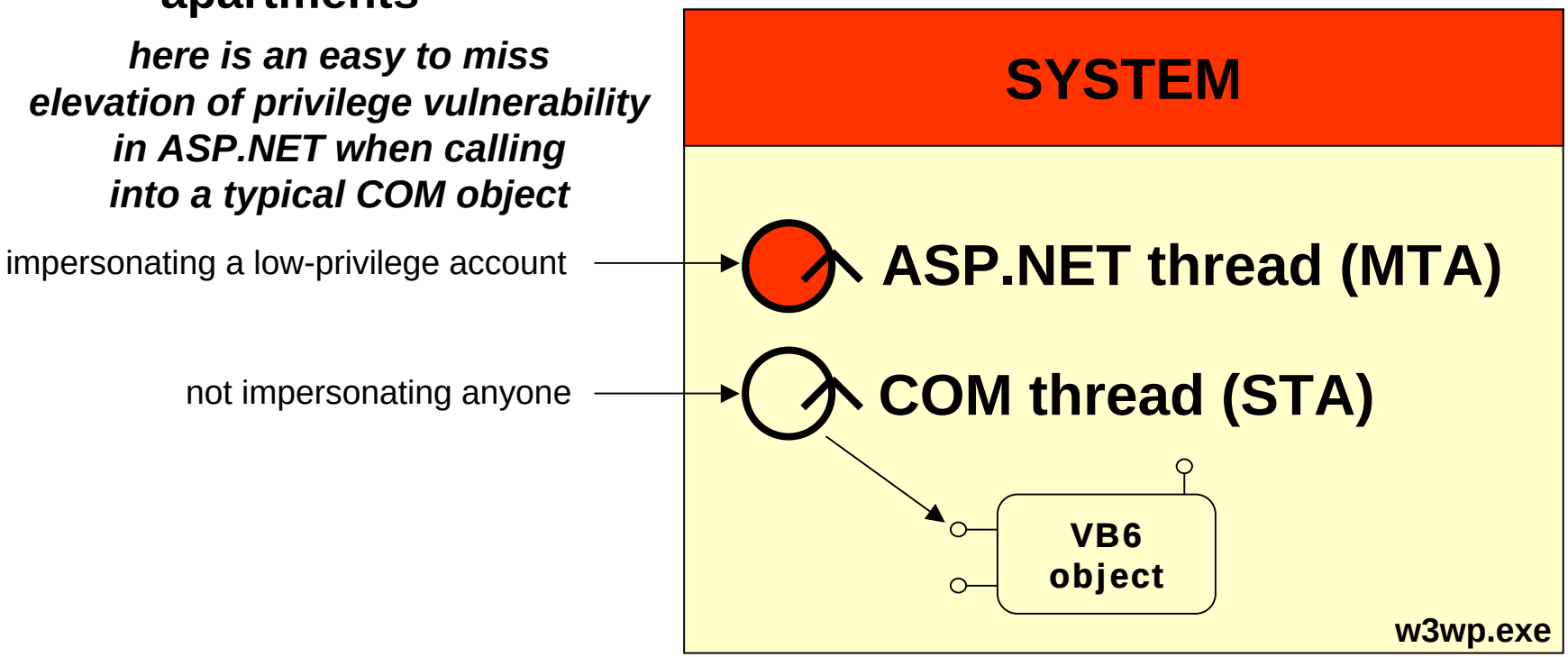
- **.NET Server 2003 added a new privilege**
  - SeImpersonatePrivilege
- **This change is being backported to older operating systems**
  - Windows 2000 SP4 (released June 26, 2003)
  - Windows XP (future service pack)
- **Must have this privilege granted and enabled to impersonate**
  - It's enabled by default if it's granted to you by policy
- **Granted by default to**
  - SYSTEM
  - Network Service
  - Local Service
- **If you run under a custom account, you'll need to grant this privilege explicitly in order to impersonate**



Impersonation is fragile: use it sparingly!

- Operating system doesn't auto-propagate thread token
- Beware asynchronous operations while impersonating
- Beware unforeseen thread switches due to COM "apartments"

here is an easy to miss elevation of privilege vulnerability in ASP.NET when calling into a typical COM object



When impersonating, you're in a wacky state

- **The thread token:**
  - is used for access checks against local kernel objects
  - is used as network identity in many cases (SQL Server integrated security, named pipes, etc.)
- **The process token:**
  - is copied to new child processes
  - is used for outgoing DCOM calls by default
  - is one security context an attacker can use if he takes over your process (say, via a buffer overflow)

Role based security and impersonation

- **Use Thread.CurrentPrincipal to track client identity**
  - rely on PrincipalPermissionAttribute for role checks
  - call IsInRole when the attribute doesn't suffice
- **Use impersonation if necessary, but keep to a minimum**
  - impersonate temporarily to access external resources as client
  - stop impersonating immediately
  - reduces complexity and helps avoid opening security holes
- **Carefully consider how asynchrony affects your design**
  - Asynchrony can cause unexpected changes in security context if you're impersonating (and it sometimes can effect Thread.CurrentPrincipal)

Example

```
using System.IO;
using System.Security.Principal;
using System.Security.Permissions;
using System.Threading;

[PrincipalPermission(SecurityAction.Demand, Authenticated=true)]
void DoSomeWorkForMyClient() {

    WindowsIdentity client = (WindowsIdentity)Thread.CurrentPrincipal.Identity;

    // temporarily impersonate the client to access a file
    WindowsImpersonationContext ctx = client.Impersonate();
    try {
        // open a file on the client's behalf
        using (Stream s = new FileStream("someFile.txt", FileMode.Open)) {
            // use the client's file...
        }
    }
    finally {
        ctx.Undo();
    }
}
```

Tokens and network credentials

- **Tokens for remote clients are usually weak**
  - authentication helped to verify the client's credentials, but most likely did not transmit them to the server
- **While impersonating a remote client**
  - server will likely be able to access local resources as that client
  - server will likely NOT be able to access remote resources as that client



Delegation

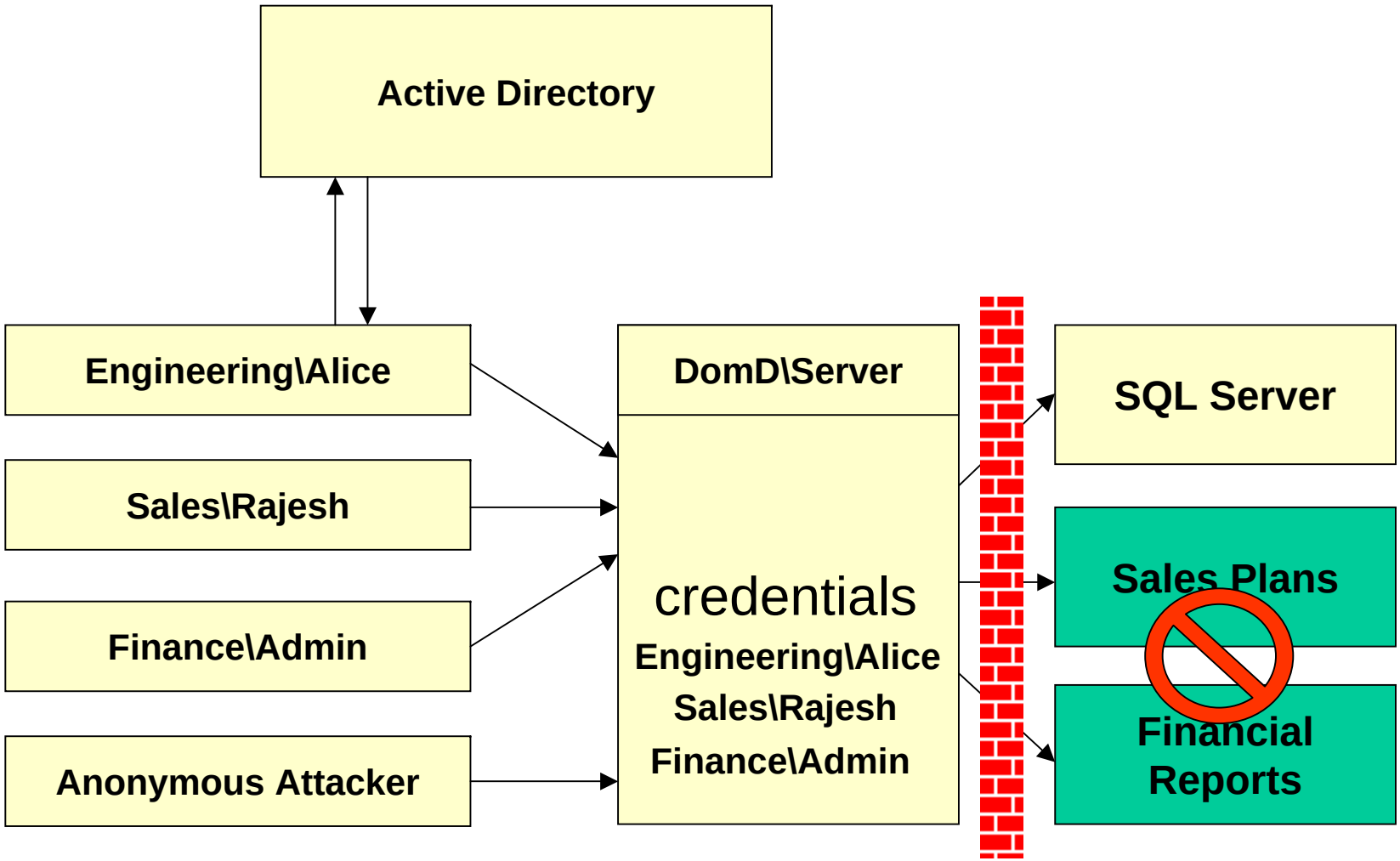
- **Kerberos allows client to send a network credential to server**
  - this feature is known as delegation
  - very dangerous feature that must be controlled carefully
- **Servers must be highly trusted to receive these credentials**
  - a server in domain A could get the delegated network credentials of a domain administrator in domain B
- **Windows 2000 and Windows Server 2003 differ in how they address this issue**



Windows 2000 unconstrained delegation

- **Server machine or service account flagged as special in Active Directory**
  - "trusted for delegation"
- **Individual user accounts may be restricted from participating**
  - Active Directory will not issue delegated tickets for any client account marked "sensitive and cannot be delegated"
- **This is the only option on Windows 2000**
  - it's very rarely used in practice because it's so loose

Unconstrained delegation, illustrated



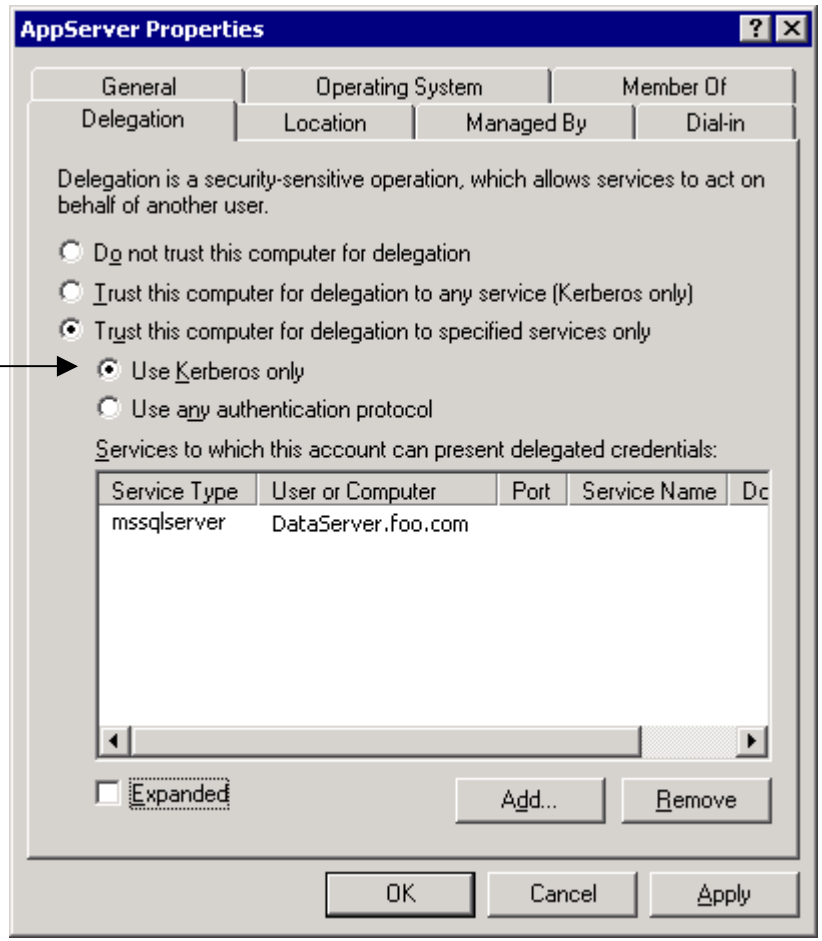
Windows Server 2003 constrained delegation

- **Server machine or service account may be flagged as trusted**
  - for unconstrained delegation, as in Win2k
  - or for constrained delegation
  - or for protocol transition, which we'll talk about later…
- **Authority only issues delegated tickets for services on the list**
  - this is called the "allowed-to-delegate-to" list
- **Individual user accounts may be restricted from participating**
  - Active Directory will not issue delegated tickets for any client account marked "sensitive and cannot be delegated"
  - just like in Win2k

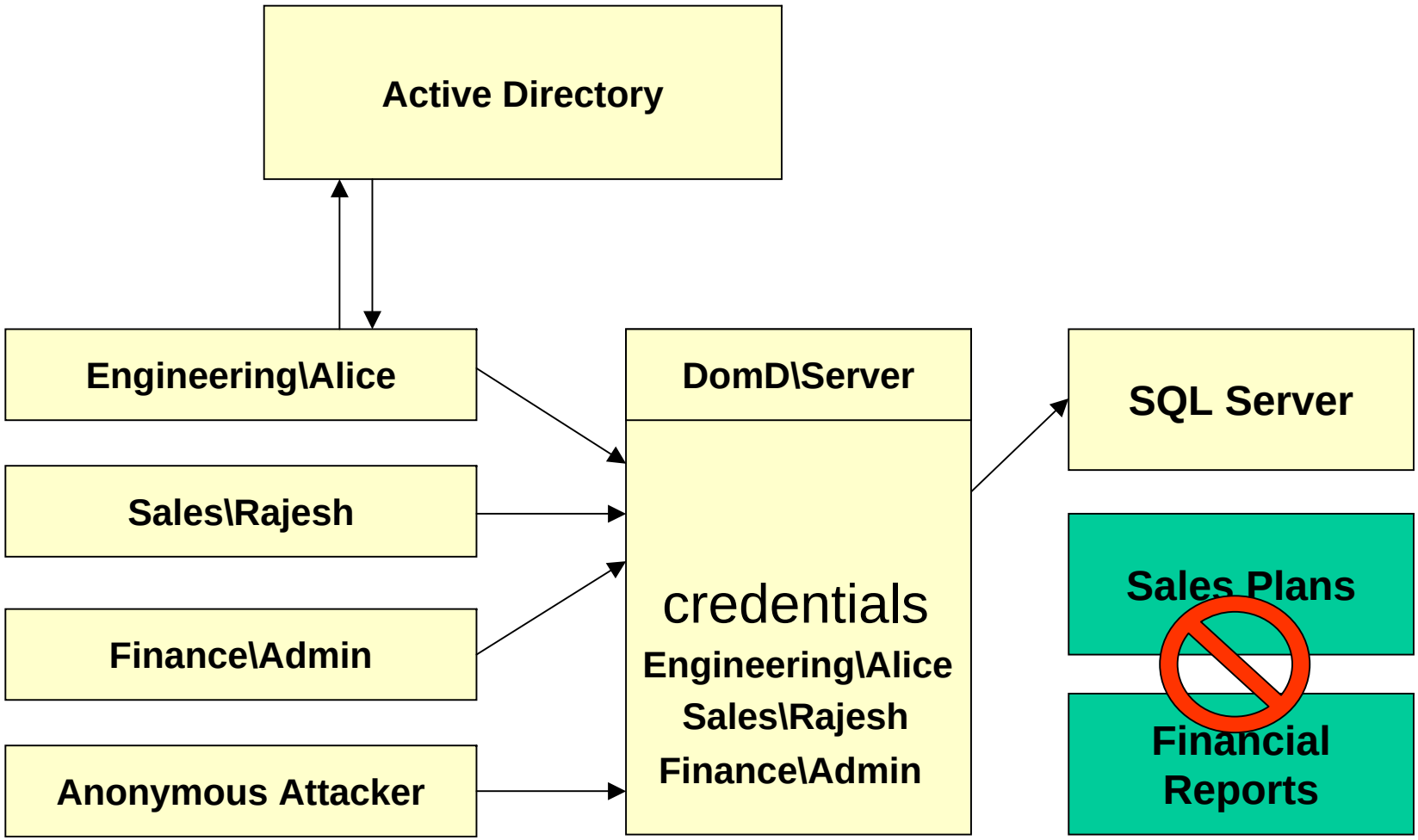


Configuring constrained delegation

“Constrained delegation”



Constrained delegation, illustrated

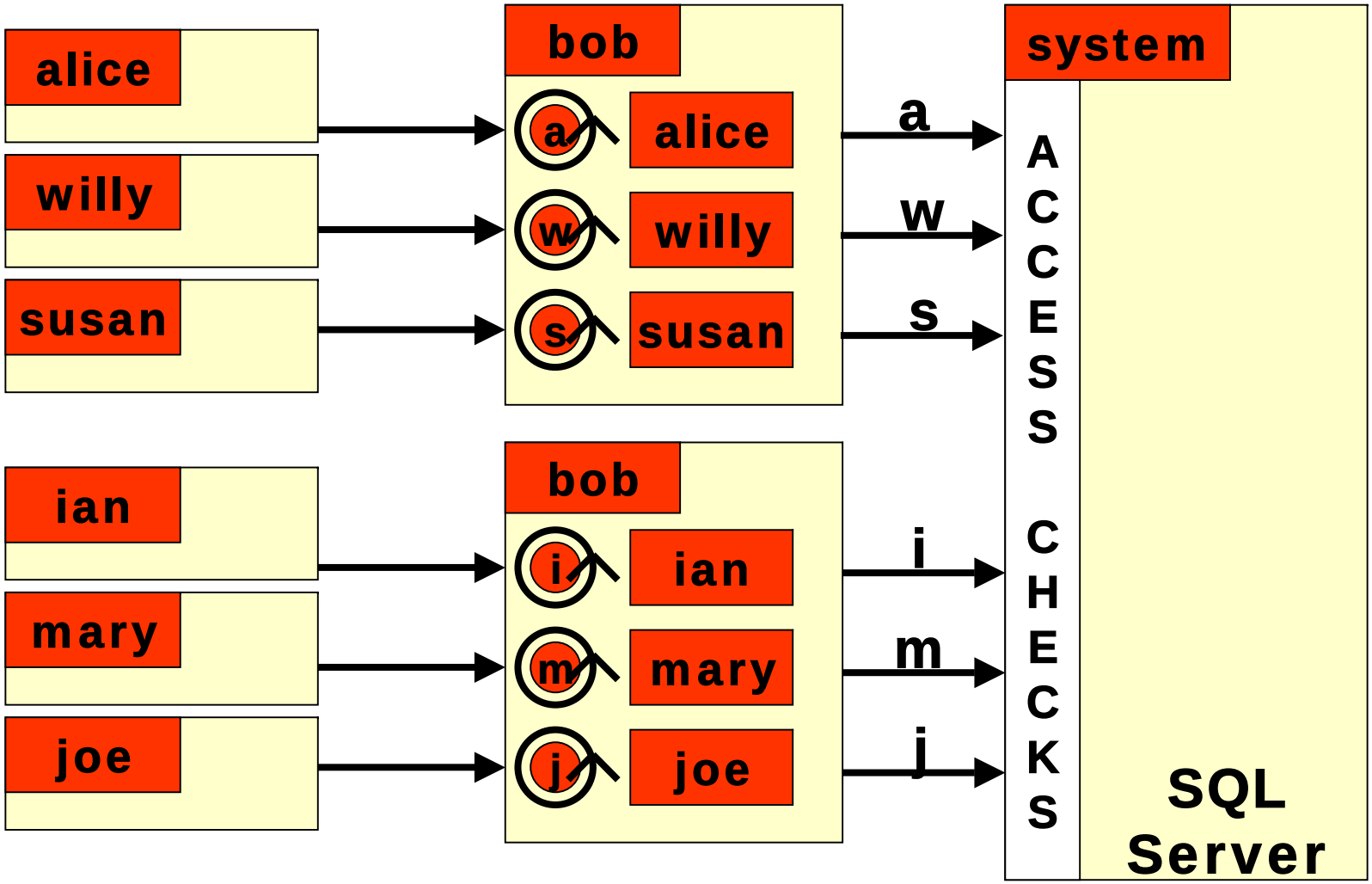


Delegation and scalability

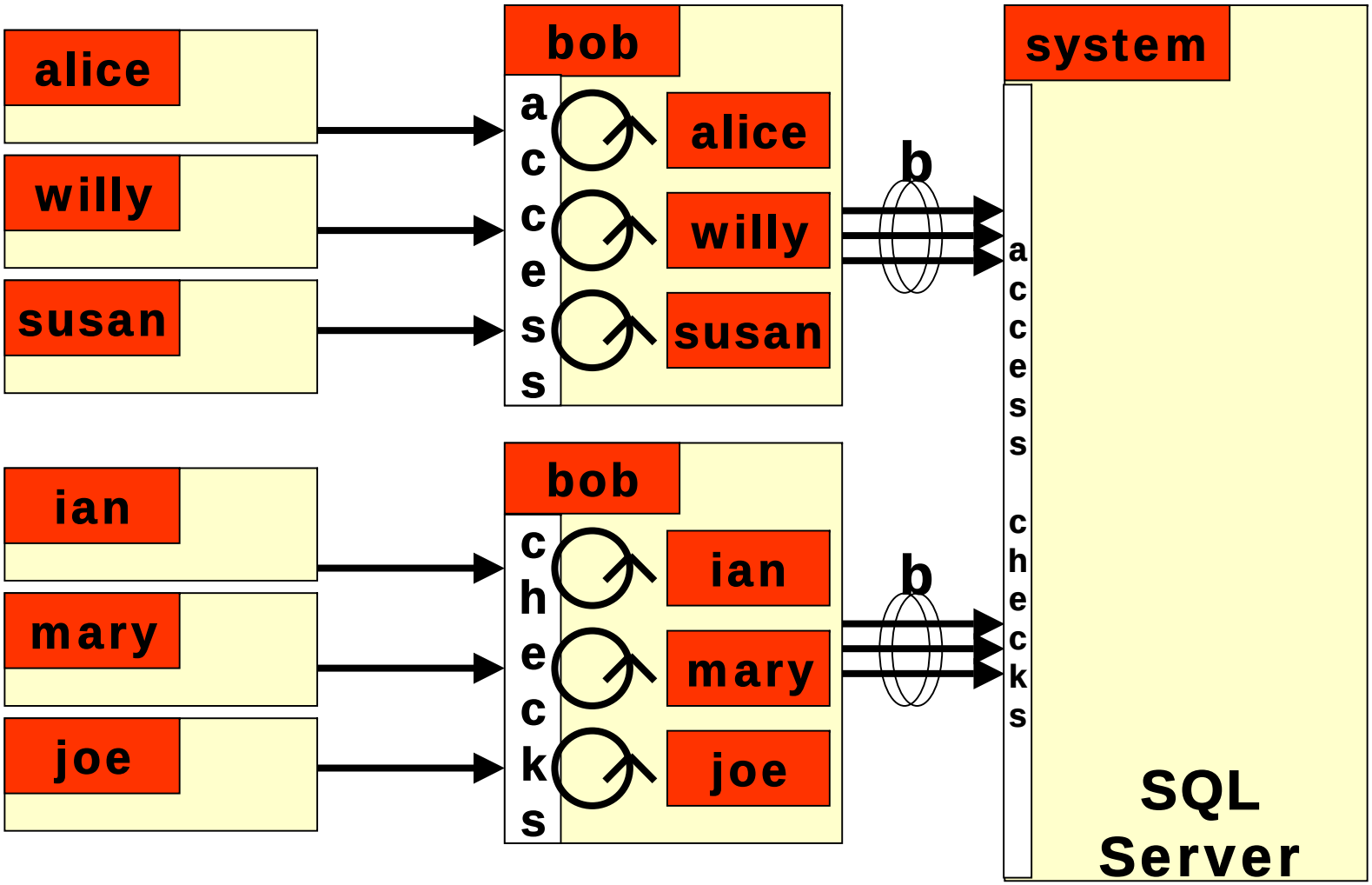
- **Most typical use of delegation is to allow access to databases**
- **But this flies in the face of connection pooling**
  - you must make a choice here
- **COM+ 1.0 was designed with this issue in mind**
  - COM+ role-based security authorizes requests in middle tier
  - middle tier accesses database under a single trusted identity
  - allows for connection pooling



Delegation, no connection pooling



Connection pooling



Authorization woes

- **The authorization problem**
  - Windows has always coupled authentication and authorization
  - group SIDs are collected during Kerb network authentication
  - this coupling is to avoid extra round-trips to authorities
- **It's expensive (infeasible, really) to discover groups manually**
  - need to visit both client's and server's authorities
  - esoteric features like group nesting, SID history, and domain quarantine make it really hard to do correctly in all cases
- **Thus we must authenticate using Kerb to discover groups**
  - but what if the client isn't able to use Kerberos?



Getting a token for the client

- **The safe way to discover group SIDs is to get a token**
  - traditionally this was done via LogonUser
  - but this required the server to know the client's password
  - example: IIS certificate mapping or basic authentication
- **Bad practice to have the server exposed to client passwords**
  - only authority should be privy to these
- **But how can we get a token without a password?**
  - enter protocol transition…

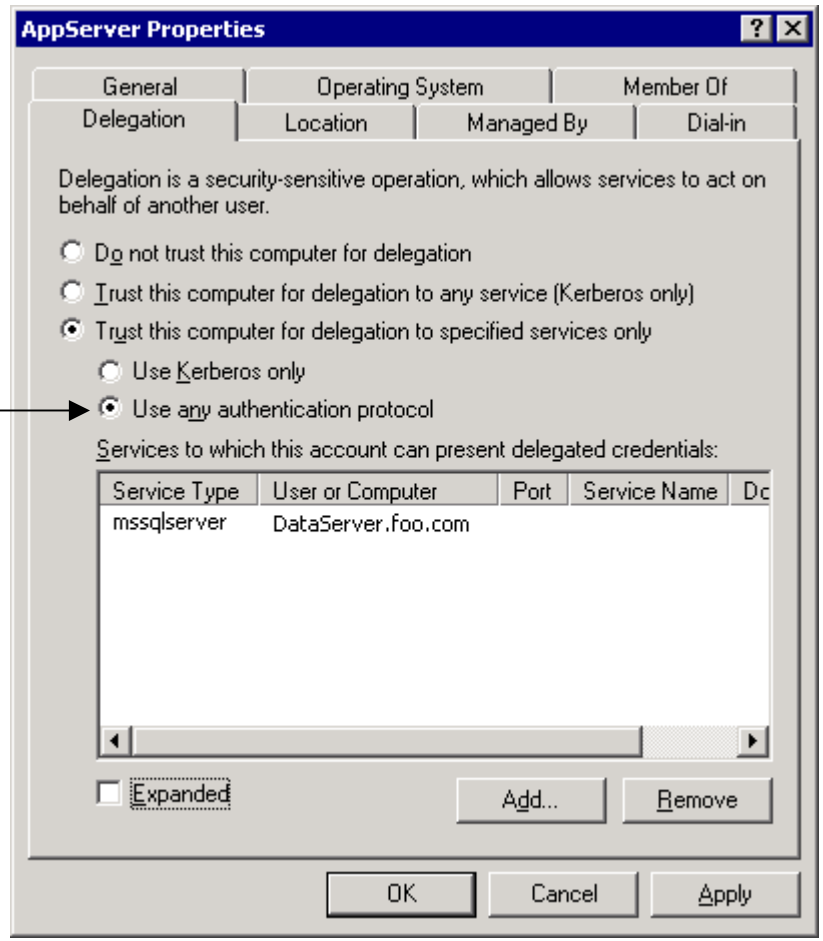


Windows Server 2003 protocol transition

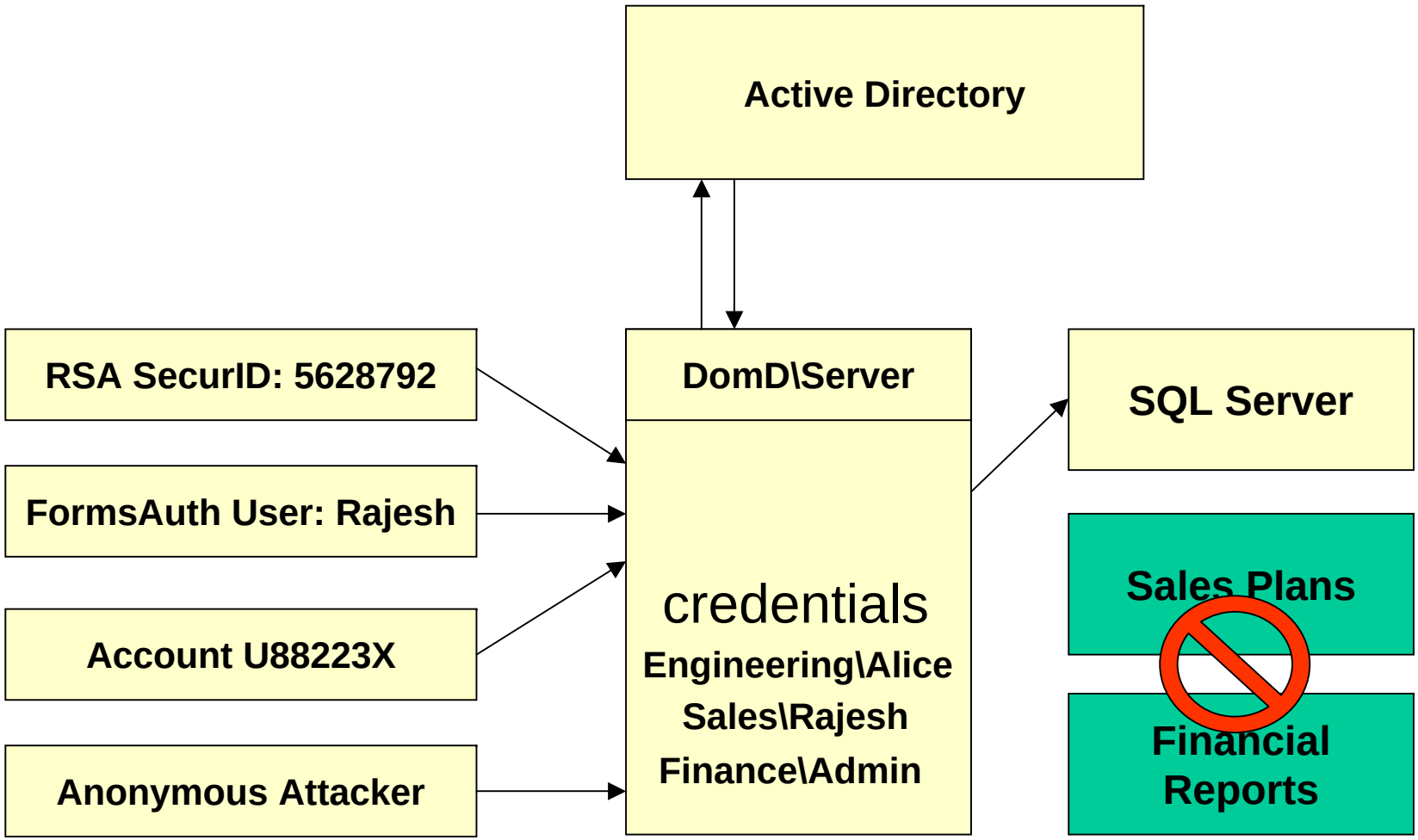
- **Protocol transition allows trusted servers to obtain tickets on a client's behalf**
  - without knowing the client's password
- **Tickets can be translated into tokens**
  - token can be used to discover groups
  - token can be impersonated to access local resources (service must be granted SeImpersonatePrivilege to do this)
- **Token may also be used to access network resources**
  - via constrained delegation
  - only possible if "allowed-to-delegate-to" list is populated
- **Thus the term "protocol transition"**
  - Server is trusted to transition from one authentication protocol (used to authenticate the client) into Kerberos on the back end

Configuring protocol transition

"Protocol transition"



Protocol transition, illustrated



Summary

- **Thread.CurrentPrincipal is a great way to track client identity**
- **Threads may have a token (think impersonation)**
- **Not all tokens have network credentials (think delegation)**
- **Windows 2000 only supports unconstrained delegation**
- **Windows Server 2003 supports constrained delegation and protocol transition**
- **For more information on security in the Windows operating system, see *Programming Windows Security***
  - http://www.develop.com/books/pws

Code Access Security

Securing mobile code in the .NET Framework

Part 1: motivation, permissions, and policy



Outline

- The problem of mobile code
- Code Access Security (CAS)
- Evidence
- Permissions
- Policy



What is mobile code?

- **Code not explicitly installed on local machine called *mobile***
  - typically obtained from a network
  - often delivered via web page
- **Mobile code packaging**
  - native code in the COM era
  - assembly in the .NET era
- **Mobile code sources**
  - your company's intranet
  - the public Internet

The old model: Full Trust

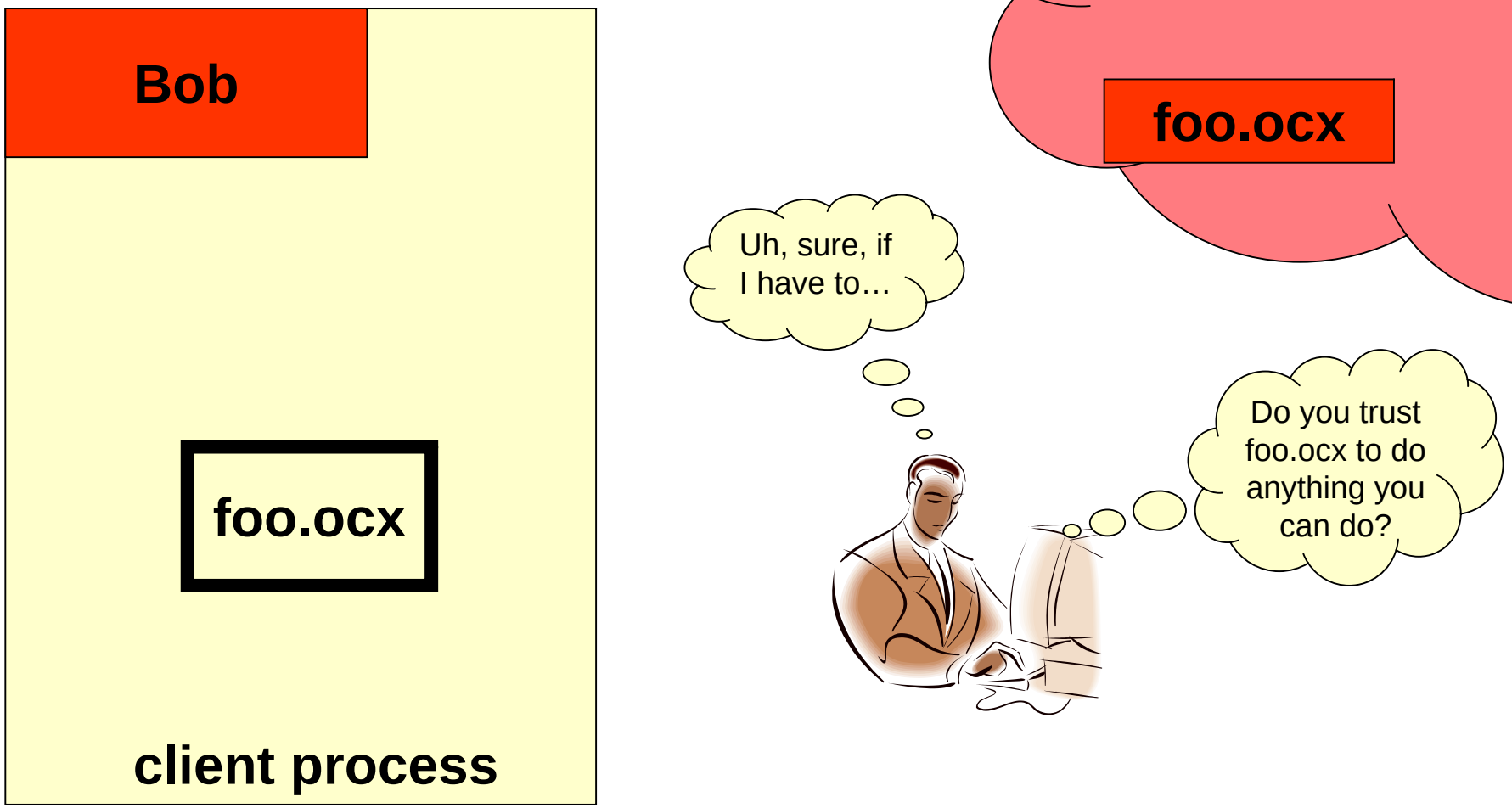
- **In the COM era there were only two options for mobile code**
  - full trust
  - completely untrusted
- **User had to decide on trust level at download time**
  - user presented with certificate
  - asked whether they trusted the vendor and their code
- **NO = mobile code not allowed to execute**
  - usually means websites don't work
- **YES = mobile code allowed to execute**
  - might give richer browsing experience
  - could run more buggy and vulnerable code base
  - could install virus or Trojan horse

The COM loader and trust levels

- **Every line of COM code runs with the same level of privilege within a single process**
  - COM code is packaged in native DLLs
  - DLL becomes an integral part of the process when loaded
  - can’t differentiate DLL code from original application code



The COM loader, illustrated

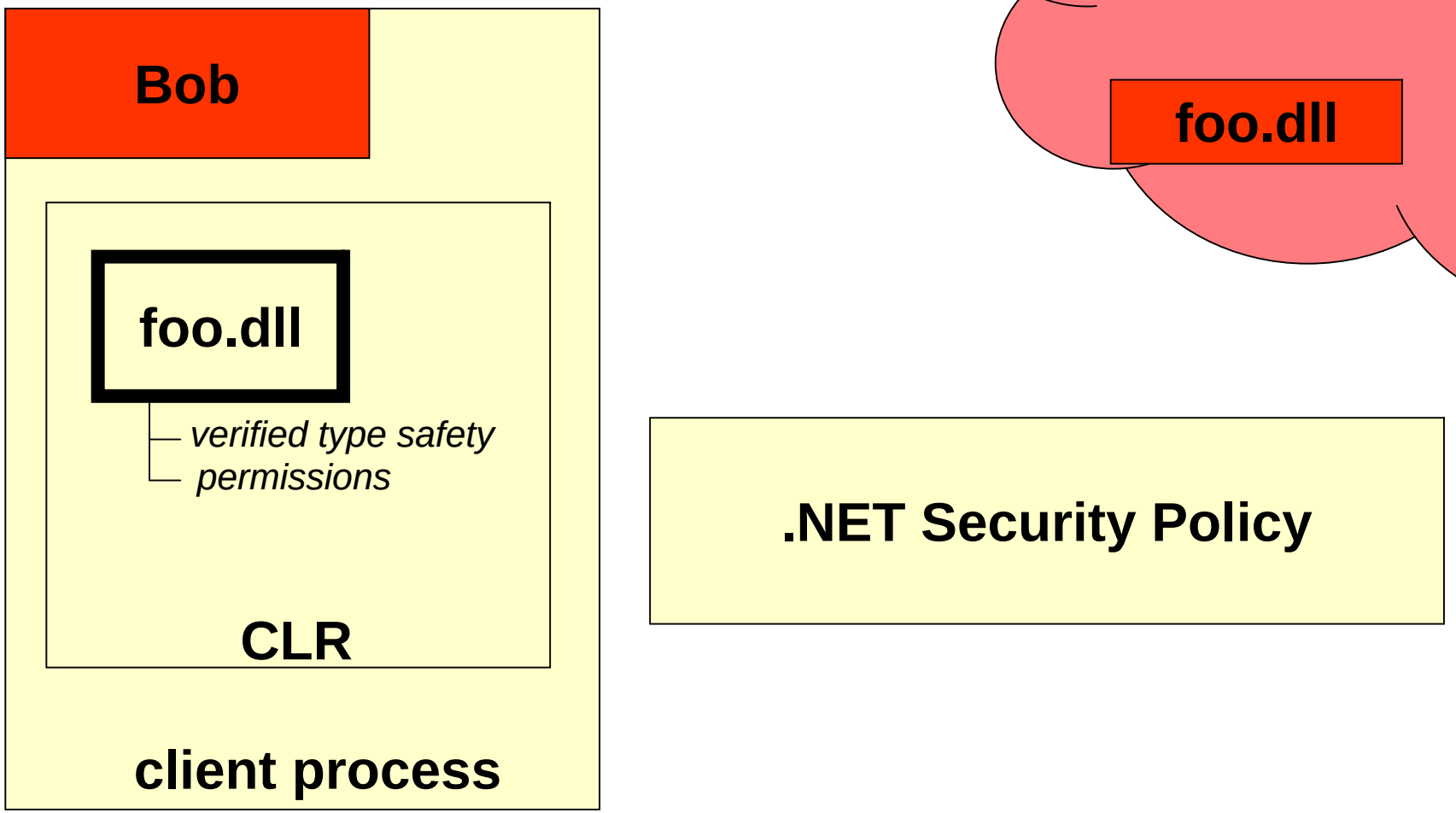


The .NET assembly loader and trust levels

- **Each .NET assembly may run with different privileges**
  - explicitly installed code usually trusted
  - mobile code typically restricted
- **.NET uses Code Access Security (CAS) model**
  - assembly loader gathers evidence like source and public key
  - security policy used to evaluate evidence
  - assembly permissions determined from evidence and policy
  - CLR restricts assembly to actions allowed by its permissions
  - user never asked trust questions



The .NET loader, illustrated



The importance of type safety

- **CLR enforces permissions to constrain mobile code**
  - possible because CLR type system is watertight
  - code is verified before execution to guarantee type safety
  - any bugs in the verification process compromises this system
- **COM based systems could not enforce type safety**
  - C++ allowed arbitrary casts
  - VB6 wasn't exempt either
  - no way to verify behavior of code



Type safety in C++

- **Without runtime type enforcement**
  - lots of funny business can go undetected at runtime
- **With runtime type enforcement**
  - invalid casts result in exceptions at runtime

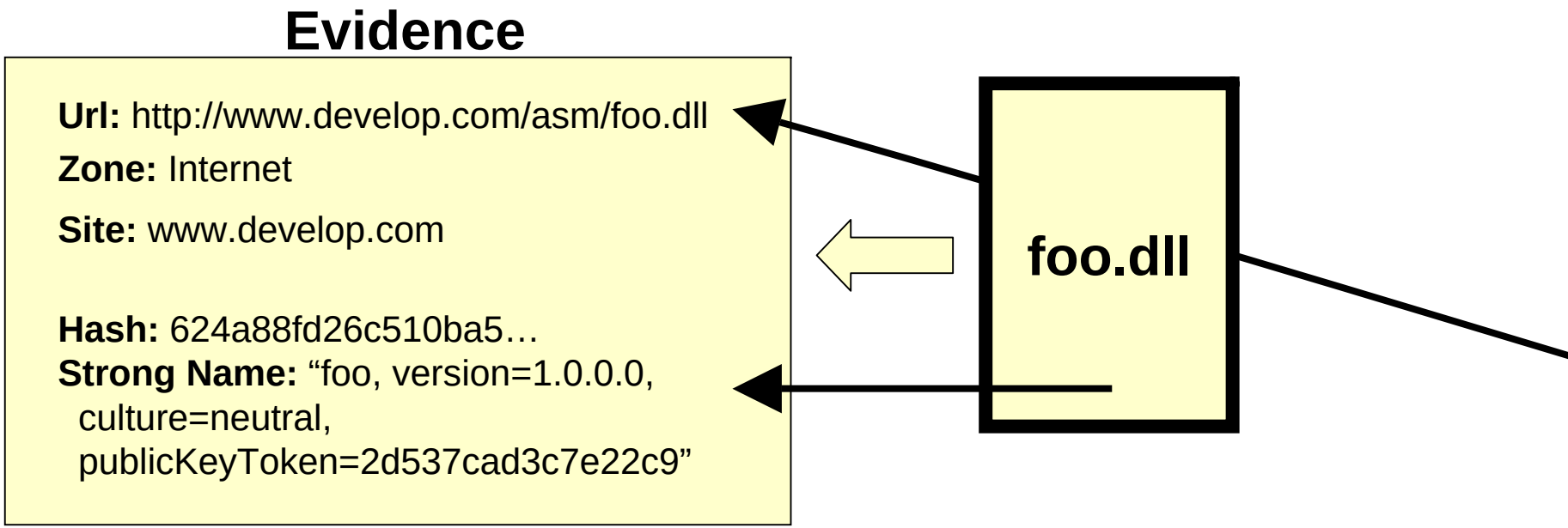
```
class DiskQuota {
  private long _maxBytes;
  // ...
};

void EvilCode(DiskQuota* pDiskQuota) {
  *(long*)pDiskQuota = MAX_LONG;
}
```




Evidence

- **Evidence comes from properties of assembly**
  - source: url, zone, site, application directory
  - author: strong name, publisher (Authenticode)
  - contents: hash



Evidence representation

- **FCL supplies classes used to represent evidence**
  - in **`System.Security.Policy`** namespace

```
Zone
Url
Site
ApplicationDirectory
StrongName
Publisher
Hash
```

Access to evidence

- **Evidence tracked with assembly**
  - available programmatically using reflection

```
class App {
  static void Main(string[] args) {
    Assembly a = Assembly.Load(args[0]);

    IEnumerator it = a.Evidence.GetEnumerator();

    // each item in this collection is an instance
    // of an evidence class (e.g., Site, Zone)
    while (it.MoveNext())
      Console.WriteLine(it.Current);
  }
}
```

Permissions

- **Permissions limit what an assembly can do**
  - run if code not verifiable?
  - access file system?
  - access the network?
  - access certain environment variables?
  - call native code (COM objects, DLLs)?
  - access files or printers without asking user?



CAS permission classes

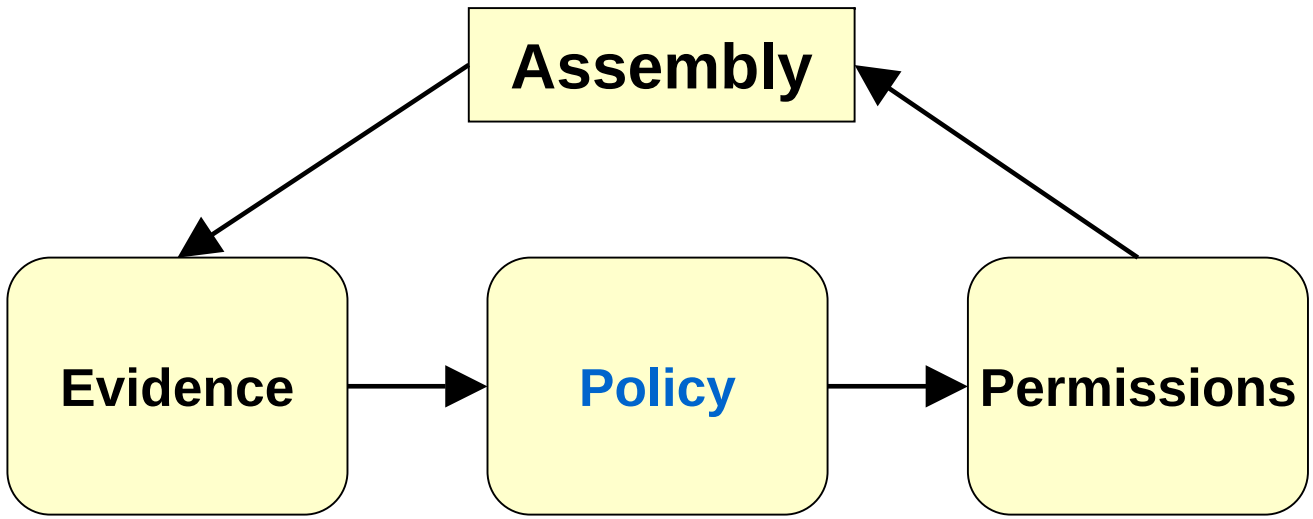
- **FCL supplies classes to represent permissions**
  - in **`System.Security.Permissions`** namespace

```
DBDataPermission
PrintingPermission
MessageQueuePermission
DnsPermission
SocketPermission
WebPermission
EnvironmentPermission
FileDialogPermission
FileIOPermission
IsolatedStorageFilePermission
ReflectionPermission
RegistryPermission
SecurityPermission
UIPermission
```



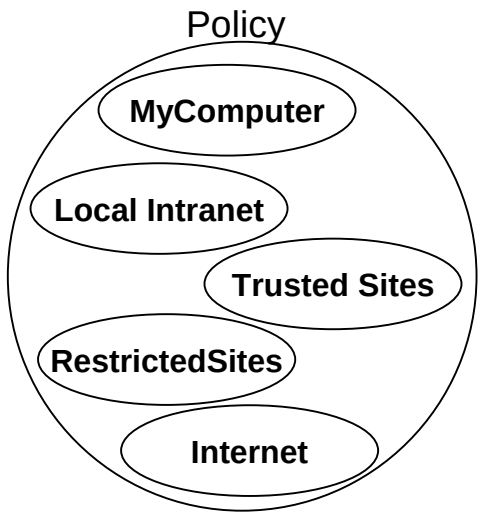

Security policy

- **Security policy determines permissions granted to assembly**
  - evidence given to policy
  - policy grants permissions based on the evidence
  - resulting permissions assigned to assembly



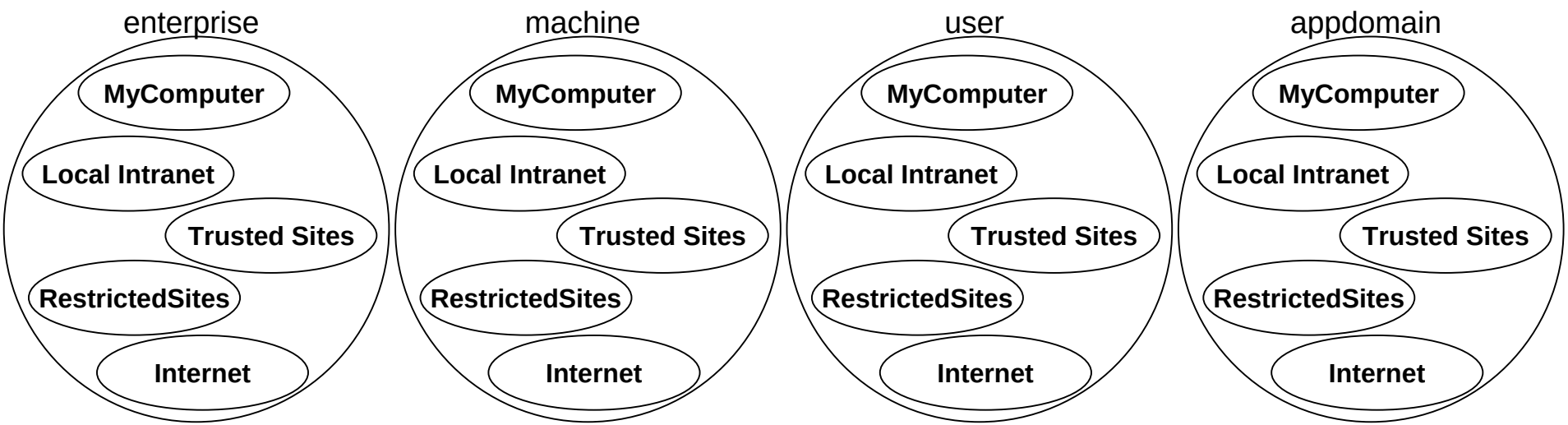
Setting policy

- **By default, policy is defined based on Internet Explorer zones**
  - My Computer
  - Local Intranet
  - Trusted Sites
  - Restricted Sites
  - Internet



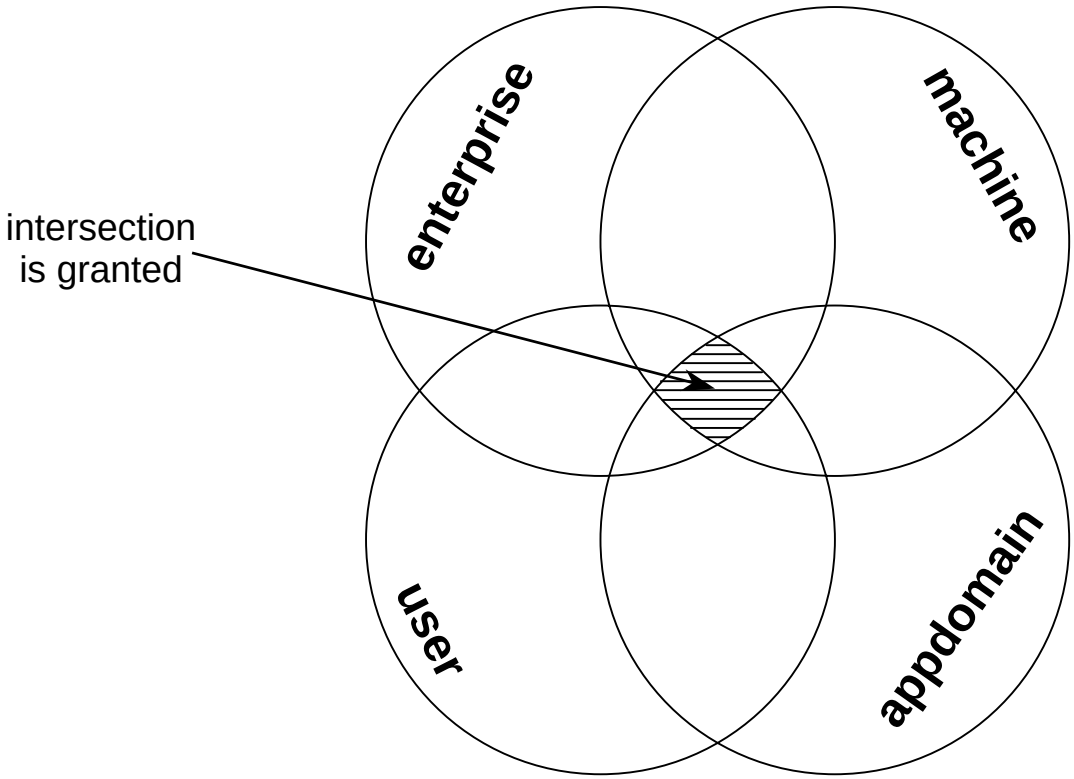
Policy levels

- **Policy comes from four sources**
  - enterprise
  - machine
  - user
  - appdomain



Policy interaction

- **Permission determined by intersecting applicable policies**
  - all policies must agree before a permission is granted



Default permissions

- **Default permission set for each policy source**
  - enterprise, user, appdomain grant full trust to all code
  - machine has several restrictions

default machine policy

My Computer	**Full Trust (no limitations at all)**
Local Intranet	**Medium Trust**
Trusted Sites	**Low Trust**
Internet	**Low Trust**
Restricted Sites	**Nothing (cannot execute)**

Customizing policy

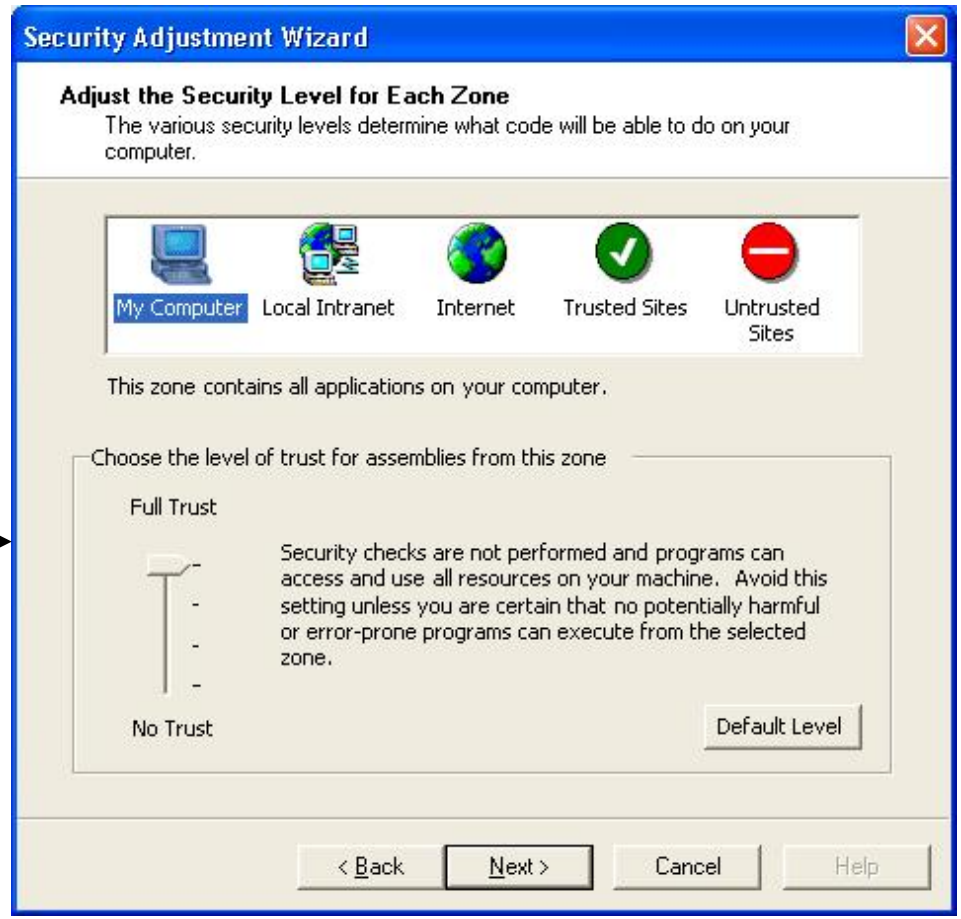
- **Can customize each policy level**
- **Framework supplies tools to simplify task**
  - predefined permission sets
  - wizards to select settings
  - editors for finer-grained control
- **Framework supplies policy classes**
  - System.Security.Policy namespace
  - can alter policy programmatically for maximum control



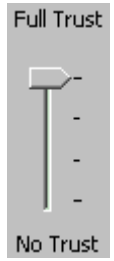
Using a wizard to adjust policy

- **Allows you to choose one of four permission sets**
  - Full Trust
  - Medium Trust
  - Low Trust
  - No Trust



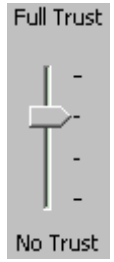


Full trust



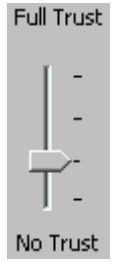
- **No CAS restrictions whatsoever**
- **Code doesn't have to be verifiable**
  - all code is verified by default unless it requests SkipVerification
- **Can call out to native code without restriction**
  - DLLs
  - COM objects
- **Don't forget that the operating system still controls access to external resources based on tokens**
  - File system, kernel objects, etc.
  - DCOM
  - Databases

Medium trust



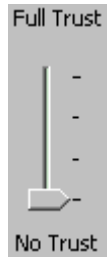
- Code may execute.
- Code must be verifiable and may not call directly into unmanaged code.
- Allowed to assert permissions (more on this later).
- May not suspend, resume, interrupt, abort other threads, or stop its own threads from being aborted by more trusted code.
- May not modify security policy.
- May not change the managed principal on a thread.
- May not create or control AppDomains.
- May not serialize objects using a SerializationFormatter (allows access to private state).
- May not inject evidence for assemblies or AppDomains.
- May not configure System.Runtime.Remoting or add extensions such as remoting sinks.
- Denied access to all environment variables, except you may read USERNAME.
- May read or write to the local file system, but only by using open/save dialogs to get a stream (exception: may read directly from AppBase directory and any subdirectories)
- Unlimited isolated storage space for your assembly, scoped by user, assembly, and machine.
- May use reflection, but only to access public members of public types or to emit new types.
- May read and write from the clipboard and put up windows of any shape or size.
- May use DNS (Domain Name Service) without restriction.
- May submit print jobs directly to the default printer, or to any other printer using a common dialog.
- May read or write to existing event logs, and create event sources and logs.
- May not access the registry, SQL databases, message queues
- May use sockets, but only to connect to the site from which the code originated

Low trust



- Same as Medium Trust, with the following further restrictions:
- May not assert permissions.
- May not read/write ANY environment variables.
- May not use the FileSave dialog to open files for writing (can still use FileOpen to read).
- May not read (via File IO) from AppBase directly without a dialog.
- Isolated storage further restricted by quota (10240 bytes), also must be scoped not only by assembly and user, but also by application.
- May only display “safe top-level windows” which is supposed to help keep mobile code from spoofing login dialogs and the like.
- While writing to the clipboard is unrestricted, only intrinsic controls such as the TextBox may read from the clipboard – user controls cannot.
- Cannot use reflection to emit code.
- Cannot use DNS (Domain Name Service) to resolve names.
- May not send print jobs directly to any printers. Must use the common print dialog in order to obtain the user’s consent.
- Only code from trusted sites can connect back to the original site

No Trust



- No permissions are granted
- Code is not even allowed to execute

Named permission sets

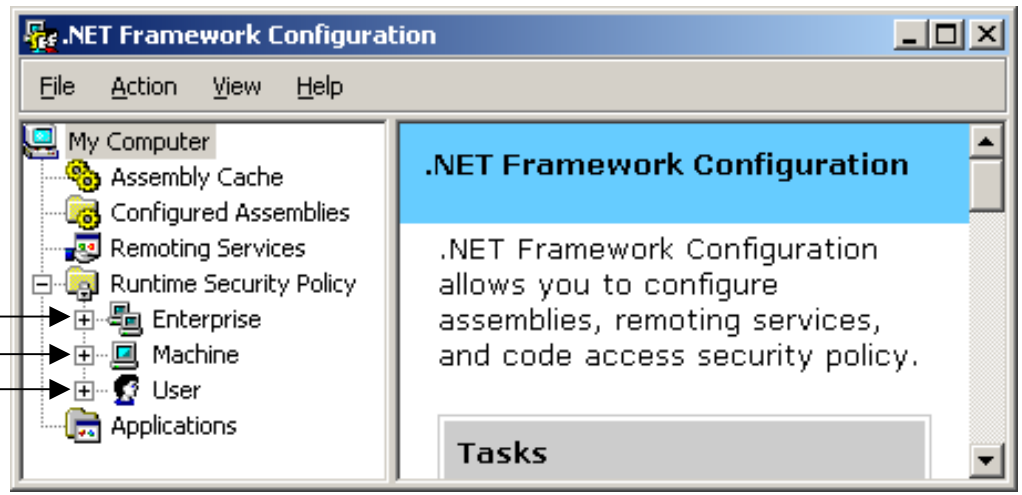
- **Permissions are organized into sets to ease administration**
- **Four built-in permission sets used by the wizards**
  - FullTrust
  - LocalIntranet (a.k.a. “medium trust”)
  - Internet (a.k.a. “low trust”)
  - Nothing (a.k.a. “no trust”)
- **Three other built-in permission sets**
  - Everything (all permissions defined by Microsoft)
  - SkipVerification (grants the right to run unverifiable code)
  - Execution (grants the right to execute)
- **Can create others...**



Drilling into security policy

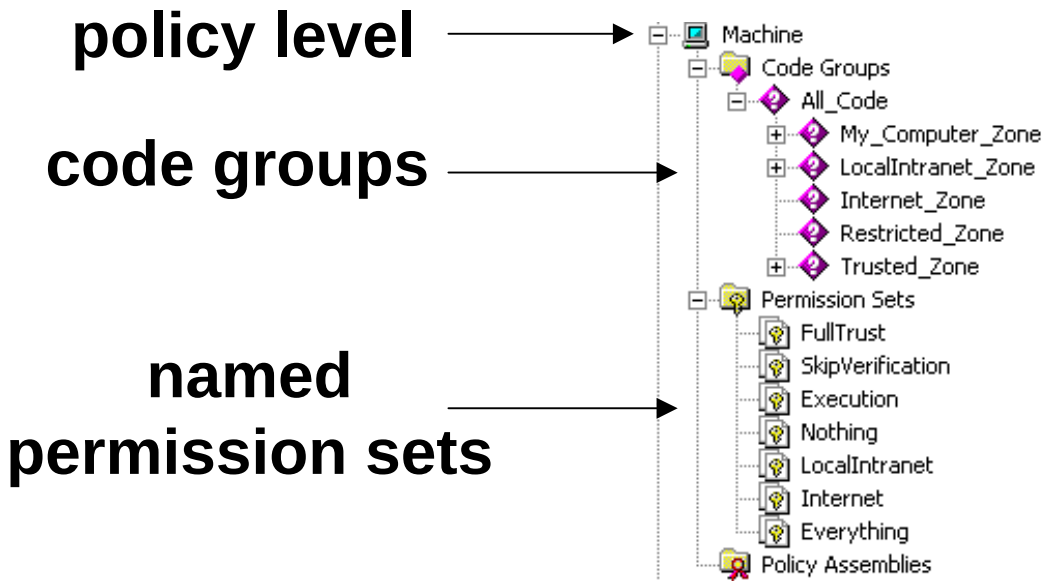
- **Policy is organized in several levels**
  - allows delegation of administration
  - restrictions at one level cannot be overridden in a lower level
- **View/edit using .NET Framework Configuration MMC snapin**

policy levels



Drilling into a policy level

- A policy level consists of a tree of code groups and a list of named permission sets



Code groups

- **Each code group has two properties**
  - a membership condition (a single test of the evidence)
  - a reference to a named permission set
- **Permission grant for a policy level comes from the union of all matching code groups**
  - child nodes evaluated only if parent node matches

Evaluating a policy level

evidence:

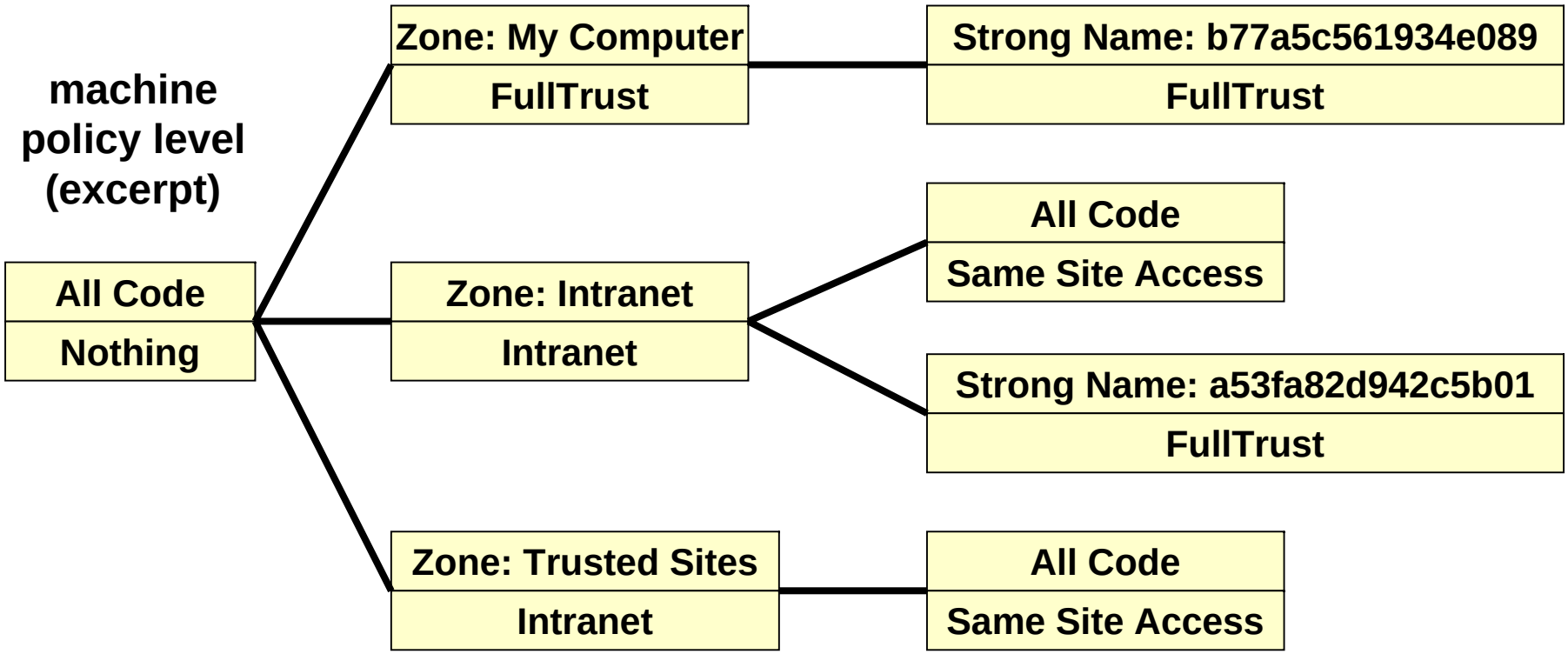
URL: http://sales/routing.dll
Zone: Intranet
StrongName: a53fa82d942c5b01

permission grants:

Nothing
Intranet
Same Site Access
FullTrust

= FullTrust

machine policy level (excerpt)



Assembly permission requests

- **Permissions gathered from policy specify the "maxgrant" for the assembly**
  - This set of permissions can be filtered by the assembly itself

```
// refusing a permission explicitly
[assembly: FileIOPermission(SecurityAction.RequestRefuse,
                            Unrestricted=true)]
```

```
// requesting minimal + optional permissions
// (any permissions not requested are implicitly refused)
[assembly: PermissionSet(SecurityAction.RequestMinimum,
                         Name="Internet")]
[assembly: PermissionSet(SecurityAction.RequestOptional,
                         File="optional.xml")]
```

Assembly permission requests

- **Minimum permission set**
  - if maxgrant doesn't include all these permissions, fail to load
  - this is an easy way to make sure that the core functionality in the assembly can be provided without security exceptions
- **Optional permission set**
  - permissions the assembly is willing to accept, but doesn't absolutely need to perform its core functionality
- **Refused permission set**
  - permissions the assembly absolutely does not want
- **Permission requests can break your code if you're not careful**
  - very often turns a fully trusted assembly into a partially trusted assembly
  - more on this later



Viewing assembly permission requests

permview.exe is your friend

```
C:\>permview \windows\microsoft.net\...\mscorlib.dll

minimal permission set:
<PermissionSet class="System.Security.PermissionSet"
               version="1">
   <IPermission class="System.Security.Permissions.SecurityPermission"
                version="1"
                Flags="SkipVerification"/>
</PermissionSet>

optional permission set:
  Not specified

refused permission set:
  Not specified
```

Programmatically adjusting policy

- **Often the policy wizard is a bit heavy handed**
  - zone-wide adjustments
  - even with further refinement, only have four choices for trust
    - Full, medium, low, or none
- **Can edit code groups and permission sets via UI**
  - automating this almost always necessary
- **For full control, can program against policy manually**
  - use SecurityManager to discover policy levels
  - use NamedPermissionSet to add new permission sets
  - use UnionCodeGroup to add new code groups
  - don't forget to save your changes via SecurityManager.SavePolicyLevel

Example: programmatically adjusting policy

first need to find the policy level you plan on adjusting (typically machine policy)

```
PolicyLevel machineLevel = null;
IEnumerator policyLevelEnumerator = SecurityManager.PolicyHierarchy();
while (policyLevelEnumerator.MoveNext()) {
    PolicyLevel lvl = (PolicyLevel)policyLevelEnumerator.Current;
    if ("Machine" == lvl.Label) {
        machineLevel = lvl;
        break;
    }
}
```

Example: programmatically adjusting policy

add a named permission set that describes whatever permissions you need

```
NamedPermissionSet nps = new NamedPermissionSet(
  "AcmeExpense Permissions", PermissionState.None);

nps.AddPermission(new FileIOPermission(PermissionState.Unrestricted));
if (null != machineLevel.GetNamedPermissionSet(nps.Name)) {
    machineLevel.ChangeNamedPermissionSet(nps.Name, nps);
}
else {
    machineLevel.AddNamedPermissionSet(nps);
}
```

Example: programmatically adjusting policy

add a code group that associates a membership condition with your named permission set

use secutil.exe to get public keys…

```
CodeGroup myCodeGroup = new UnionCodeGroup(
    new StrongNameMembershipCondition(
        new StrongNamePublicKeyBlob(pubkey),
        "AcmeExpense",
        null),
    new PolicyStatement(nps));
myCodeGroup.Name = "AcmeExpense Application";
myCodeGroup.Description = "Grants the AcmeExpense app access to ...";

machineLevel.RootCodeGroup.AddChild(myCodeGroup);
SecurityManager.SavePolicyLevel(machineLevel);
```

save the policy level when you're done

Extracting a public key into a byte array

```
C:\>secutil –s someSignedAssembly.dll

Microsoft (R) .NET Framework SecUtil 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Public Key =
{ 0, 36, 0, 0, 4, 128, 0, 0, 148, 0, 0, 0, 6, 2, 0, 0, 0, 36, 0, 0, 82,
83, 65, 49, 0, 4, 0, 0, 1, 0, 1, 0, 39, 248, 152, 209, 178, 241, 251,
168, 253, 210, 19, 210, 216, 222, 225, 229, 140, 57, 195, 183, 124,
162, 125, 84, 69, 41, 216, 50, 235, 108, 84, 140, 112, 4, 244, 1, 169,
135, 9, 176, 43, 34, 138, 250, 129, 52, 71, 209, 167, 94, 166, 18, 124,
230, 36, 132, 190, 79, 63, 162, 166, 103, 81, 129, 131, 182, 63, 79,
78, 136, 246, 15, 137, 91, 174, 76, 220, 67, 246, 187, 66, 171, 250,
78, 127, 248, 23, 153, 124, 113, 14, 195, 150, 176, 230, 158, 85, 244,
152, 111, 109, 11, 177, 96, 241, 114, 2, 49, 235, 42, 186, 65, 24, 215,
109, 174, 38, 162, 2, 162, 6, 136, 2, 2, 154, 207, 4, 177 }
Name =
someSignedAssembly
Version =
1.0.982.19355
Success
```

CAS can be completely disabled on a machine

- **You should be aware of this**
- **Requires admin privileges**
  - caspol -s off
- **Can turn code access security back on via a similar command**
  - caspol -s on
- **Can check programmatically**

```
using System;
using System.Security;

class CheckSecurity {
  static void Main() {
    Console.WriteLine(".NET Code Access Security is currently {0}",
      SecurityManager.SecurityEnabled ? "enabled" : "DISABLED!!");
  }
}
```

Policy wrap-up

- **Loader discovers evidence**
- **Evidence is fed as input into policy**
- **Each of the four policy levels are evaluated, one at a time**
  - union of permission grants for each matching code group
- **Intersect the permissions granted at each level**
- **Apply any assembly permission requests**
- **Result is permissions granted to assembly**



Summary

- CAS is the .NET security model for dealing with mobile code
- Evidence is discovered by the loader
- Policy takes evidence and turns it into permissions
- Permissions say what your code can and cannot do

- …no point to any of this if CAS is disabled on the machine



Code Access Security

Securing mobile code in the .NET Framework
Part 2: enforcement, tips, and custom permissions



Outline

- Enforcement
- Tips for writing mobile code
- Using Isolated Storage
- Applying CAS to local code
- Implementing custom permissions



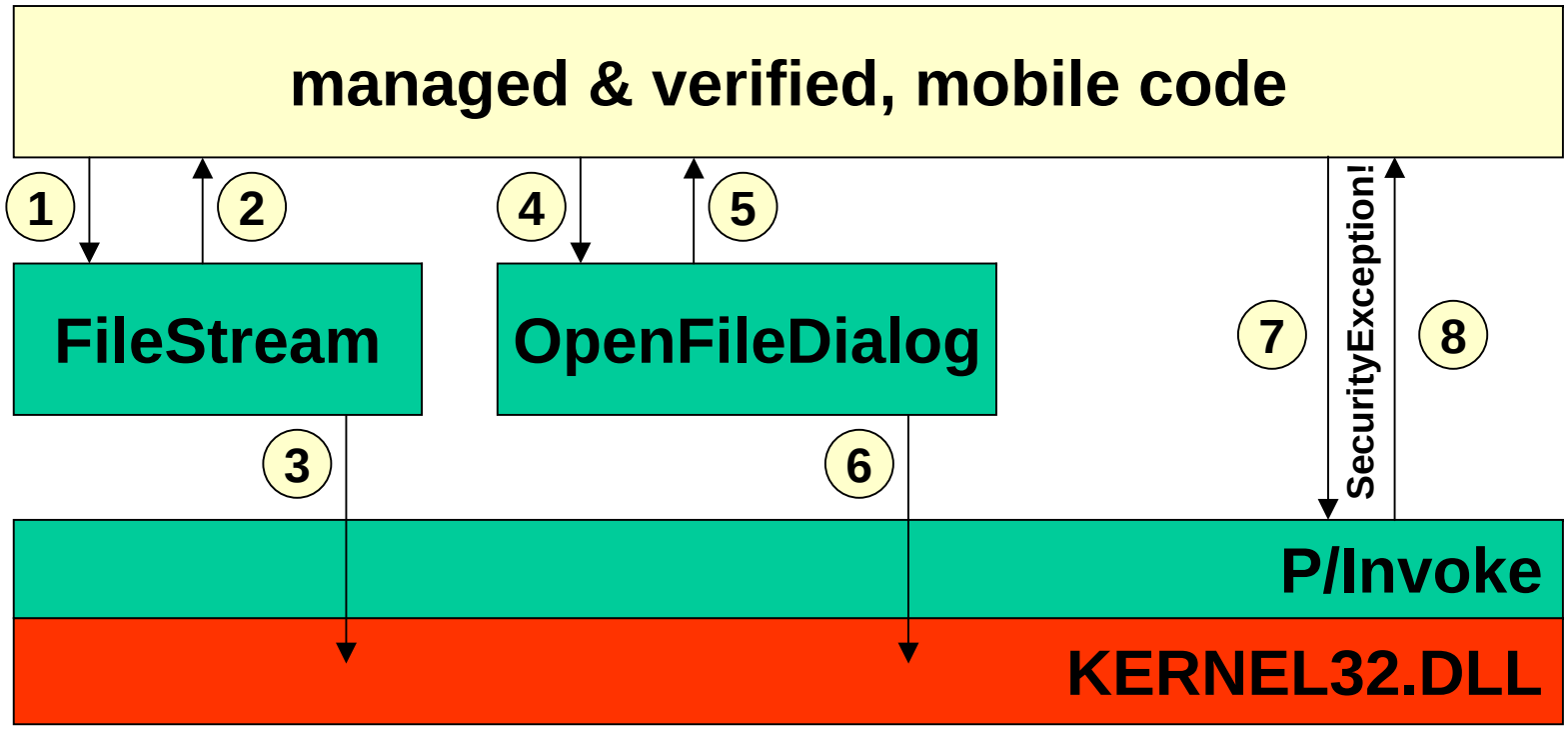
Enforcing security

- **At runtime, the .NET Framework classes demand permissions before performing sensitive operations**
  - Type safe managed code can't get around these checks
  - Mobile code restricted from calling unmanaged code directly
- **Demands may be implemented two ways**
  - Imperative – write the code to call Demand()
  - Declarative – use an attribute to force a demand



Enforcing security

2 demand FileIOPermission
5 demand FileDialogPermission
8 demand SecurityPermission



1 `new FileStream(@"c:\temp\foo.xml");`

4 `openFileDialog.ShowDialog();`

7 `CreateFile(@"c:\temp\foo.xml", ...);`

Example: the file stream constructor

```
// excerpt from mscorlib.dll
public FileStream(string path, FileAccess desiredAccess) {
    FileIOPermissionAccess access = _calcAccess(desiredAccess);

    // describe the action we are going to perform
    CodeAccessPermission perm = new FileIOPermission(access,
                                                     path);
    // demand that the caller can do this
    perm.Demand();

    // call through interop to get a real file handle
    Win32Native.CreateFile(...);
}
```

This is an “imperative” demand – note we are calling Demand() programmatically

Example: declarative demands

```
// excerpt from mscorlib.dll
namespace System.Threading {
  public class Thread {
    [SecurityPermission(SecurityAction.Demand,
                        ControlThread=true)]
    public void Suspend() {
      this.SuspendInternal();  // actually do the work
    }
  }
}
```

Other types of demands

- **LinkDemand**
  - Checks at JIT time whether calling assembly satisfies demand
  - Often used to deny access to mobile code
    - LinkDemand that caller has full trust
  - Often used to constrain caller by strong name
    - Allows types to be exposed publicly from an assembly while constraining who can use them
- **InheritanceDemand**
  - Checks at load time whether derived class satisfies demand
  - Same uses as LinkDemand, but helps constrain who can override methods
- **Both typically make use of “identity permissions”**

Example: LinkDemand

```
[StrongNameIdentityPermission(SecurityAction.LinkDemand,
  PublicKey="0024000004800000940000006020000" +
            "00240000525341310004000001000100" +
            "6F8DC651FF981820321523DB748F4EB7" +
            "0E08C658CB37D355A81A3162B8BB2440" +
            "1AF243F0C698623CD3A0916B3055F1C9" +
            "9148F350D4750AF7231245CD54761964" +
            "0C21F6EE3633EC0D44C708EA50A7010D" +
            "15521719C33D1BBD5987AE9930B35637" +
            "9DBB7A5367592046E5AEC0725623B378" +
            "04566E5BC92B5E9508CE19DB49FEDBD3")]
public class AcmeHelper {
  public void DoSomethingHelpful() {
    // this class can only be called by code
    // that has been signed by the private key
    // associated with the above public key
    // (or someone hacked their CLR to skip this check)
  }
}
```

Viewing declarative demands, asserts, etc.

permview.exe is your friend, once again

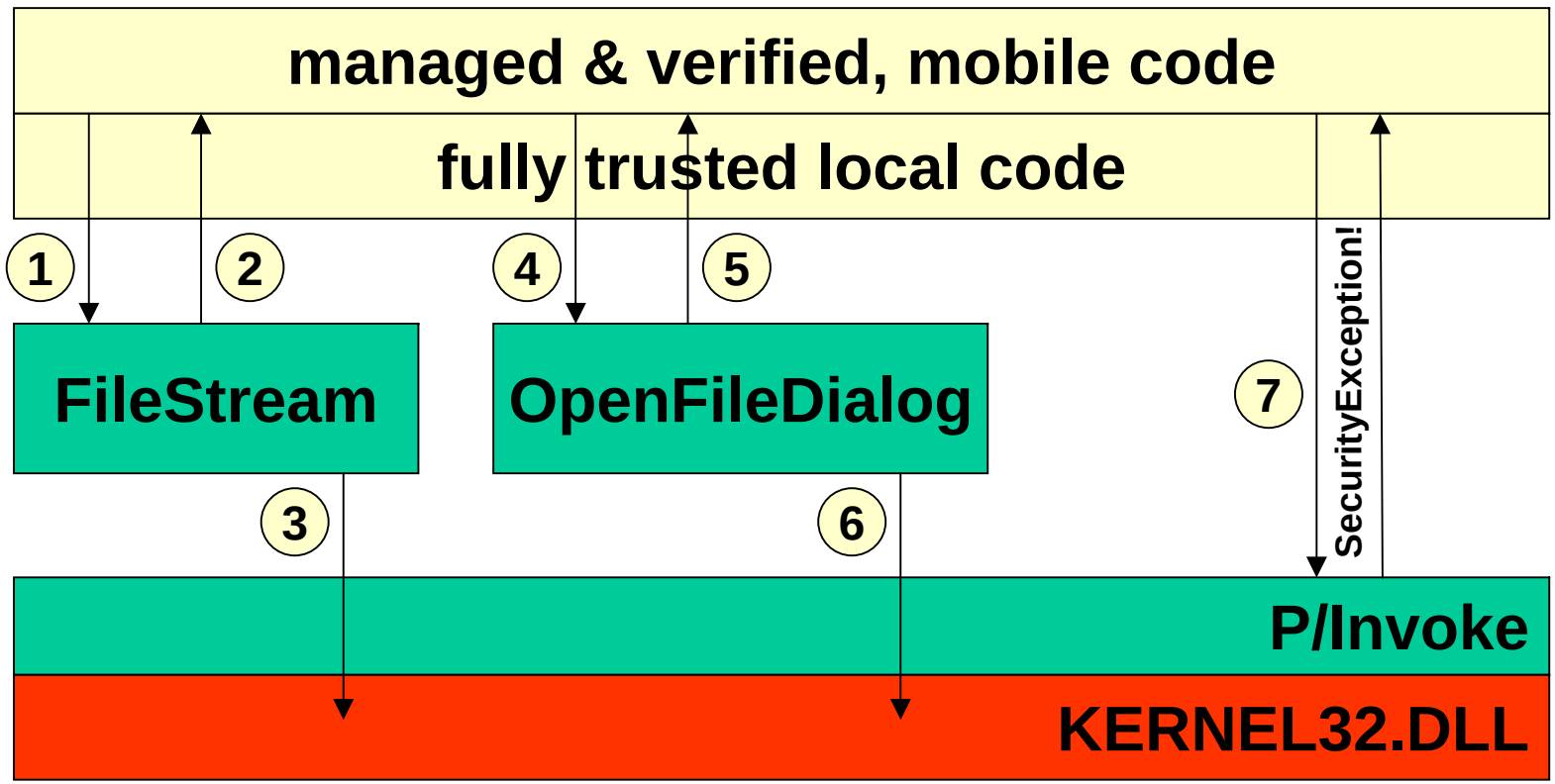
```
C:\>permview /decl \windows\microsoft.net\...\mscorlib.dll

Method System.Threading.Thread::Suspend() Demand permission set:
<PermissionSet class="System.Security.PermissionSet"
               version="1">
   <IPermission class="System.Security.Permissions.SecurityPermission"
                version="1"
                Flags="ControlThread"/>
</PermissionSet>

Method System.AppDomain::SetData() LinktimeDemand permission set:
<PermissionSet class="System.Security.PermissionSet"
               version="1">
   <IPermission class="System.Security.Permissions.SecurityPermission"
                version="1"
                Flags="ControlAppDomain"/>
</PermissionSet>
...
```

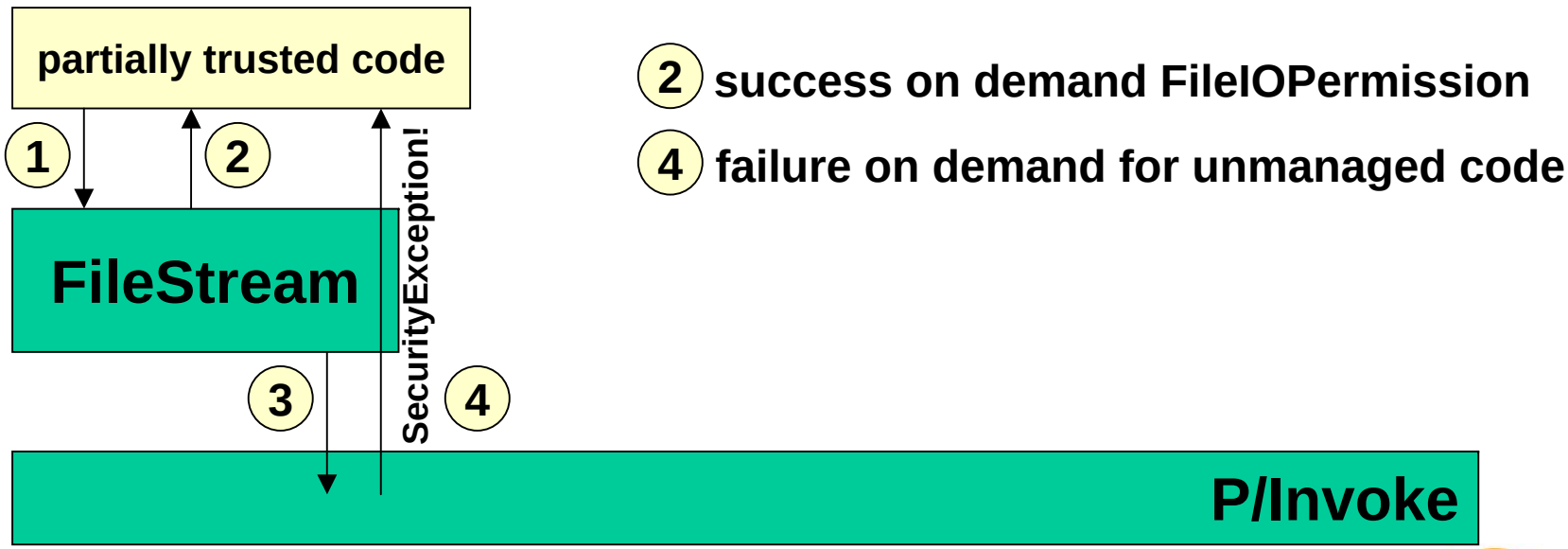
Security demands walk the stack

- **Prevents less trusted components from "luring" more trusted components into doing their dirty work**
  - (note that link time demands are an exception to this rule)



The stack walk is almost always a good thing

- **Except when you're trying to build a secure gateway**
  - such as the FileStream object
- **If the stack walk wasn't controllable, creating a FileStream would actually require *two permissions*, not just one**
  - FileIOPermission demanded by FileStream
  - SecurityPermission(UnmanagedCode) demanded by P/Invoke



Controlling the stack walk

- **For calling to native code (most common)**
  - use SuppressUnmanagedCodeSecurityAttribute
  - suppresses the stack-walking demand for UnmanagedCode, turning it into a LinkDemand that only affects the direct caller
  - thus direct calling assembly must have UnmanagedCode perm
  - important performance optimization!
- **For all other cases**
  - use Assert
  - an assembly that calls Assert must have the permission it is asserting, and must also have SecurityPermission(Assertion)
  - Stack walk still occurs, it's just halted when assertion marker is found on the stack

Example: SuppressUnmanagedCodeSecurity

```
// excerpt from mscorlib.dll
public FileStream(string path, FileAccess desiredAccess) {
    FileIOPermissionAccess access = _calcAccess(desiredAccess);

    // describe the action we are going to perform
    CodeAccessPermission perm = new FileIOPermission(access,
                                                     path);
    // demand that the caller can do this
    perm.Demand();

    // call through interop to get a real file handle
    // note that no stack walk will occur!
    Win32Native.CreateFile(...);
}
```

```
// also in mscorlib.dll
[SuppressUnmanagedCodeSecurity]
internal class Win32Native {
    [DllImport("kernel32.dll")]
    internal static extern int Createfile(...);
}
```

Example: Assert

```
// this function needs to read an environment variable in order to
// do its work, but callers might not have permission to read FOO
// themselves
public static void DoSomeWork() {
  CodeAccessPermission perm = new EnvironmentPermission(
                      EnvironmentPermissionAccess.Read,
                      "FOO");

  // For this to work, this assembly (not its callers) must have
  //  1) the permission we are trying to assert
  //  2) the right to assert (SecurityPermission(Assertion))
  perm.Assert();

  string foo = Environment.GetEnvironmentVariable("FOO");

  // do some more work...
}
```

Don’t misuse Assert and SUCS

dumb code alert

```
// this totally subverts security policy using Assert
public static string ReadEnvironmentVariable(string name) {
  new EnvironmentPermission(EnvironmentPermissionAccess.Read,
                            name).Assert();

  return Environment.GetEnvironmentVariable(name);
}
```

```
// subverts security policy via SuppressUnmanagedCodeSecurity
public static string CallSleep(int milliseconds) {
  Sleep(milliseconds);
}

[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
internal static extern void Sleep(int milliseconds);
```

dumb code alert

CAS stack markers: Assert, Deny, PermitOnly

- **An assertion is really just a marker on the stack**
  - when an assertion marker is encountered during a stack walk, if the asserting assembly has the permission being asserted, that permission is granted immediately (the stack walk ends)
- **Deny and PermitOnly markers work similarly**
  - when a deny marker is encountered during a stack walk, the permission will be denied immediately (the stack walk ends)
- **Deny and PermitOnly can be used to choke down permissions before calling to less trusted code**
  - helps secure extensibility points in your application
- **Deny and PermitOnly are helpless against fully trusted code**
  - fully trusted code can just Assert and party on



Partially trusted code

- **Partially trusted code can't call strongly named assemblies**
  - not unless the target assembly explicitly allows it
  - local assemblies without strong names cannot be seen by typical mobile code (browser's AppBase is nowhere near your code)
  - assemblies with strong names can be GACked, thus the restriction
- **Why worry?**
- **Isn't CAS strong enough to allow this?**

What if mobile code could use this class?

dumb code alert

```
public class HopeThisIsntUsedByEvilCodeBecauseItsNotRobust {
  public string Name;
  public string Password;

  // this entire function consists of
  // horribly bad code you should NEVER EVER use
  public bool IsValidUser() {
    new DbDataPermission(PermissionState.Unrestricted).Assert();
    SqlConnection conn = new SqlConnection(
      "initial catalog=accounts;user id=sa;password=;")
    conn.Open();
    cmd.CommandText = string.Format(
      "select count(*) from users where name='{0}'" +
      "and password='{1}'", Name, Password);
    return ((int)cmd.ExecuteScalar()) > 0;
  }
}
```

dumb code alert

How to allow partially trusted code to call you

- **Step 1: review your code for security holes**
- **Step 2: fix the holes**
- **Step 3: goto step 1 until you're really comfortable**
  - Microsoft is still in this loop with many of its assemblies
- **Step 4: apply an attribute to your strongly named assembly**
  - System.Security.AllowPartiallyTrustedCallersAttribute
  - step 4 is much, much easier than steps 1-3, so be vigilant

```
using System.Reflection;
using System.Security;
[assembly: AssemblyKeyFile("publicKey")]
[assembly: AssemblyDelaySign(true)]
[assembly: AllowPartiallyTrustedCallers]
```

Applying CAS to non-mobile code

- **Why not use CAS for local code?**
  - it's a good idea in theory
  - each assembly runs with only the permissions it needs
- **In practice there is a major roadblock**
  - most system assemblies aren't marked with AllowPartiallyTrustedCallersAttribute
- **AllowPartiallyTrustedCallersAttribute not present on:**
  - System.Runtime.Remoting.dll
  - System.Management.dll
  - and others…
- **Microsoft is working on it…**
  - for now, beware restricting local code with CAS



Tips for writing mobile Intranet code

- **Use only managed, verifiable code**
  - don't use Managed C++ or C# /unsafe switch
- **Avoid System.Runtime.InteropServices**
  - that is, don't call to COM objects or unmanaged DLLs
- **Avoid using System.Runtime.Serialization**
  - consider using System.Xml.Serialization instead
- **Avoid using System.Runtime.Remoting**
  - consider using System.Web.Services instead
- **Use common dialogs**
  - OpenFileDialog, SaveFileDialog, PrintDialog
- **Only reflect against accessible members**
  - don't try to use reflection to get to private stuff
- **Use AllowPartiallyTrustedCallersAttribute where necessary**
- **Consider using Isolated Storage…**



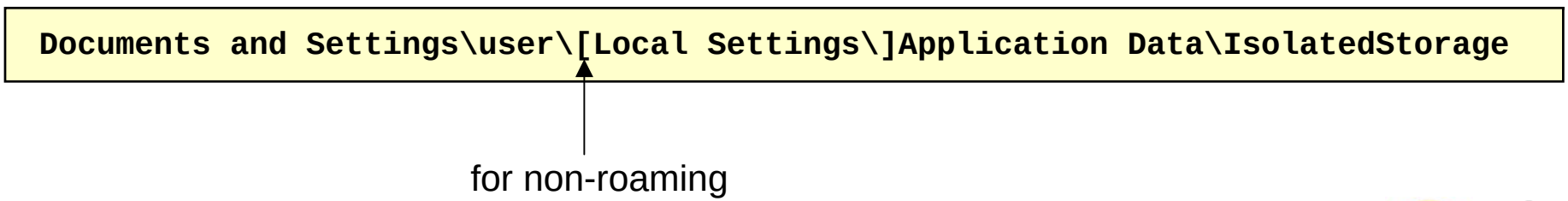
Isolated storage

- **Allowing partially trusted code to specify file names and paths is dangerous**
  - past experience tells us this*
- **The .NET Framework provides a virtualized file system**
  - tucked away in a protected directory in the real file system
- **Goal is to allow partially trusted code to persist data without exposing the real file system**
- **There are actually several isolated stores**
  - each provides a virtualized file system
  - code chooses a store based on scope requirements

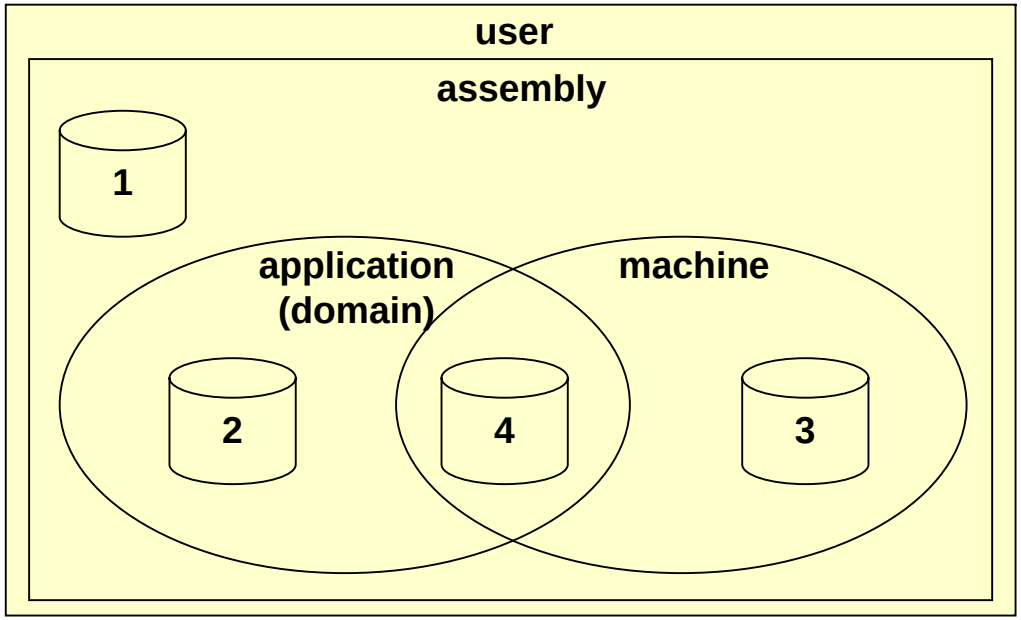


How isolation scoping works

- **Always isolated based on user and assembly**
  - Assembly ID based on strong name, publisher, URL, site, zone
- **May isolate also based on AppDomain**
  - This is the assembly name of the main application in which you are loaded
- **May also choose to have storage roam with user**
  - Roaming storage is a different store than non-roaming storage
- **Moral of the story**
  - If you want your data back, you need to make the same choices about AppDomain and roaming
- **Physical storage on hard drive**



The four supported isolated storage scopes



	Permission required	Which method to use on IsolatedStorageFile
1	AssemblyIsolationByRoamingUser	GetStore(User | Assembly | Roaming)
2	DomainIsolationByRoamingUser	GetStore(User | Assembly | Domain | Roaming)
3	AssemblyIsolationByUser	GetStore(User | Assembly) GetUserStoreForAssembly()
4	DomainIsolationByUser	GetStore(User | Assembly | Domain) GetUserStoreForDomain()

Using isolated storage

```
IsolatedStorageFile s = IsolatedStorageFile.GetUserStoreForDomain();
Console.WriteLine("Current Size: {0}", s.CurrentSize);
Console.WriteLine("Maximum Size: {0}", s.MaximumSize);
if (s.GetFileNames("foo").Length > 0) {
    using (FileStream media = new IsolatedStorageFileStream("foo",
           FileMode.Open, FileAccess.Read, FileShare.Read, s))
    using (StreamReader r = new StreamReader(media)) {
        Console.WriteLine(r.ReadToEnd());
    }
}
else {
    using (FileStream media = new IsolatedStorageFileStream("foo",
           FileMode.Create, FileAccess.Write, FileShare.None, s))
    using (StreamWriter w = new StreamWriter(media)) {
        w.Write("Testing 123");
    }
}
```

Extending security policy with custom gateways

- **What if you need mobile code to be able to access a set of COM components or DLLs?**
  - choice 1: modify policy to make that code fully trusted
  - choice 2: write your own gateway that demands a custom permission, then grant that permission to the mobile code
- **Steps to building a custom gateway**
  - Write and deploy a permission assembly
    - custom permission class
    - corresponding custom attribute for declarative use
  - Write the gateway
  - Adjust policy with your new permission



Implementing a permission

- **Derive a class from CodeAccessPermission or ResourcePermissionBase**
  - implement IUnrestrictedPermission
  - implement all required methods
  - try to avoid dependencies on assemblies other than mscorlib

```
bool IsUnrestricted();
IPermission Copy();
IPermission Intersect(IPermission target);
bool IsSubsetOf(IPermission target);
SecurityElement ToXml();
void FromXml(SecurityElement element);
```

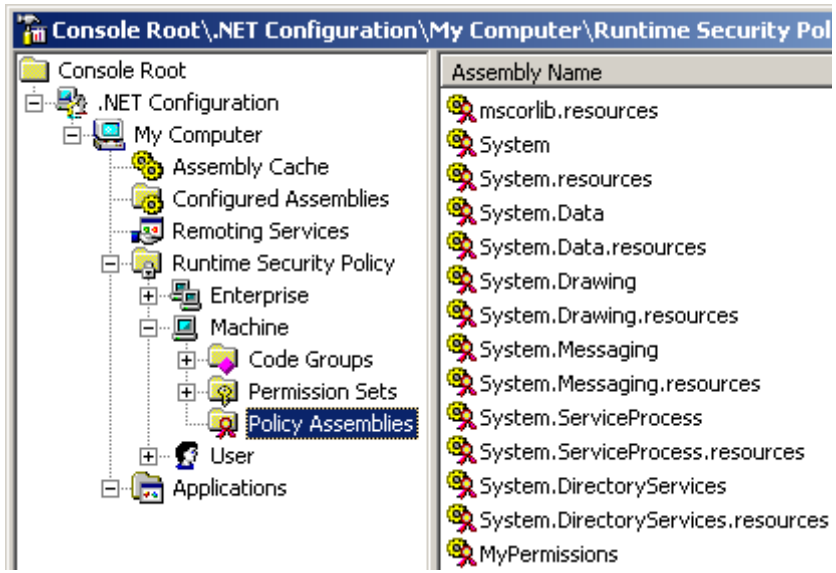
Implementing a permission attribute

- **For each permission, provide a corresponding attribute**
  - allows users declarative as well as imperative control

```
[AttributeUsage(AttributeTargets.Assembly |
                AttributeTargets.Class    |
                AttributeTargets.Struct   |
                AttributeTargets.Method   |
                AttributeTargets.Constructor)]
public sealed class BeepPermissionAttribute :
                                CodeAccessSecurityAttribute {
  public int MaxBeeps;
  public BeepPermissionAttribute(SecurityAction action)
    : base(action) {
    base.Unrestricted = false;
  }
  public override IPermission CreatePermission() {
    return new BeepPermission(base.Unrestricted, MaxBeeps);
  }
}
```

Working with permission assemblies

- **Preparing the permission assembly**
  - include your permission and permission attribute
  - strongly name it
  - mark it with AllowPartiallyTrustedCallersAttribute
  - install it into the GAC and keep the GAC fresh*
- **Add the assembly to the list of trusted policy assemblies**
  - use mscorcfg to do this



Writing a gateway class

- **Before accessing the guarded resource, demand an appropriate permission**
  - may need to use assert or SuppressUnmanagedCodeSecurity to "convert" a generic demand to a more specific one

```
public class Beeper {
    public static void Beep(int howManyBeeps) {
        new BeepPermission(howManyBeeps).Demand();
        for (int i = 0; i < howManyBeeps; ++i)
            Win32.MessageBeep(0);
    }
}
[SuppressUnmanagedCodeSecurity]
internal class Win32 {
    [DllImport("user32.dll")]
    internal static extern void MessageBeep(uint n);
}
```

Injecting custom permissions into policy

- **Write a little program to create or augment a permission set**
  - import with mscorcfg and refer code groups to it

```
static void Main() {
  NamedPermissionSet ps = new NamedPermissionSet(
                                  "MyPermissionSet",
                                  PermissionState.None);
  int maxBeeps = 5;
  ps.AddPermission(new BeepPermission(maxBeeps));
  Console.WriteLine(ps.ToXml());
}
```

```
<PermissionSet class="System.Security.NamedPermissionSet"
               version="1"
               Name="MyPermissionSet">
   <IPermission class="DM.Security.BeepPermission, ..."
                version="1"
                MaxBeeps="1"/>
</PermissionSet>
```

Summary

- Permission demands walk the stack to avoid luring attacks
- Writing mobile code means understanding CAS
- Isolated storage is a reasonable way to store per-user state from partially trusted code
- CAS can be applied to local code as well – this will get easier in the future
- You can extend CAS to guard custom resources or change the way policy works



Web Application Security

Authentication and Authorization in IIS6 and ASP.NET



Outline

- IIS 6 process model
- ASP.NET security contexts
- The HTTP pipeline
- Authentication
- Authorization
- Forms Authentication

IIS 6

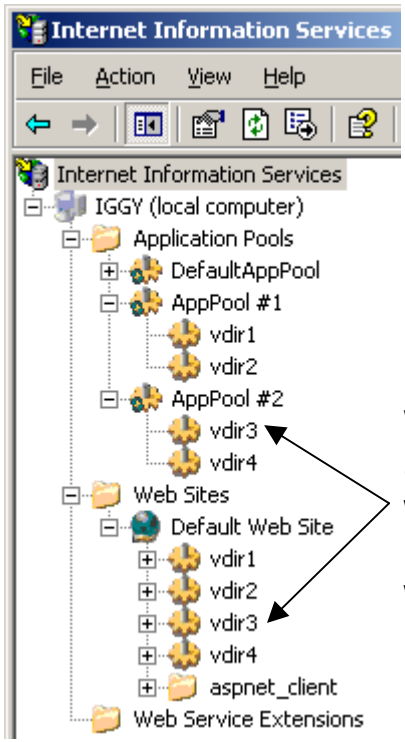
- **IIS is not installed by default**
- **When installed, “locked down” – only serves static content**
  - use IIS Manager: Web Service Extensions to enable dynamic content
- **Application Pools**
  - represents an instance of w3wp.exe
  - can configure identity of worker process via pool

IIS 6 process model

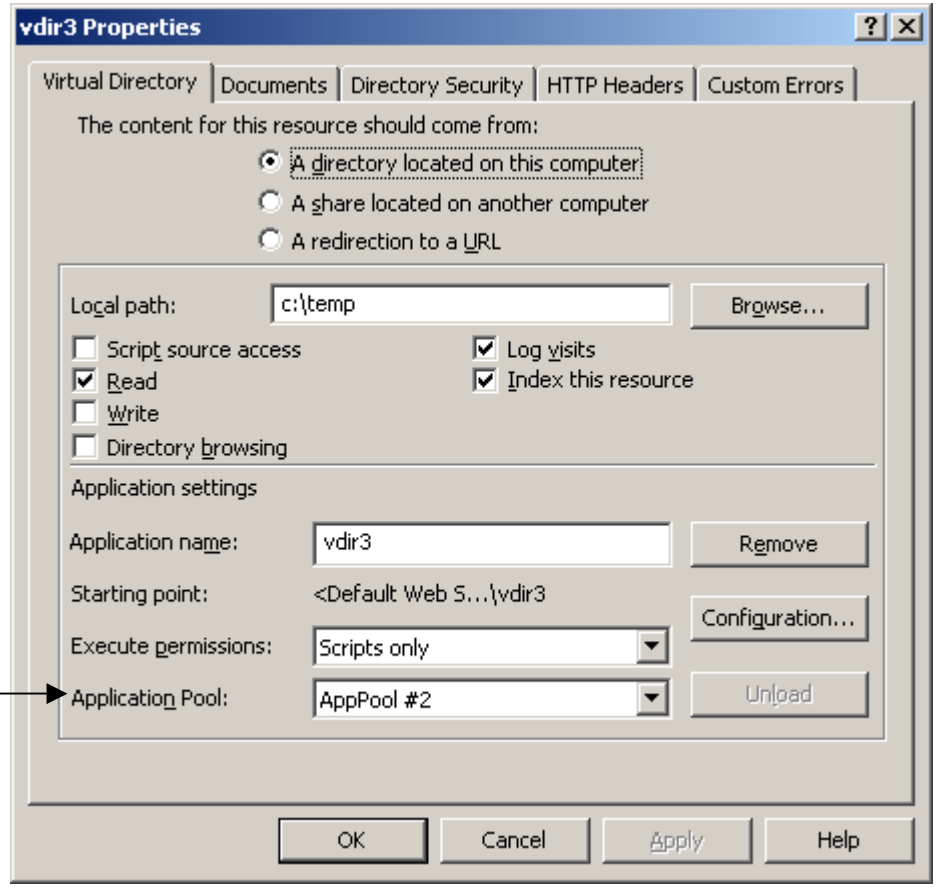
- **IIS 6 provides a process model that encourages isolation and running with least privilege**
  - requests routed directly from kernel to worker process
  - unlike IIS 5 where process isolation caused an extra context switch (from inetinfo.exe to the worker process)
- **Number of worker processes is configurable**
  - each “application pool” has its own worker process
  - one application pool by default, but you can create more
  - can configure identity and other settings via pool properties
- **Web application isolation is configurable**
  - each web application (virtual directory) assigned to a pool



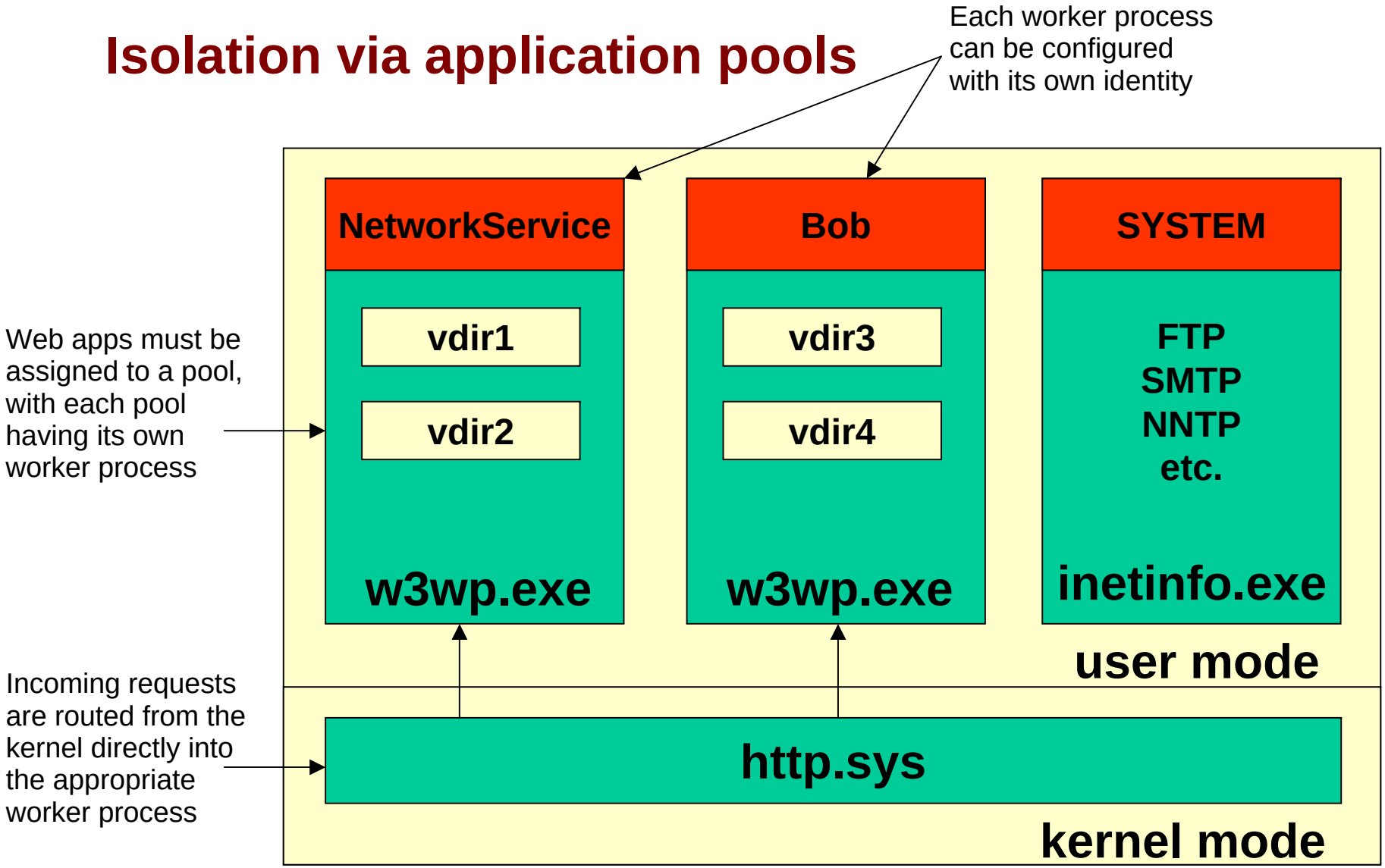
Isolation via application pools



Web apps must be assigned to a pool, with each pool having its own worker process



Isolation via application pools

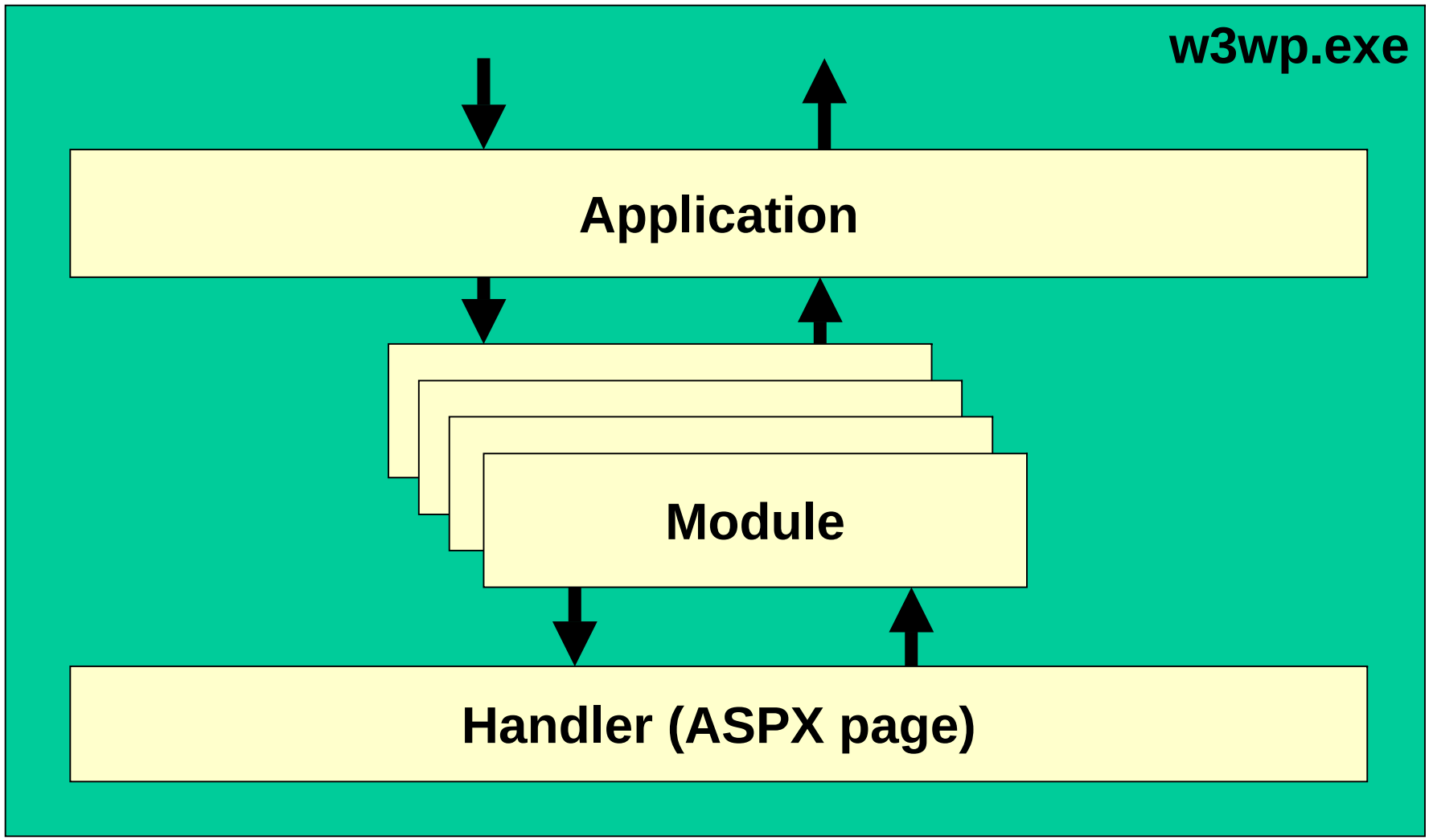


The HTTP pipeline in ASP.NET

- **ASP.NET handles requests via a pipeline**
- **Handler is endpoint for request**
  - an ASPX page is an example of a handler
- **Modules are pluggable extensions**
  - act like filters in the channel



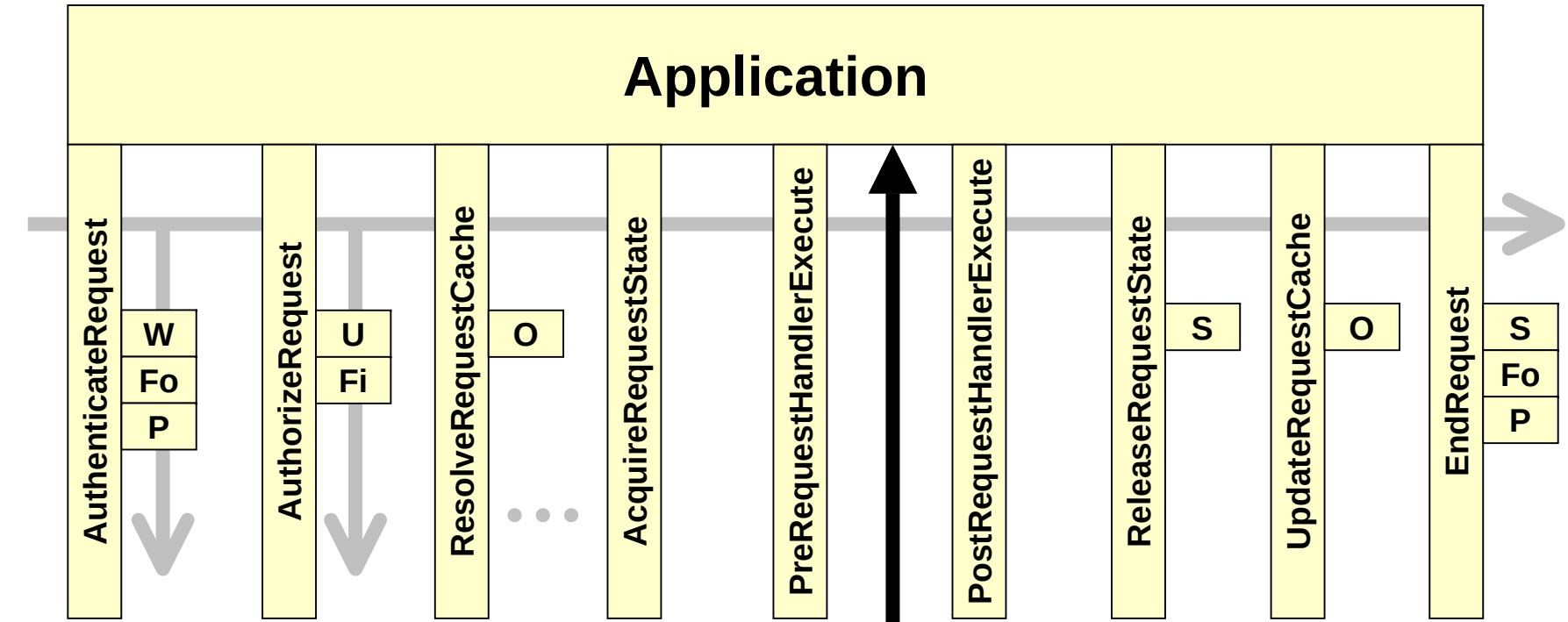
The HTTP pipeline from 25,000 feet



The importance of modules

- **Modules get to pre- and post-process each request**
- **Perfect place to put security code**
  - page developers can't "forget" to call your security logic
- **This is where you'll find all the built-in security**
  - windows authentication
  - forms authentication
  - passport authentication
  - file authorization
  - url authorization
- **Understanding how modules work is critical**
  - modules register for a fixed set of events
  - events are fired in order

The pipeline in more detail



Modules

- OutputCache
- Session
- WindowsAuthentication
- FormsAuthentication
- PassportAuthentication
- UrlAuthorization
- FileAuthorization

Security contexts

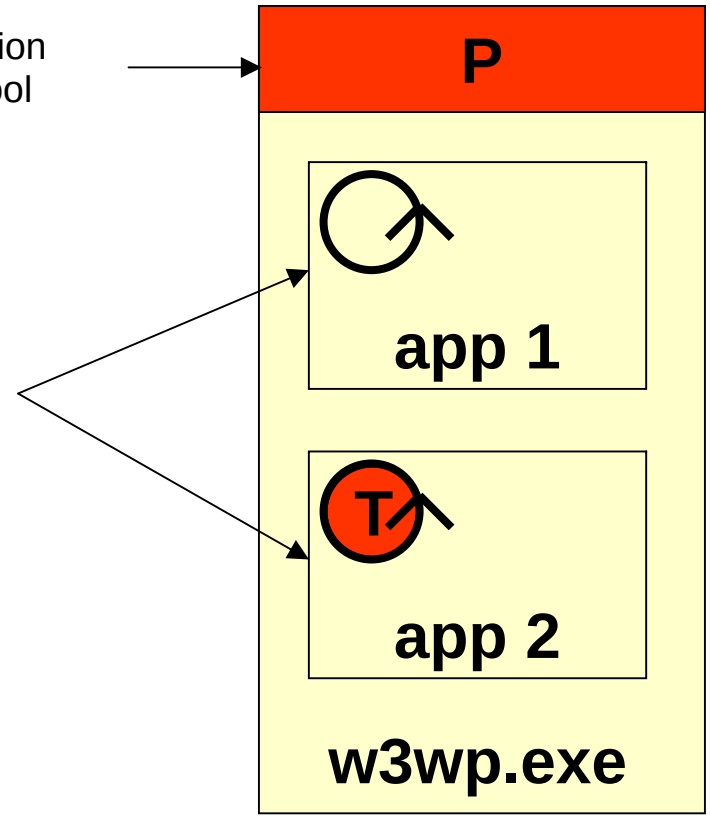
- **ASP.NET apps track both server and client identity**
- **Token tracks who the SERVER is**
  - process token
  - thread token (if you choose to turn on impersonation)
  - discover via WindowsIdentity.GetCurrent()
- **Managed identity tracks who the CLIENT is**
  - discover via HttpContext.User or Page.User



Process and thread token

process token configured once per application pool, affects all web apps running in that pool

impersonation configurable by each individual web application



Configuring the process token

- In IIS 6, use application pool configuration
- In IIS 5, use machine.config:

```
<!-- excerpt from machine.config -->
<processModel
    ...
    User='MACHINE'
    Password='Autogenerate'
    ...
/>
```

Configuring impersonation

- **Can turn on impersonation with a switch in web.config**
  - thread token will be obtained from IIS
  - emulates IIS 5 where every call impersonated the client (or the anonymous user account, typically IUSR_MACHINE)

```
<!-- web.config -->
<configuration>
  <system.web>
    <identity impersonate='false|true'/>
  </system.web>
</configuration>
```

Client authentication and HttpContext.User

- **ASP.NET provides several ways to authenticate clients**
  - native Windows authentication (default)
  - forms-based authentication
  - Passport authentication
- **Use <authentication> tag to control this**
- **Results stored in HttpContext.User**
  - also accessible via Page.User

```
<!-- web.config -->
<configuration>
  <system.web>
    <authentication mode='Windows|Forms|Passport|None'/>
  </system.web>
</configuration>
```

Authentication modes

- **None**
  - HttpContext.User refers to an anonymous identity
- **Windows**
  - HttpContext.User refers to a WindowsPrincipal
  - wraps the token IIS got for the principal
- **Forms**
  - HttpContext.User refers to a GenericPrincipal
  - name is whatever name you specify
  - roles empty by default (you can inject roles, more later)
- **Passport**
  - HttpContext.User refers to a PassportIdentity



Authentication in the pipeline

- **HttpContext.User is initialized during AuthenticateRequest**
  - usually via one of the built-in modules
    - WindowsAuthenticationModule
    - FormsAuthenticationModule
    - PassportAuthenticationModule
- **There is an undocumented event that fires immediately after**
  - its name is “DefaultAuthentication”
  - one hardwired module always registers for this event, its name is “DefaultAuthenticationModule”
  - this module has only one job
    - copy HttpContext.User → Thread.CurrentPrincipal
- **This allows you to use PrincipalPermissionAttribute**



Accessing the three ASP.NET security contexts

```
<%@import namespace='System.Security.Principal'%>
<%@import namespace='System.Runtime.InteropServices'%>

<%
IPrincipal myClient = this.User;

IPrincipal myThreadToken =
   new WindowsPrincipal(WindowsIdentity.GetCurrent());

RevertToSelf();  // remove any thread token

IPrincipal myProcessToken =
   new WindowsPrincipal(WindowsIdentity.GetCurrent());
%>

<script runat='server'>
[DllImport("advapi32.dll")]
static extern bool RevertToSelf();
</script>
```

Authorization

- **You can write code to do manual authorization**
  - declarative principal checks via PrincipalPermissionAttribute
  - imperative principal checks via HttpContext.User.IsInRole
- **You can also leverage two modules in the pipeline**
  - FileAuthorizationModule
  - UrlAuthorizationModule



File-based authorization

- **Always (and only) in effect with Windows authentication**
  - `<authentication mode='Windows'/>`
- **The managed principal in this case is a WindowsPrincipal**
  - holds a Windows access token
- **Token used in access check against file system DACL**
  - DACL = Discretionary Access Control List
- **This is exactly how classic ASP used to work**
- **IIS already enforces this for unmanaged resources**
  - static HTML files
  - images
  - etc.



URL-based authorization in web.config

- **Controls access to all managed resources**
  - this form of authorization works with all types of authentication
- **Does NOT control access to unmanaged resources by default**
  - only controls access to file extensions mapped to ASP.NET
  - you can change this mapping if you like (may reduce perf)

```
<configuration>
  <system.web>
    <authorization>
      <allow roles='Managers,Friends'/>
      <deny users='?'/>
    </authorization>
  </system.web>
</configuration>
```




URL-based authorization in subdirectories

- **You can drop an authorization section in a subdirectory**

```
<configuration>
  <system.web>
    <authorization>
      <allow roles='Subscribers'/>
    </authorization>
  </system.web>
</configuration>
```

- **You can also use a location tag**
  - control subdirectories
  - control individual files

```
<configuration>
  <location path='subscriptionOnlyDirectory'>
    <system.web>
      <authorization>
        <allow roles='Subscribers'/>
      </authorization>
    </system.web>
  </location>
</configuration>
```




Form-based authentication

- **Uses least-common-denominator technologies for portability**
  - simple HTML forms for gathering authentication information
  - cookies for maintaining a session
- **Designed to be flexible and extensible**
  - most of its internals are publicly documented classes
  - optional password management
- **Requires cookies**
  - no direct support for URL munging, for instance
  - the forms auth cookie is distinct from the session cookie



Cookies

- **To understand Forms auth you must understand cookies**
  - cookie mechanism documented in RFC 2965
  - web site sends state to user agent
  - user agent echoes state back to server with each request
  - user agent must not leak state across domains
  - cookies can be transient or persistent
  - nothing stops a client from modifying state in a cookie
  - nothing stops an eavesdropper from recording a cookie being sent over the Internet using raw HTTP

Basics

- **Basic Forms Authentication**
  - easy to get started with forms auth
  - turn it on in your root web.config file
  - design a login form that collects a user name and password/pin
  - upon submission, verify the password/pin and call FormsAuthentication.RedirectFromLoginUrl()
  - set authorization requirements to force a login



Enabling forms authentication

```
<!-- web.config in vroot -->
<configuration>
  <system.web>
    <authentication mode='Forms'>
      <forms name='ASPXAUTH'
             loginUrl='login.aspx'
             protection='All'
             requireSSL='false'
             timeout='30'
             slidingExpiration='false'
             path='/'
      />
    </authentication>
  </system.web>
</configuration>
```

shown here are the default values for each attribute

Example: a simple login form

```
<%@page language='c#' %>
<%@import namespace='System.Web.Security' %>

<form runat='server'>

<table><tr>
<td>Name:</td>
<td><asp:TextBox id='name' runat='server'/></td>
</tr><tr>
<td>Password:</td>
<td><asp:TextBox id='pwd' TextMode='password'
                          runat='server'/></td>
</tr></table>
<p><asp:CheckBox id='persist' runat='server'
  Text='Log me in automatically from this computer'/>

<p><asp:Label id='msg' runat='server'/>
<p><asp:Button Text='Login' runat='server'/>

</form>
```

Example: handling login

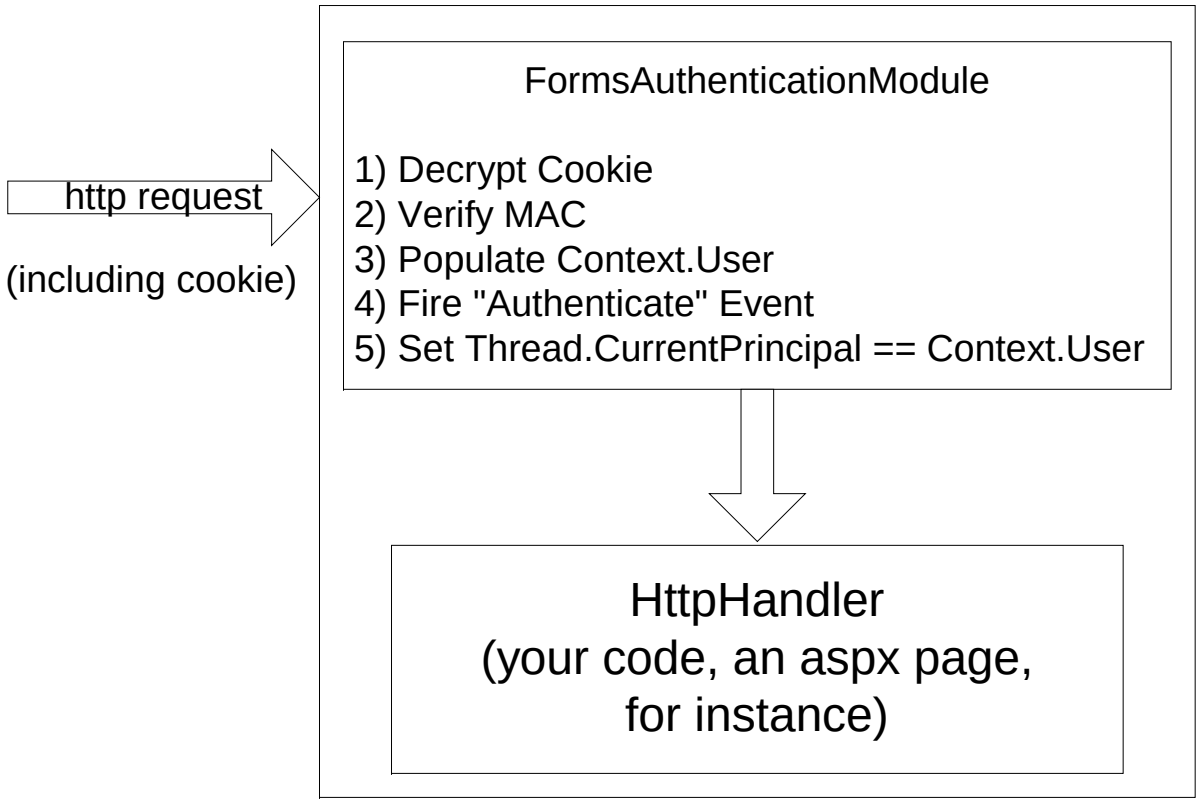
```
<script runat='server'>
void OnLogin(Object sender, EventArgs eventArgs) {
  if (authenticateUserSomehow(name.Text, pwd.Text)) {
    // only redirect if password is valid
    FormsAuthentication.RedirectFromLoginPage(
         name.Text, persist.Checked);
  }
  else {
    // otherwise leave them on this page
    msg.Text = "Unknown user name or password";
  }
}
</script>
```

Mechanics

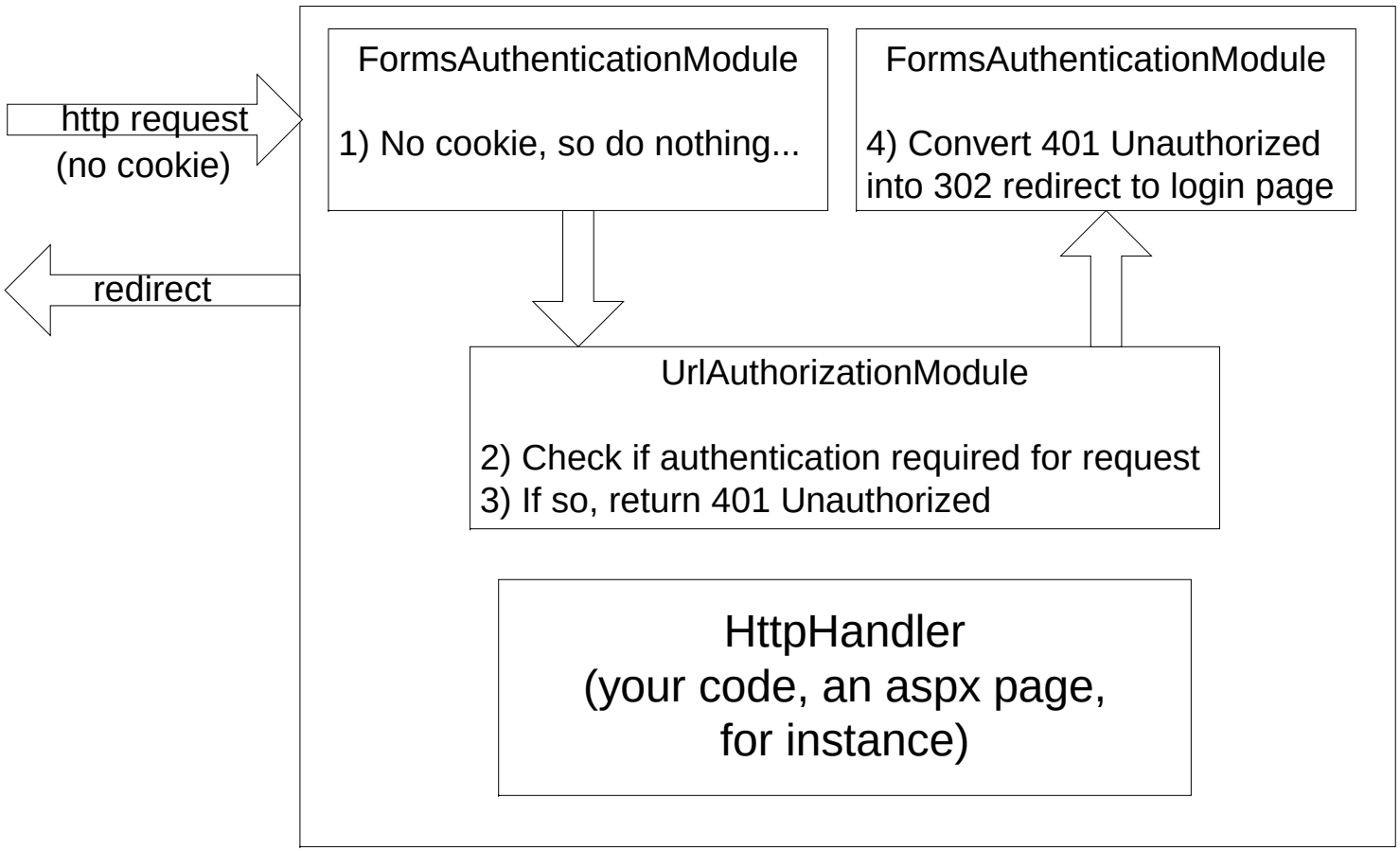
- **Forms Authentication Mechanics**
  - FormsAuthenticationModule preprocesses all requests
  - cookie turned into IPrincipal and associated with context
  - if no cookie and auth is required, redirects to login page
  - after login, redirected back to original page, sending cookie
  - cookie contains version, name, timestamp, optional user data
  - cookie protected with encryption and MAC (by default)



Transparent conversion of cookies to context



Transparent redirection to login page



Postprocessing

- **Postprocessing after Authentication**
  - often useful to do some additional work after FormsAuthenticationModule does its magic
  - add roles to principal...
  - convert user data in cookie to session state...
  - to do any of these things, need to hook the AuthenticateRequest event
  - can do this via global.asax or by writing a module
  - to add roles, just replace the principal!



Adding roles using global.asax

```
<%@application language='C#'%>
<%@import namespace='System.Security.Principal'%>

<script runat="server">
void Application_AuthenticateRequest(object sender, EventArgs args) {

  // get the principal produced by forms authentication
  IPrincipal originalPrincipal = HttpContext.Current.User;

  if (null != originalPrincipal) {
    IIdentity id = originalPrincipal.Identity;

    // TODO: lookup real roles based on id.Name
    string[] roles = { "ClubMember", "Swimmer" };

    // replace the principal with a new one with roles
    HttpContext.Current.User = new GenericPrincipal(id, roles);
  }
}
</script>
```

Forms authentication in a web farm

- **Remember cookies are MAC protected and encrypted**
  - ASP.NET creates new keys each time it's installed
  - this breaks a web farm
- **Can synchronize keys across a web farm**
  - set the <machineKey> element in machine.config

```
<system.web>
  <machineKey validationKey='[128 hexadecimal digits]'
              decryptionKey='[48 hexadecimal digits]'
              validation='SHA1'/>
</system.web>
```

Protecting cookies

- **The security of authentication cookies is weak unless they are protected**
  - cookies can be undetectably altered by users unless protected by a MAC
  - cookies can be stolen by an eavesdropper if sent over a non-secure channel
  - persistent cookies can be stolen off a user's hard drive
  - use SSL to encrypt all traffic to your secure site, or use the Secure option on your cookies (this is a SHOULD, not a MUST in RFC 2965 though) via requireSSL='true'
  - cookie paths don't work well with IIS due to case insensitivity (cookies are case sensitive, but IIS is not)



Summary

- **Your code runs within several security contexts**
  - process token
  - thread token
  - managed identity
- **Forms authentication provides an application level authentication mechanism**
  - as opposed to an operating system level mechanism
  - don’t need to give web clients NT accounts
- **If you use forms authentication, use SSL as well!**



Web Service Security

Making the web secure for ecommerce



Outline

- **Today's methods**
  - SSL
  - Client authentication
  - Intermediaries
- **The future: WS-Security**
  - Xml Canonicalization
  - Xml Signature
  - Xml Encryption



Web service security today

- **Most web services today are not secure**
- **Most web services published at xmethods.com, for example**
  - run over http, not https
  - don't bother with authentication at all, or
  - require user name and password be sent in the clear
  - email passwords to clients
- **Most secure web services are private**
  - not published in web service directories
  - used by a cooperating group of business partners
  - security almost always based on SSL

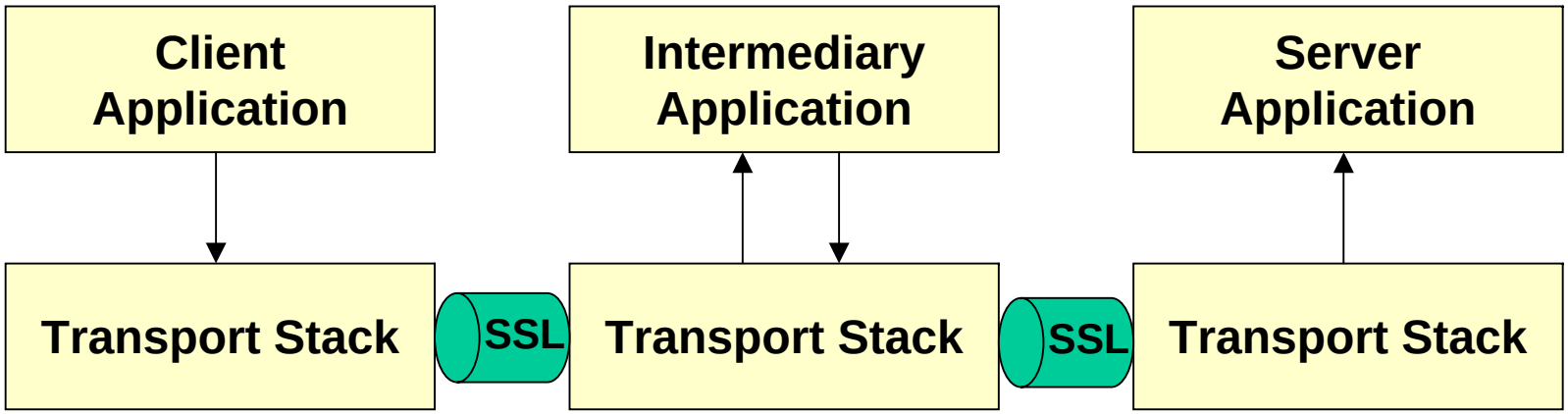
Using SSL to secure a web service

- **Benefits**
  - most HTTP stacks already support SSL
  - SSL provides authentication, integrity, and confidentiality
  - XML carried over secure transports doesn't have to conform to any particular security standard
- **Drawbacks**
  - SSL itself has some issues
    - establishing SSL sessions is expensive
    - SSL is very session oriented, making load balancing hard
  - intermediaries break end-to-end security
  - SSL is only widely supported over HTTP, while web services can run over other protocols such as SMTP



Transport level security and intermediaries

- **SSL is a transport level security mechanism**
  - who does the client see?
  - who does the server see?



Guidelines on using SSL

- **Decide how to authenticate clients**
  - certificates are powerful but hard to manage by clients
  - passwords are easier but often chosen or managed badly
- **Manage your own private key carefully**
  - buy your server certificate from a well-known authority
  - store backup copy of password protected PFX file in one vault
  - store password in another vault
- **Harden the web server as much as possible**
  - see http://www.microsoft.com/security for tools
- **Remember SSL only secures the pipe**
  - need to write secure code and consider insider attacks as well

Client authentication using certificates

- **Issue your own client certificates**
  - no need to involve a third party (like Verisign) unless client needs a single certificate for use with multiple vendors
  - certificate server is a good way to do this on the .NET platform
- **Require SSL with client certificates in IIS**
  - tell IIS which authorities you trust via a Certificate Trust List
  - decide whether or not to map certs to NT accounts
- **Educate your clients about their private key**
  - most people don't have a clue that it even exists
  - teach them how to back it up (export to password protected PFX file)
  - teach them what to do if it's compromised (alert you so you can revoke it!)



Client authentication using passwords

- **Manage passwords carefully**
  - do not store cleartext passwords on your server – store salted hashes instead
  - never send password material via unsecured email
  - don’t accept low entropy passwords
- **Make sure every web method requires SSL**
  - most implementations require username and password or session token to be sent with each request, so protect them
- **Be careful with session tokens**
  - if you issue them, make sure that guessing one is infeasible (use a long number generated from a good random source)

The future of web service security: GXA?

- **GXA = Global Xml (Web Services) Architecture**
- **WS-Security written by Microsoft, IBM, VeriSign**
  - transferred to Oasis for standardization
  - see http://www.oasis-open.org/committees/wss/
- **Hoists security up from transport**
  - allows for end-to-end security in the face of intermediaries
  - opens the door for alternatives to SSL



Implementations of WS-Security

- **Web Services Enhancements (WSE) from Microsoft**
  - http://msdn.microsoft.com/webservices/building/wse
- **Web Service Toolkit from IBM**
  - http://www.alphaworks.ibm.com/tech/webservicestoolkit



WS-Security

- **WS-Security spec focuses on three main mechanisms**
  - packaging security tokens in XML
  - message integrity
  - message confidentiality
- **Leverages existing W3C specs**
  - Details how these specs should be applied to SOAP



Foundations of WS-Security

- **To understand WS-Security, need to spend some time looking at the underlying specs**
  - XML Canonicalization
  - XML Signature
  - XML Encryption



The need for canonicalization

- **XML 1.0 serialization is pretty relaxed**

```
<person name='Bob' age='24'/>
<person  age="24"  name="Bob"    />
<person name="Bob" age='24'></person>
```

These documents are all equivalent as far as XML 1.0 is concerned

- **While the XML infoset represented by these documents are equivalent, the serialized forms are different**
  - When encoded into byte streams, they have different values
  - Thus their signatures will be different
- **Need to agree on a canonical serialized form before signatures will make any sense**

Canonical XML

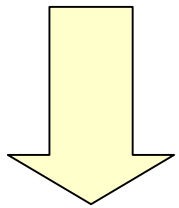
- **Canonical XML 1.0**
  - http://www.w3.org/TR/xml-c14n (also RFC 3076)
  - W3C Recommendation 15 March 2001
- **Describes the transformation of an XML document into a canonical byte stream**
  - UTF-8 encoding
  - CDATA sections replaced with their character content
  - <foo/> converted to <foo></foo>
  - Attribute values delimited by double quotes
  - Attributes and namespaces are arranged lexicographically
  - Whitespace outside of the document element and within start and end tags is normalized
  - and so on…

Example of canonicalization

```
<person name='Bob' age='24'/>
```

```
<person  age="24"  name="Bob"    />
```

```
<person name="Bob" age='24'></person>
```

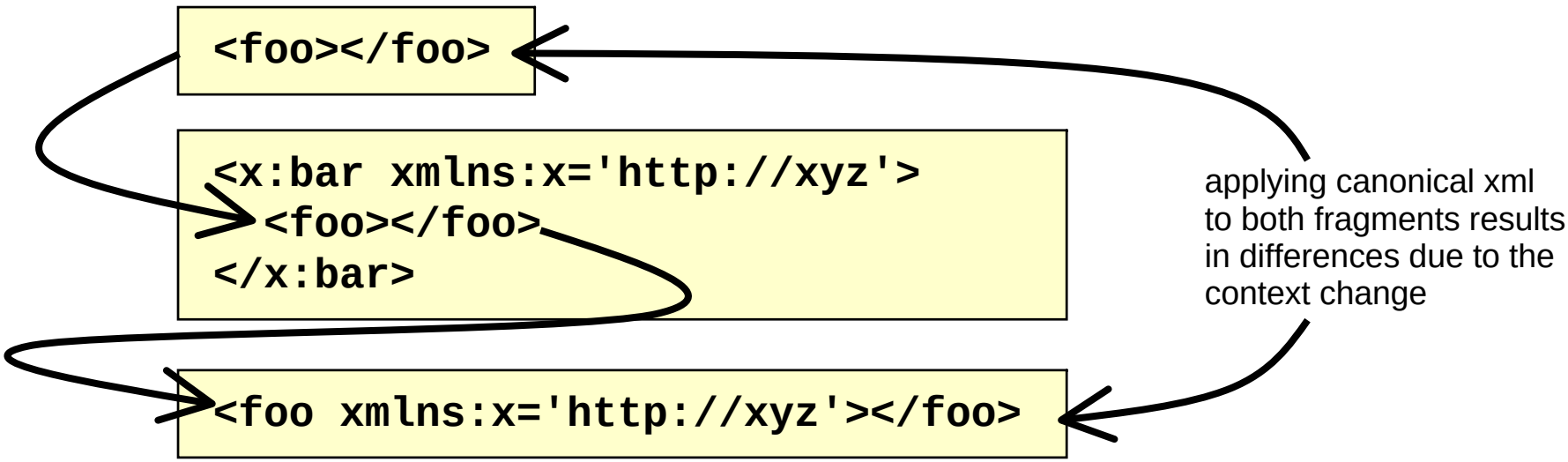


```
<person age="24" name="Bob"></person>
```

Canonical XML

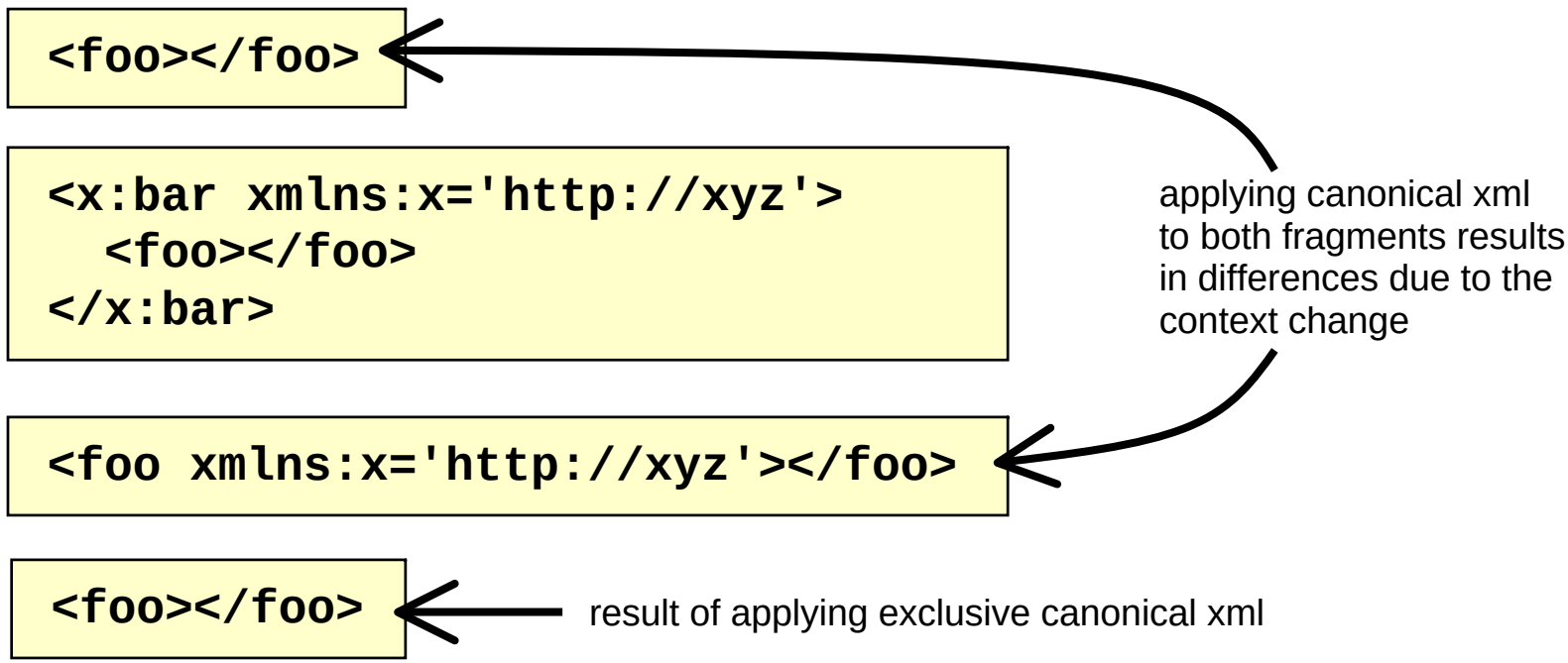
Canonicalization and SOAP

- **SOAP is all about enveloping XML fragments for transport**
  - Canonical XML doesn't address the context changes that occur during enveloping
  - Can cause problems with resulting signatures



Exclusive Canonical XML

- **Exclusive XML Canonicalization 1.0**
  - http://www.w3.org/TR/xml-exc-c14n
  - W3C Recommendation 18 July 2002
- **Provides better support for fragments**



XML Signature

- **XML-Signature Syntax and Processing**
  - http://www.w3.org/TR/xmldsig-core (also RFC 3275)
  - W3C Recommendation 12 February 2002
- **What XML Signature does**
  - Signs fragments of XML documents
  - Records the details and the signature value itself in a <Signature/> element
  - Provides hints about which key was used for the signature

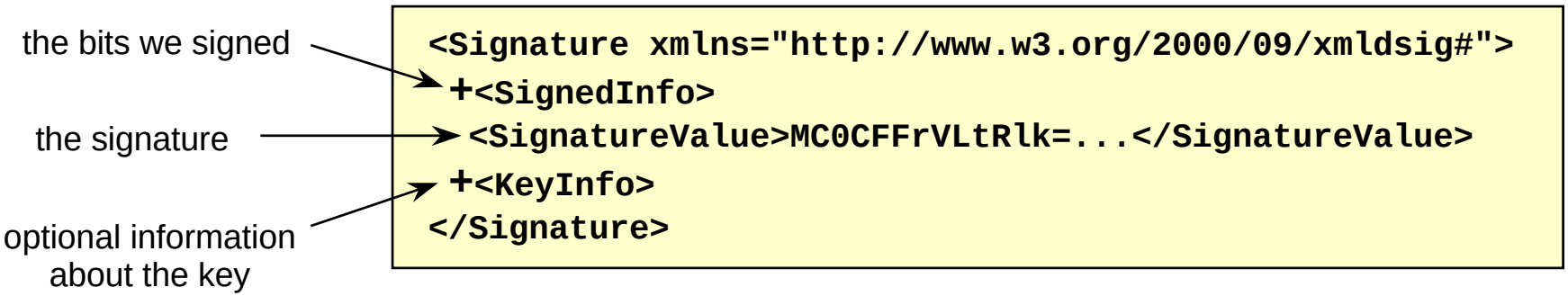


An XML signature in the wild

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod
      Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
    <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
    <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
      <Transforms>
        <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>
  <KeyInfo>
    <KeyValue>
      <DSAKeyValue>...</DSAKeyValue>
    </KeyValue>
  </KeyInfo>
</Signature>
```

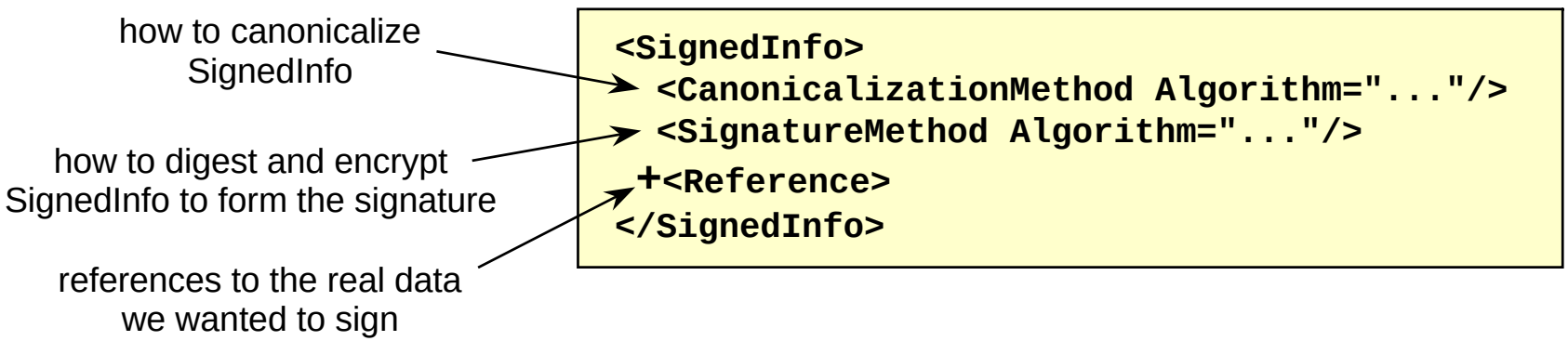
Inside an XML signature

- **A typical signature contains three parts**
  - the data being signed
  - the actual bits of the resulting signature
  - information about the key used for the signature



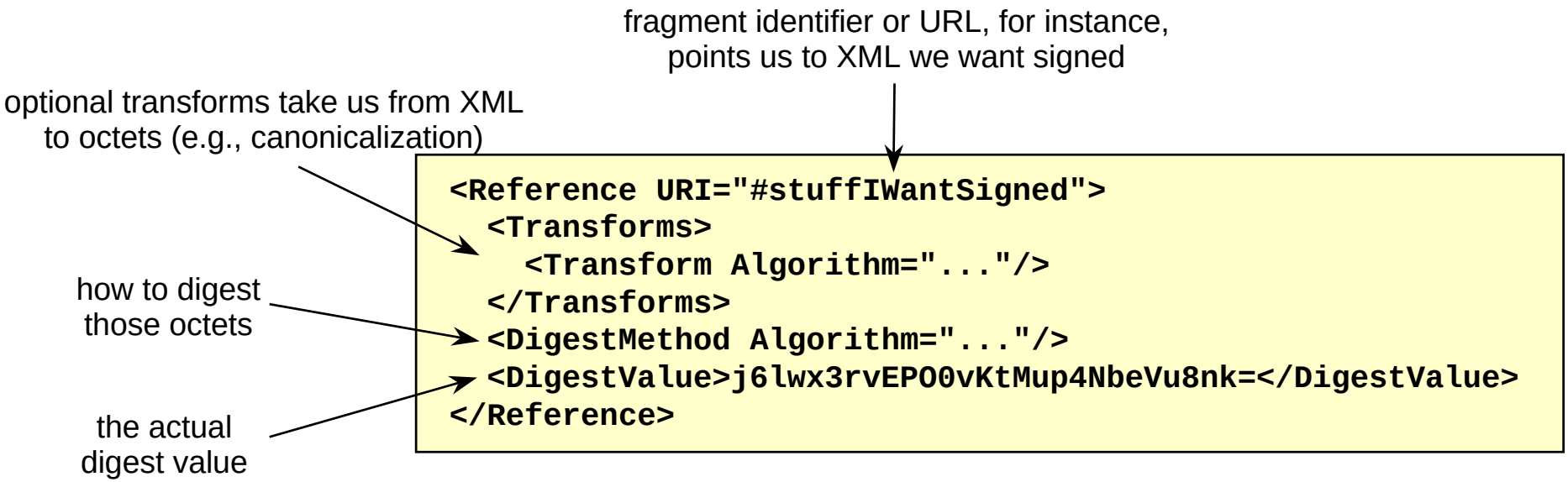
What really gets signed?

- **SignedInfo contains all the bits that are physically signed**
  - contains detailed instructions on how to
    - canonicalize SignedInfo
    - digest (hash) SignedInfo
    - encrypt the resulting digest to form the final signature
  - indirectly references XML node sets
    - this is the information that is logically being signed



Referencing real XML we want to sign

- **References link the signature to the XML we wanted to sign**
  - pulls in a digest (hash) of each reference to be signed
  - signing the digest value binds the content to the signer's key



WS-Security and signatures

- **XML Signature binds XML to a key**
- **WS-Security takes this one step further**
  - binds the signature to a claim
  - <KeyInfo/> contain a WS-Security element called SecurityTokenReference
  - security tokens (may contain key material) can be packaged inside the document or be pulled from external sources
- **Example of usage**
  - bind a web service request to a capability issued by a licensing service as opposed to a user's identity
  - allows for interesting authorization and delegation scenarios

Security tokens

- **Security token is a collection of one or more claims**
  - claims can be about identity, group membership, key ownership, capability, etc.
- **Example: Kerberos ticket issued by Active Directory**
  - contains one identity claim
  - contains zero or more group membership claims
  - claim is signed by some authority (domain controller)
- **Example: X.509 certificate issued by VeriSign**
  - claim binds a name to a public key
  - claim is signed by an authority (VeriSign)
- **Example: User name**
  - contains one identity claim
  - signed (endorsed) by no one



Packaging security tokens into XML

- **WS-Security provides a means to do this**
  - Binary security tokens
  - User name security tokens



Binary security tokens

- **Binary security tokens have two important attributes**
- **@ValueType**
  - Kerberos ticket granting ticket - wsse:Kerberosv5TGT
  - Kerberos server ticket - wsse:Kerberosv5ST
  - X.509 certificate - wsse:X509v3
- **@EncodingType**
  - Base 64 - wsse:Base64Binary
  - Hex - wsse:HexBinary
- **Both values and encodings are extensible**
  - Just use a namespace other than that for WS-Security
- **Content is the encoded value of the token**

A binary security token

```
<wsse:BinarySecurityToken wsu:Id="myKerbTicket"
                          ValueType="wsse:Kerberosv5ST"
                          EncodingType="wsse:Base64Binary">
MIIEZzCCA9CgAwIBAgIQEmtJZc0...
</wsse:BinarySecurityToken>
```



wsse is the conventional prefix for the WS-Security namespace:
http://schemas.xmlsoap.org/ws/2002/04/secext

Binding a signature to a claim

```
<ds:Signature>
  <!-- signature body omitted for brevity -->
  <ds:KeyInfo>
    <wsse:SecurityTokenReference>
      <wsse:Reference URI='#myKerbTicket'/>
    </wsse:SecurityTokenReference>
  </ds:KeyInfo>
<ds:Signature>
```

this signature is now linked to a binary security token defined elsewhere in the document

ds is the conventional prefix for the XML Signature namespace: http://www.w3.org/2000/09/xmldsig#"

The user name token

- **Allows for passing a user name**
  - and optionally password information
- **Password can be sent in the clear, or digested**
  - in both cases, need a secure channel or it's pointless
- **Password can also be digested with a nonce and timestamp**
  - if nonce and/or timestamp present, digest calculated as follows
    - SHA1(nonce + timestamp + password)

```
<wsse:UsernameToken>
  <wsse:Username>NNK</wsse:Username>
  <wsse:Password Type="wsse:PasswordDigest">FEdR...</wsse:Password>
  <wsse:Nonce>FKJh...</wsse:Nonce>
  <wsu:Created>2001-10-13T09:00:00Z </wsu:Created>
</wsse:UsernameToken>
```

A problem with user name tokens

- **To validate the digested password, must have access to cleartext password on server**
  - passwords are normally used when humans are involved
  - humans use the same password everywhere
  - forcing servers to have access to cleartext passwords is bad
  - leads to domino effect if one server is compromised
- **A better way**
  - client_master_key = SHA1(realm + password)
  - where realm is a string provided by the web service
  - use client_master_key as the “password” described for UserNameToken and you’ll be much better off

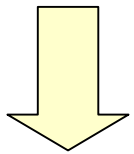


XML Encryption

- **XML-Encryption Syntax and Processing**
  - http://www.w3.org/TR/xmlenc-core
  - W3C Candidate Recommendation 02 August 2002
- **What XML Encryption does**
  - pulls out selected fragments of an XML document and replaces with an <EncryptedData> element
  - supports key encryption via the <EncryptedKey> element
  - supports <KeyInfo> from XML Signature as well to bind encrypted data to a key

Example: encrypting XML

```
<user>
  <name>Alice</name>
  <creditCards>
    <card type='Visa'>1234 4321 1234 4321</card>
    <card type='Discover'>4321 1234 4321 1234</card>
  </creditCards>
</user>
```



```
<user>
  <name>Alice</name>
  <xenc:EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'>
    <xenc:CipherData>
      <xenc:CipherValue>A9CgAwIBAgI...</xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedData>
</user>
```

Key encryption

- **Public keys are never used to encrypt bulk data**
- **Need a way to communicate encrypted session keys**
- **XML Encryption provides this via <EncryptedKey>**
  - also provides a way to reference <EncryptedData> elements that are encrypted with the key via <ReferenceList>
  - WS-Security recommends using this feature



Example: key encryption

```
<root> <!-- namespace declarations omitted for brevity -->
  <xenc:EncryptedKey>
    <xenc:EncryptionMethod Algorithm='...'/>
    <ds:KeyInfo>
      <ds:KeyName>Alice</ds:KeyName>
    </ds:KeyInfo>
    <xenc:CipherData>
      <xenc:CipherValue>A9CgAwIBAgI...</xenc:CipherValue>
    </xenc:CipherData>
    <xenc:ReferenceList>
      <xenc:DataReference URI='#encryptedFragment'/>
      <xenc:DataReference URI='cid:image'/>  <!-- attachment -->
    </xenc:ReferenceList>
  </xenc:EncryptedKey>

  <xenc:EncryptedData Id='encryptedFragment'>
    <xenc:EncryptionMethod Algorithm='...'/>
    <xenc:CipherData>
      <xenc:CipherValue>6caySYw68209shI...</xenc:CipherValue>
    <xenc:CipherData>
  </xenc:EncryptedData>
</root>
```

Framing

- **WS-Security defines a SOAP header for security**
  - security tokens
  - signature elements
  - encrypted keys
- **Defines order of operations within header**
  - should be able to “unwind” a message by traversing Security header in document order
  - intermediaries should maintain this order by adding any new operations to the top of the header
- **Defines recipient of header**
  - only one Security header may omit actor attribute
  - others are destined for named intermediaries (SOAP actors)



Example: SOAP with WS-Security header

```
<soap:Envelope
      xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
      xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
      xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
  <soap:Header>
     <wsse:Security>
        <wsse:BinarySecurityToken ...
        <xenc:EncryptedKey ...
        <ds:Signature ...
     </wsse:Security>
  </soap:Header>
  <soap:Body>
     <xenc:EncryptedData ...
  </soap:Body>
</soap:Envelope>
```

Note order of operations allows you to correctly unwind message

Summary

- **Web services security is still in its infancy**
- **For now, stick with tried and true methods like SSL**
- **WS-Security applies existing specs to web services**
  - XML Canonicalization
  - XML Signature
  - XML Encryption
- **Expect churn during standardization**
- **Don’t expect a standard solution anytime soon**

Remoting

Securing System.Runtime.Remoting



Why bother securing .NET remoting?

- **.NET remoting will mainly be used on a corporate intranet**
  - the corporate intranet is protected by firewalls
  - so aren't we safe from external attacks?
- **Firewalls can slow down external attacks, but they aren't a silver bullet**
  - the best defenses are layered ones
  - insider attacks can be much more devastating
  - ignore security on the intranet at your own peril



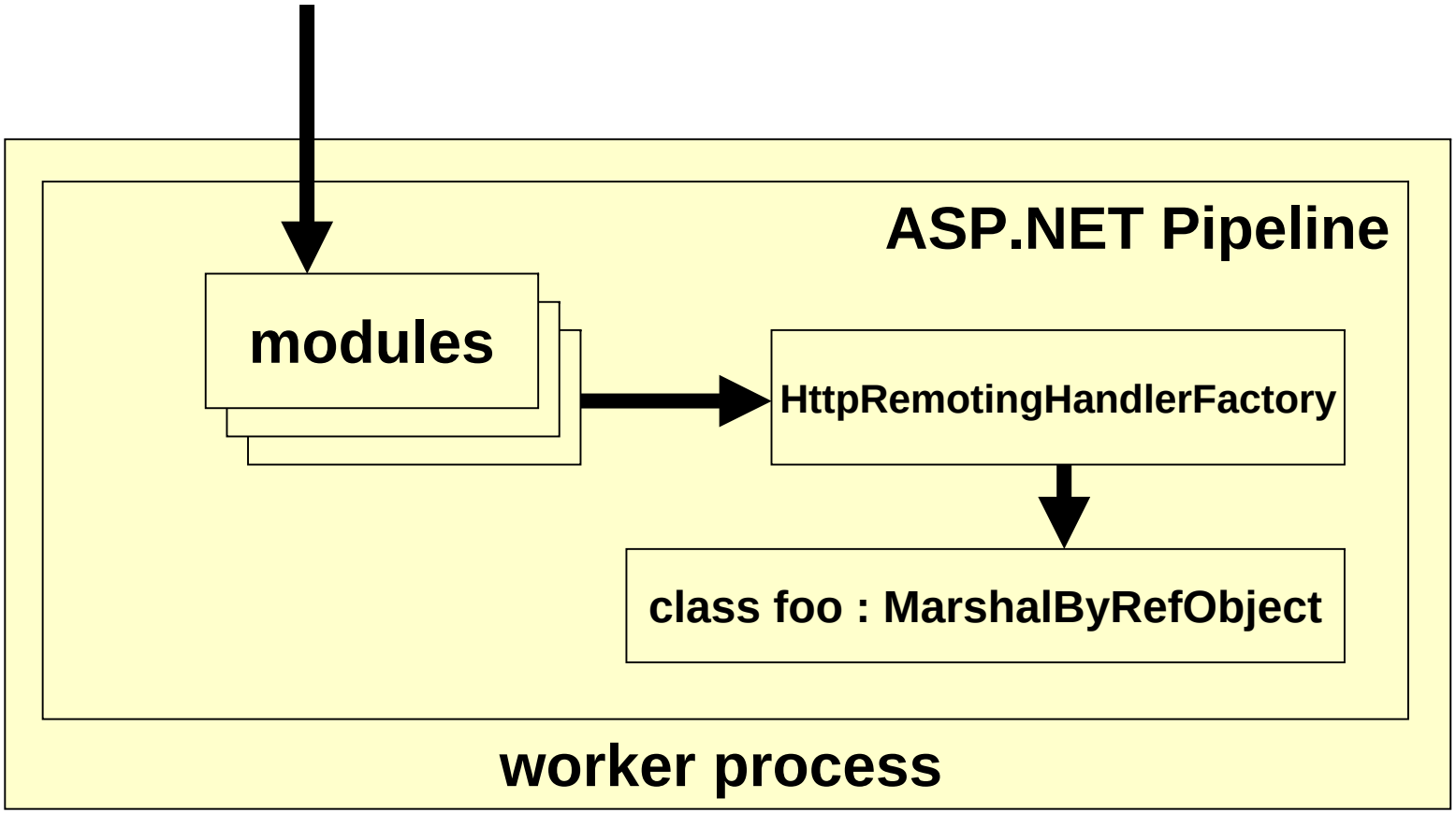
Solution one: hosting in IIS

- **ASP.NET comes with a special handler for remoting**
  - HttpRemotingHandlerFactory
- **Reads server remoting configuration from web.config**
- **Passes HTTP requests through to remoting channel**
  - requires that you use Http channel
  - you can use any formatter you like (binary, soap, custom)



Hosting in IIS

http://acme.com/myApp/foo.soap



How to host in IIS

- **Set up a virtual directory in IIS**
- **Build server object into a library assembly (.DLL)**
  - drop this assembly into “bin” directory under vroot, or install in GAC
- **Create a web.config file and wire it all up**
  - must use http channel when hosting in IIS
  - prefer binary formatter for efficiency
  - server URI must end with “.soap” or “.rem” to map to .NET remoting handler in ASP.NET pipeline



web.config example

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref='http'>
     <serverProviders>
      <formatter ref='binary'/>
     </serverProviders>
    </channel>
   </channels>
   <service>
    <wellknown mode='Singleton' type='Calc, server'
               objectUri='calc.soap'/>
   </service>
  </application>
 </system.runtime.remoting>
</configuration>
```

Client configuration

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref='http' useDefaultCredentials='true'>
     <serverProviders>
      <formatter ref='binary'/>
     </serverProviders>
    </channel>
   </channels>
   <service>
    <wellknown mode='Singleton' type='Calc, server'
               objectUri='calc.soap'/>
   </service>
  </application>
 </system.runtime.remoting>
</configuration>
```

Security goals

- **Authentication**
- **Message Integrity**
- **Message Confidentiality**
- **Authorization**

Security when hosting in IIS

- **IIS has many options for authentication**
  - basic
  - digest
  - integrated (Kerberos)
  - SSL (optionally with client certs)
- **SSL is required if you want**
  - message integrity
  - message confidentiality
  - mutual authentication



Typical configurations: SSL + Basic Authentication

- **Benefits:**
  - works well with firewalls & proxies
  - passwords easy to work with for clients
  - provides mutual authentication, integrity, confidentiality
- **Drawbacks:**
  - doesn't leverage single sign on (client must provide cleartext password)
  - server code has access to client's cleartext password



Typical configurations: Integrated Authentication (Kerberos)

- **Benefits:**
  - leverages single sign on
  - great for Intranet clients
- **Drawbacks:**
  - generally does not work with firewalls or proxies
  - must add SSL into the mix to achieve
    - mutual authentication
    - message integrity
    - message confidentiality



Typical configurations: SSL with client certificates

- **Benefits:**
  - works well with firewalls & proxies
  - provides mutual authentication, integrity, confidentiality
- **Drawbacks:**
  - human clients have trouble managing personal certificates
  - certificates aren't as mobile as passwords
    - smart cards can help here



Authorization

- **When hosting in IIS, your server code runs under ASP.NET**
- **To get access to the client principal, be sure to set authentication mode to “Windows”**
  - this is the default setting in machine.config anyway
- **Use normal ASP.NET techniques for authorization**
  - Thread.CurrentPrincipal.IsInRole()
  - PrincipalPermission
  - PrincipalPermissionAttribute
- **Uncaught exceptions will propagate back to client**
  - this is bad (includes a server-side stack trace)
  - trap, log, and rethrow exception at top level of call to avoid giving away too much information
  - consider using a remoting sink to automate this



IIS hosting and callbacks

- **Callbacks will be completely unsecured**
  - No authentication
  - No message integrity protection
  - No message confidentiality protection
- **We really need a built-in solution**



Solution two: the SSPI remoting sink

- **In August, 2002, Microsoft published two unsupported samples for the .NET Framework**
  - SSPI wrapper[1]
  - Remoting security channel sinks[2]
- **To use, must install both on client and server**
- **Uses SSPI to add security (e.g., Kerberos) to the channel**
  - authentication
  - message integrity
  - message confidentiality



The SSPI remoting sink

- Does not require IIS
- Does not require SSL
- Doesn’t fall down in the face of callbacks
- Will eventually be built in (but not in Everett)



Using the SSPI remoting sink

- **Wire the sink into the channel via configuration file**
  - must do this on both client and server
- **Use remoting like you normally would**
  - current sample autoimpersonates client
  - this is broken, watch for fix in future



client.exe.config

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref='http'>
     <clientProviders>
      <formatter ref='binary'/>
      <provider ref='sspi'
                securityPackage='kerberos'
                impersonationLevel='impersonate'
                authenticationLevel='packetPrivacy'/>
     </clientProviders>
    </channel>
   </channels>
  </application>
 </system.runtime.remoting>
</configuration>
```

server.exe.config

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref='http' port='4243'>
     <serverProviders>
      <provider ref='sspi'
                securityPackage='kerberos'
               impersonationLevel='impersonate'
                authenticationLevel='packetPrivacy'/>
      <formatter ref='binary'/>
     </serverProviders>
    </channel>
   </channels>
   <service>
    <wellknown type='Bob, server'
               mode='Singleton' objectUri='bob'/>
   </service>
  </application>
 </system.runtime.remoting>
</configuration>
```




machine.config

- **The preceding example config files made use of ref='sspi'**
  - adjust machine.config as follows to make this work

```
<system.runtime.remoting>
  <!-- other stuff omitted for brevity -->
  <channelSinkProviders>
    <clientProviders>
      <formatter id="soap" ... />
      <formatter id="binary" ... />
      <provider id="sspi" type="see notes"/>
    </clientProviders>
    <serverProviders>
      <formatter id="soap" ... />
      <formatter id="binary" ... />
      <provider id="sspi" type="see notes"/>
    </serverProviders>
  </channelSinkProviders>
</system.runtime.remoting>
```

Summary

- .NET remoting has no built in, supported, security model
- Can layer security on by hosting in IIS
- Can layer security on using an unsupported sample from MS
- Expect improvement in the future (not Everett, though ☹)

COM+

Securing System.EnterpriseServices



Outline

- Why use COM+?
- Security settings
- Role-based access checks
- Deployment model

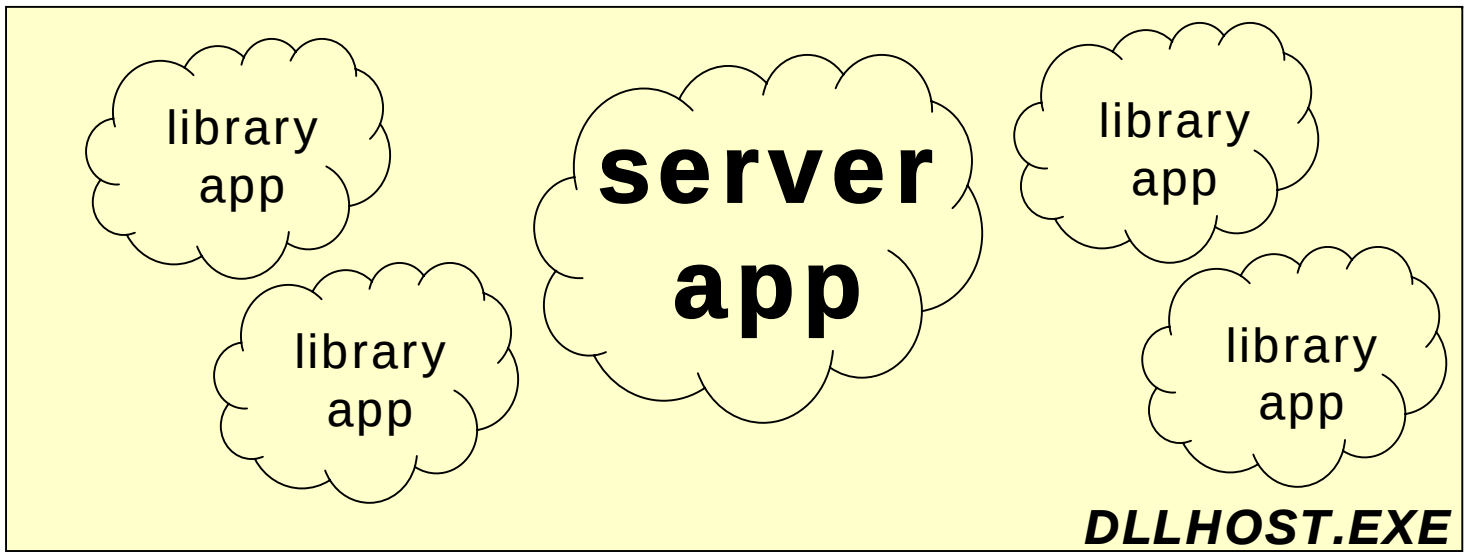


Why use COM+?

- **COM+ (a.k.a. Enterprise Services) provides many features that corporate developers need**
  - Secure hosting and networking via DCOM
    - Kerberos authentication
    - Message integrity
    - Message confidentiality
  - Role-based security tied to Windows logon
  - Distributed transactions
  - Deployment model
- **May also need to simply support existing COM-based code**

Server applications vs. library applications

- COM+ allows both inproc and out-of-proc activation
- Library applications have very little control over process-wide security settings
- Process-wide security settings determined by the single server application in the process



Process-wide security settings

- Authentication level
- Impersonation level
- Role-based access checks



Authentication level

- **Controls how much protection is provided for the channel**
  - Client and server both specify this, and COM uses the most secure of the two settings

Level	Authenticate connection	MAC Protect		Encrypt
		Headers	Payload	Payload
None				
Connect	X			
Packet	X	X		
Packet Integrity	X	X	X	
Packet Privacy	X	X	X	X

use Packet Privacy unless you've got a really good reason not to

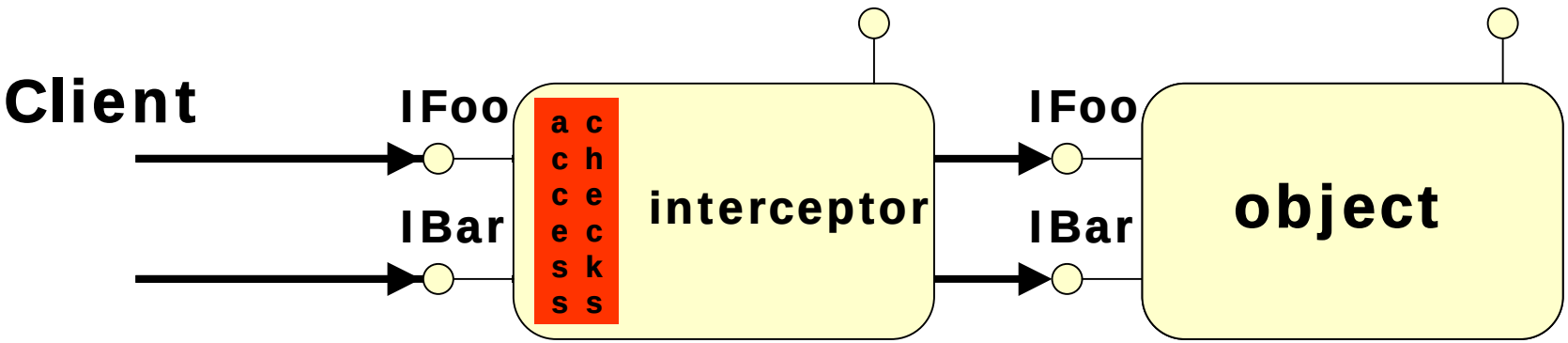
Impersonation level

- **Servers requiring authentication can impersonate clients**
- **Client can restrict what servers can do while impersonating**
  - Choose whether to send network credentials to server
  - Choose whether server can open local objects with credentials
- **Note that impersonation level is a client-side setting**

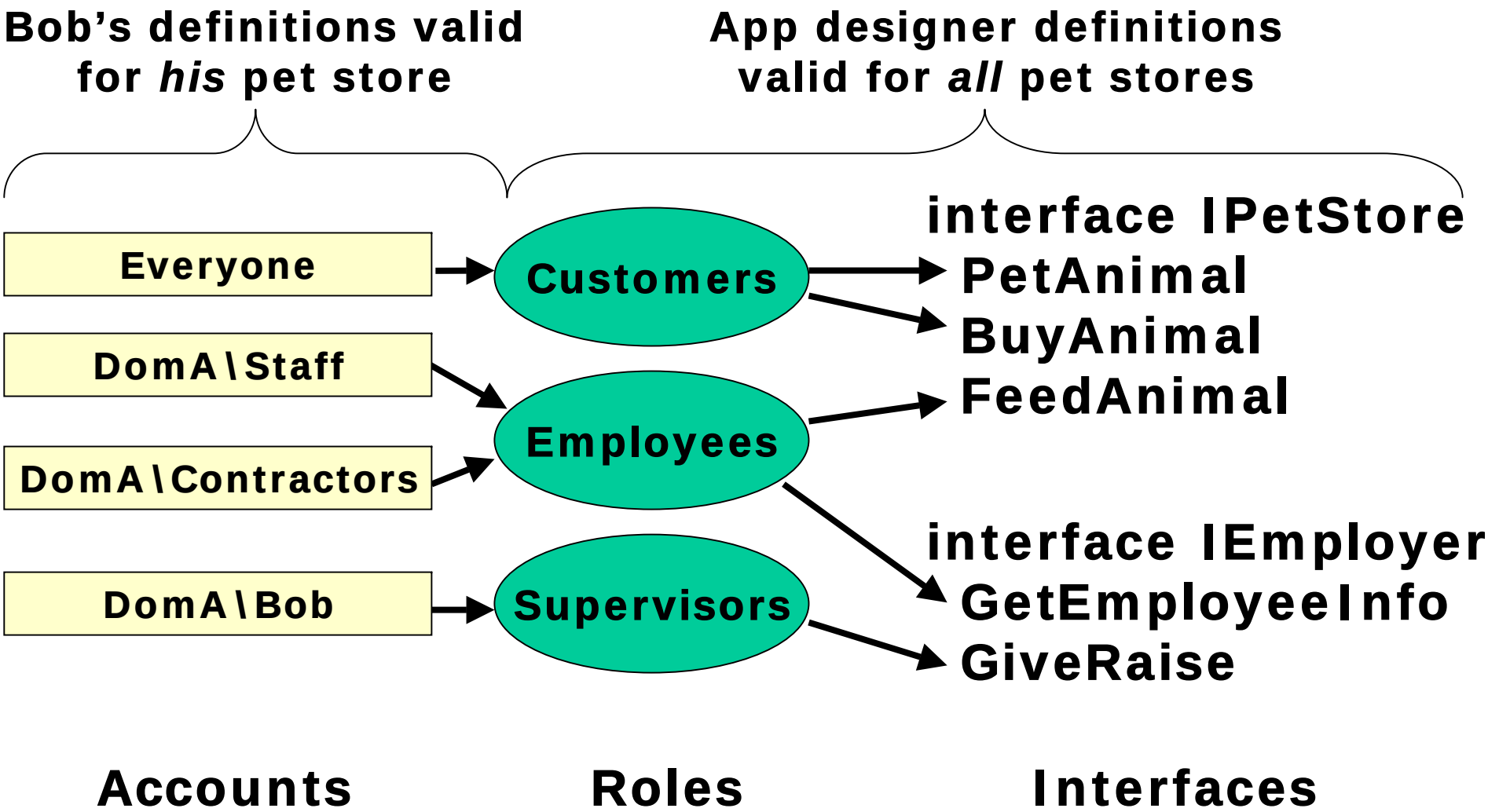
Level	Server can look at token	Server can use token to open kernel objects	Server can use client's network credentials
Anonymous			
Identify	**X**		
Impersonate	**X**	**X**	**local***
Delegate	**X**	**X**	**remote***

Role-based access checks

- COM+ has a very nice infrastructure for role-based access control
- It's all based on the notion of interception
- By moving the access checks out of the object and into the interception layer, they can be configured declaratively



Roles illustrated

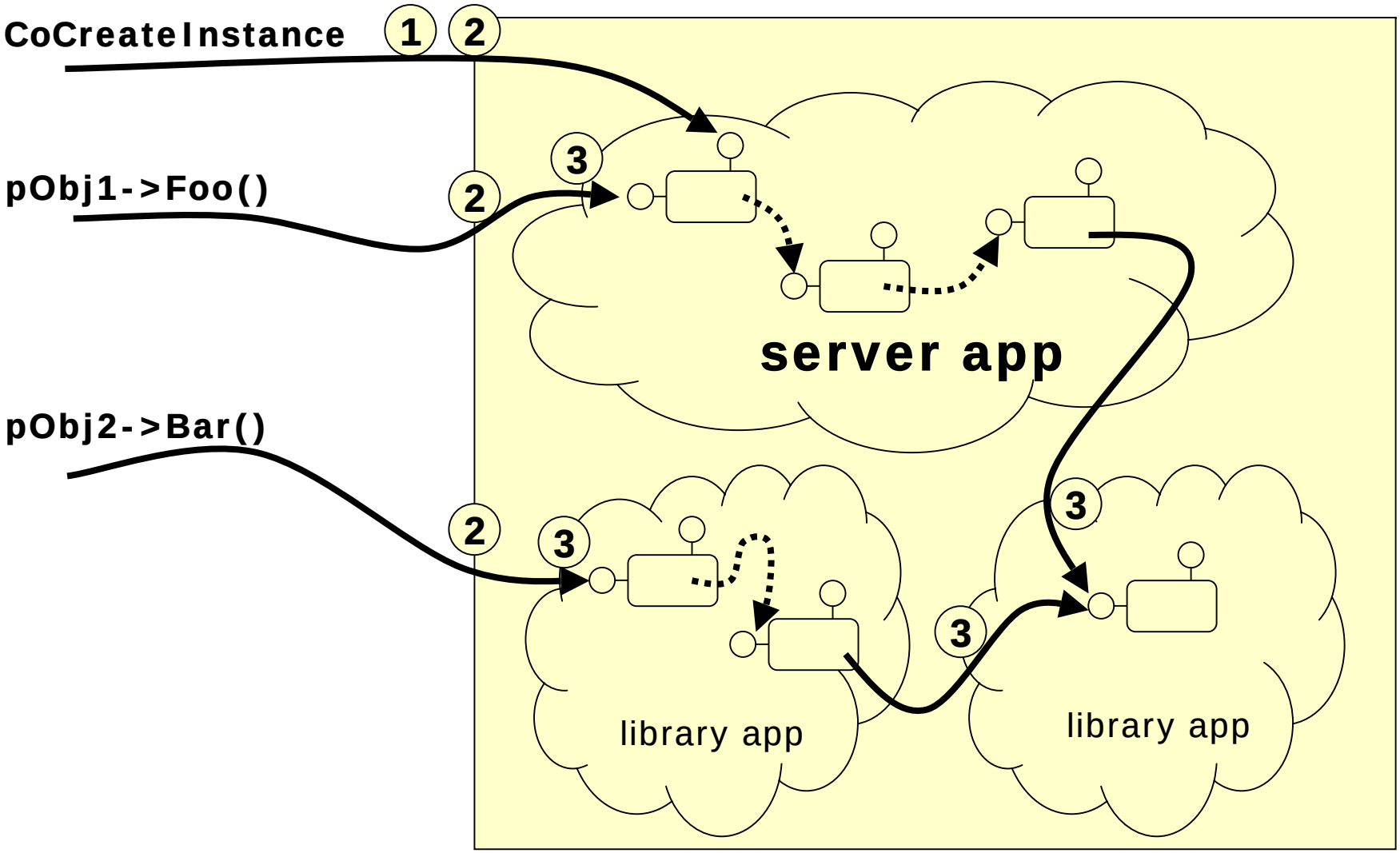


Role-based access checks

- **COM+ performs three stages of access checks**
  - 1) At server process launch, client must be member of a role
  - 2) At server process entry, client must be member of a role
  - 3) At application entry, client must be in a role that grants access to the class, interface, or method being called
- **Once inside an application, calls between objects are not checked (we assume they trust each other)**



Three stages of COM+ role-based access checks



Custom logic for role-based access checks

- **Server can query the roles of the caller at runtime**
  - Allows you to make runtime decisions based on your own logic

```
namespace System.EnterpriseServices {
  public sealed class ContextUtil {
    public static bool IsCallerInRole(string role);

    // ... other non-security related stuff
  }
}
```

Example

```
public class Bank {
  public void Withdraw(long accountID, long amount) {
    if (amount > 5000 && !ContextUtil.IsCallerInRole("Supervisor"))
      throw new SecurityException("Must be supervisor");

    // ... perform the withdrawal
  }
}
```

Enabling role-based access checks

- **Checks must be enabled at both the application and component level**
  - Otherwise all three stages of role-based checks are disabled
  - Also, IsCallerInRole always returns true



Deploying managed code in COM+

- **System.EnterpriseServices provides everything you need**
  - Derive component from ServicedComponent
  - Assembly needs a strong name
  - Assembly level attributes specify COM+ application settings
    - ApplicationName
    - ApplicationActivation
    - ApplicationAccessControl
  - Other attributes you need to know about
    - ComponentAccessControl
    - PrivateComponent
    - SecurityRole
    - SecureMethod



Example

```
[assembly: AssemblyDelaySign(true)]
[assembly: AssemblyKeyFile(@"..\..\pubkey")]
[assembly: ApplicationName("Pet Store")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(true,
  Authentication     = AuthenticationOption.Privacy,
  ImpersonationLevel = ImpersonationLevelOption.Identify,
  AccessChecksLevel  = AccessChecksLevelOption.ApplicationComponent)]

[ComponentAccessControl(true)]
[SecureMethod]
public class PetStore : ServicedComponent, IPetStore {
    [SecurityRole("Customers")] public void PetAnimal()     {}
    [SecurityRole("Customers")] public void BuyAnimal()     {}
    [SecurityRole("Staff")]     public void FeedAnimal()    {}
    [SecurityRole("Owners")]    public void GiveAwayMoney() {}
}
```

Deploying managed code in COM+

- **REGSVCS.EXE installs managed applications into the COM+ catalog**
  - Creates a type library
  - Creates (and configures) the application
  - Creates roles
  - Imports (and configures) all public classes into COM+ catalog
- **REGSVCS.EXE must be run with administrative privileges**

install → `regsvcs c:\appDir\petstore.dll`

remove → `regsvcs /u c:\appDir\petstore.dll`

Summary

- **COM+ provides a complete security framework**
  - Unlike System.Runtime.Remoting at the moment
- **Can easily deploy managed code in COM+**
- **COM+ isn't going away any time soon**



Dumb Code

How to avoid coding security holes into your apps



Outline

- **Fear user input**
  - buffer overflow
  - format string vulnerabilities
  - canonicalization errors
  - SQL injection
  - cross site scripting
- **Beware storing secrets**
- **Use good ACLs**
- **Watch for race conditions**
- **Keep attackers guessing**
- **Run with least privilege**



Fear user input

- **Must validate all input from any external, untrusted source**
  - best frame of mind is to assume it's purposely malformed
  - failure to do this is the lead cause of security vulnerabilities
- **Tests you can perform**
  - is the input the right size?
  - does the input conform to a legal format?
- **Application must carefully define what is legal input**
  - don't try to test for exceptional cases – you'll always miss one
- **Do your own "taint checking"**
  - take a lesson from Perl

Buffer overflows

- **Bug: failure to detect that input is bigger than buffer can hold**
- **Result: attacker sends arbitrary machine code and you run it**
  - This is the number one security problem in C and C++ code

```
void main() {
  foo();
}

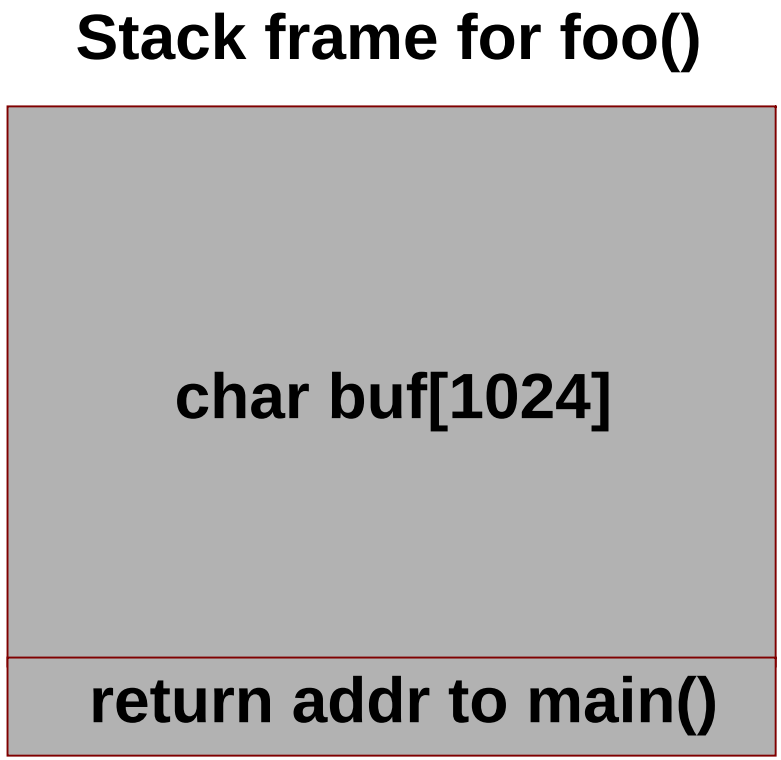
void foo() {
  char buf[1024];
  readUserName(buf);
  logUserName(buf);
}
```

A stack ripe for smashing

```
void main() {
  foo();
}

void foo() {
  char buf[1024];
  readUserName(buf);
  logUserName(buf);
}

void readUserName(char* buf) {
  strcpy(buf, form.name);
}
```

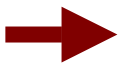


Smashing the stack and running arbitrary code

```
nop
nop
nop
open internet connection
download attack EXE
launch attack EXE
ExitProcess()
```

kernel32.dll:0x77E61EE8

Kernel32.dll



```
...
77E61EDE  push  ecx
77E61EDF  push  dword ptr ds:[77ED67E8h]
77E61EE5  push  dword ptr [eax+18h]
77E61EE8  call  edi
77E61EEA  cmp   eax,ebx
...
```

Smashing the heap

- **C++ apps have heaps that contain data and function pointers**
  - an object has state, but it also has a “vptr”
- **What if a buffer in the heap overflows?**
  - could overwrite vptrs
- **First function in vtable usually is virtual destructor**
  - Attacker reroutes destructor to point to other code
- **Bottom line: buffer overflows can lead to nasty security compromises, wherever they occur**



Finding buffer overflows in source code

- **Search for commonly misused functions**
  - strcpy, strcat, memcpy, sprintf
    - must check size of destination buffer manually
  - strncpy, strncat
    - size of buffer specified in characters, not bytes!
  - see list of functions in Appendix A of *Writing Secure Code*
- **Consider banning some functions from use!**



Format string vulnerabilities

- Bug: allowing the attacker to control a format string
- Result: attacker sends arbitrary machine code and you run it

- Spotting the bug:

```
printf(userSuppliedString);        // BUG BUG
printf("%s", userSuppliedString); // correct
```

- Never allow the user to specify any part of a format string



How a format string attack works

- **Format string causes printf to read values up the stack**
  - if no arguments passed to look at, printf is looking at stuff it shouldn't be looking at (local variables, return addresses, etc.)

```
// correct usage of printf
int i = 1, j = 2, k = 3;
char* s = "Hello world";
printf("%d%d%d%s", i, j, k, s);

// bad usage of printf
printf("%d%d%d%s");
```

this will likely crash the app when printf tries to dereference the fourth element on the stack, treating it as a string pointer

How a format string attack works

- **Format string can also cause printf to *write* to the stack**
  - via the somewhat esoteric (but fully supported) %n specifier

```
// correct usage of printf
int i = 1, j = 2, k = 3;
int bytesOutputSoFar;
printf("%d%d%d%n", i, j, k, &bytesOutputSoFar);

// incredibly bad usage of printf
printf("%d%d%d%n");
```

this causes printf to walk up the stack
and overwrite a four byte value with
the number of bytes it's output so far

Avoiding format string vulnerabilities

- **Never allow user input to creep into format strings**
  - Keep the attacker in the data channel
- **Check all usages of the following functions**
  - printf, wprintf, vprintf, _tprintf, etc.
  - sprintf, swprintf, vsprintf, _tsprintf, etc.
  - fprintf, fwprintf, vfprintf, _tfprintf, etc.
- **Check logging and tracing functions that take format strings**
  - where most vulnerabilities are being found in Unix today



Canonicalization errors

- Bug: failure to canonicalize resource names
- Result: various security policy violations

- A resource (files for example) can have many names:

```
C:\foo\bar\somelongfilename.txt
C:\foo\bar\somelongfilename.txt.
C:\foo\bar\somelo~1.txt
..\foo\bar\somelo~1.txt.
Z:\somelongfilename.txt::$DATA
\\server\share\bar\somelong%66ilename.txt
```

- Attackers often use alternate names to get access to files they shouldn't normally be allowed to see



Tips for avoiding canonicalization errors

- **Avoid making decisions based on user provided file names (or names of other resources)**
- **For server applications, don't let clients tell you where to find a file**
  - don't allow clients to specify full paths – you should always provide the base location yourself
  - don't rely on your PATH environment variable to find files
- **Don't let multi-byte encodings surprise you**
  - there are often several ways to encode a single character
- **Canonicalize user provided names carefully**
  - see CleanCanon.cpp from *Writing Secure Code*



SQL injection attacks

- **Bug: concatenating SQL commands with raw user input**
- **What's wrong with the following C# code?**
  - imagine this was the code behind an ASP.NET login form

```
string sql = "select * from users where name='" +
             txtName.Value +
             "' and password = '" +
             txtPwd.Value +
             "'";
cmd.CommandText = sql;
IDataReader reader = cmd.ExecuteQuery();
```

SQL injection attacks, cont.

- **Here's the input you'd expect to get from a legitimate user**
  - Name: `Bob`
  - Password: `nU3!gx7`
  - Resulting SQL:

```
select * from users where name='Bob' and pwd='nU3!gx7'
```

- **Here's what an attacker might send instead**
  - Name: `Bob'--`
  - Password: `hack`
  - Resulting SQL:

```
select * from users where name='Bob'--' and pwd='hack'
```

Avoid SQL injection attacks

- **Use parameterized queries or stored procs**
  - stop building SQL statements using string concatenation

```
// a parameterized query
// keeps the attacker out of the control channel
string sql = "select * from users where name=@n and pwd=@p";
cmd.CommandText = sql;
cmd.Parameters.Add(new SqlParameter("@n", name.Text));
cmd.Parameters.Add(new SqlParameter("@p", password.Text));
IDataReader reader = cmd.ExecuteQuery();
```

Never echo unfiltered input back as HTML

- **Known as “cross site scripting”**
- **The basic problem**
  - one can submit HTML that is then served to another
  - HTML can contain scripts
- **What can happen to a victim**
  - cookies can be stolen
  - COM objects can instantiated and scripted with untrusted data
  - user input can be intercepted
- **How to avoid cross site scripting**
  - all user input should be filtered, as usual
  - all output that may contain user data should be escaped
    - HttpServerUtility.HtmlEncode is your friend!



Don’t rely on client side validation for security

- **Know what client side validation is for**
  - gives clients a better user experience
  - reduces load on your server from accidental bad input
- **Client side validation provides no real security**
  - clients don’t have to use your form to submit requests
  - always validate input when it arrives at the server
  - think about Perl’s “taint checking” and try to apply the same ideas to your own code
- **ASP.NET validation controls are a great utility**
  - provides client with immediate feedback on errors via Jscript
  - provides server protection by validating input on server side



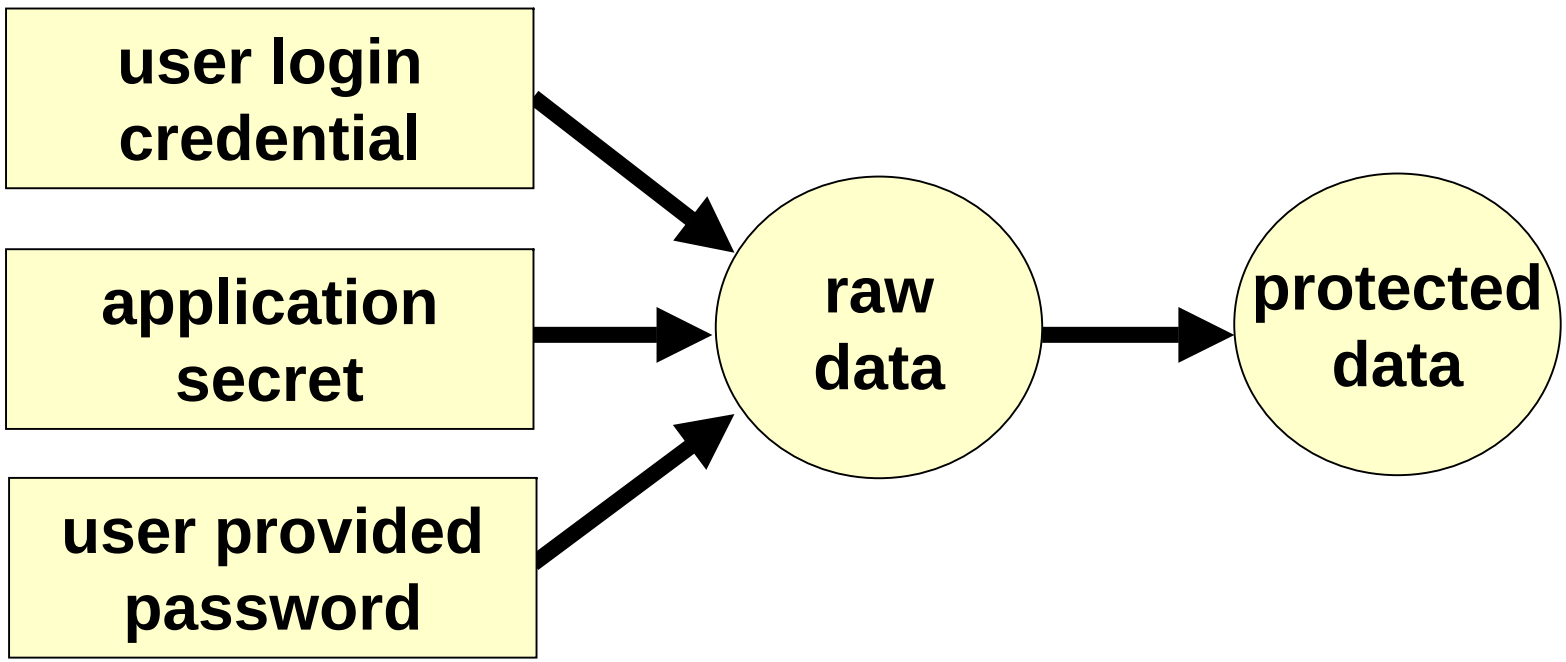
Beware storing secrets

```
<configuration>
 <system.web>
  <identity userName='Bob' password='HereIsMySecret'/>
 </system.web>
</configuration>
```

- **Storing secret data on a machine is tricky**
  - how do you protect it?
  - how do you read it?
  - what stops someone else from reading it as well?
- **Can encryption help?**
  - where do you store the encryption key?
- **Avoid storing sensitive data in config files**

Data protection API

- **DPAPI consists of two functions that simplify secret storage**
  - CryptProtectData, CryptUnprotectData
  - Supported on Windows 2000, XP, .NET Server
- **Protection derived from up to three sources**



ASP.NET and DPAPI

- **A patch for ASP.NET allows you to encrypt passwords**
  - See knowledge base article 329290
  - You'll need the patch and a tool called aspnet_setreg.exe
- **Uses DPAPI with the machine's credentials**
  - if the machine is compromised, the attacker can decrypt these strings
  - but if you absolutely must specify these passwords, this is better than using cleartext

```
<identity impersonate='true'
  userName='registry:HKLM\...\ASPNET_SETREG,userName'
  password='registry:HKLM\...\ASPNET_SETREG,password' />
```

Use good ACLs

- What should you pass for lpMutexAttributes?

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL bInitialOwner,
  LPCTSTR lpName
);
```

- Most sample code passes NULL
- Who will have access to the mutex if you do this?



Passing NULL for LPSECURITY_ATTRIBUTES

- **Passing NULL can do one of three things**
  - in a hierarchical system (files, registry, directory service) the parent object provides inheritable entries in its DACL
  - in a flat system (processes, mutexes, named pipes, etc.) the default DACL in the caller's token is used
    - SYSTEM granted full permissions
    - Default owner granted full permissions
  - might do something different depending on the type of object
- **Creating objects shared by different principals is tricky**
  - a mutex used to synchronize a service and a client app
  - a named pipe used to communicate over the network



The NULL DACL

- A NULL DACL is not the same as passing NULL

```
// creating a mutex with a NULL DACL
// allows anyone (even null sessions) to use it
SECURITY_DESCRIPTOR sd;
InitializeSecurityDescriptor(&sd,
  SECURITY_DESCRIPTOR_REVISION);
SetSecurityDescriptorDacl(&sd, TRUE, NULL, FALSE);
SECURITY_ATTRIBUTES sa = { sizeof sa, &sd, FALSE };
HANDLE h = CreateMutex(&sa, FALSE, "MyMutex");
```

```
// passing NULL to CreateMutex
// allows SYSTEM or the creator to use it
HANDLE h = CreateMutex(NULL, FALSE, "MyMutex");
```

Avoid using NULL DACLs

- **Granting full control to everyone opens several holes**
- **Anyone who can reference the object can:**
  - delete the object (if it's deletable)
  - change the DACL
  - take ownership away from you so you can't get it back
- **NULL DACLs are easy to code up, thus tempting**
- **Resist the temptation – figure out an access control policy that makes sense, then use it**
  - you can adjust the default DACL in your token to allow you to pass NULL and get a reasonable ACL for new objects
  - or use the low level ACL apis to create simple DACLs
  - use the access control editor UI for more sophisticated DACLs

Watch for race conditions

- **Timing bugs can lead to compromises**
  - Thread one loads plaintext into a buffer
  - Thread one begins to encrypt the buffer
  - Thread two begins to read the buffer before encryption is finished
- **Howard suggests using separate buffers for plaintext and ciphertext**
  - Thread two would be reading from a ciphertext only buffer
  - Race condition is still a bug, but doesn't compromise security



Keep attackers guessing

- **Security errors should lead to two distinct outputs**
  - vague message to user, with instructions on how to proceed
  - detailed error message to internal log
  - give user an identifier so tech support can find the detailed message
- **Avoid giving away free information**
  - one of the first stages of an attack is reconnaissance
  - banners or headers that give away system information should be omitted or changed wherever possible
  - you must still assume an attacker knows what you're running
    - TCP/IP fingerprinting is pretty effective
    - see http://uptime.netcraft.com/up/graph/ for an example



Run with least privilege

- **Code should run with minimum possible permissions**
  - by doing this, you're allowing the OS to erect walls around you
  - if your code has a bug, this limits the damage it can do
  - if an attacker exploits a bug, this limits the damage he can do
  - it's all about giving you time to detect and react (patch the bug)
- **Server code**
  - choose your process identity wisely
  - factor highly privileged code into a separate process
- **Mobile code**
  - make use of the CAS sandbox

Summary

- Fear user input
- Don’t rely on client-side validation for security
- Beware storing secrets
- Use good ACLs
- Don’t give attackers free information
- Run with least privilege