Securitate Software

X Windows security

(objects and file system)

Object Properties - Definitions

- **fundamental unit of abstraction for Windows resources**
- similar to the class/object concept in OOP (a type and more instances)
- Windows kernel object manager (KOM)
  - responsible for kernel-level management of objects
  - object types are called system objects or securable objects
- provide a **uniform view and access control mechanism** for all system resources, regardless of their type

Object Properties – System (securable) objects

- types
  - directory service objects, file-mapping objects
  - inter-process synchronization objects (Event, Mutex, Semaphore, WaitableTimer)
  - job objects, processes and threads, services
  - network shares, NTFS files and directories, registry keys
  - named and anonymous pipes, printers
- a complete list of object types got using WinObj utility
- instantiated /connected to using functions Create*() / Open*()
  - return an object handle (HANDLE)
- release objects done by CloseHandle()

Object Properties – Object Namespaces

- objects can be named or unnamed (anonymous)
- anonymous objects can be shared between processes only by duplicating an object handle or through inheritance
- named objects are stored in a hierarchical structure, called object namespace
- there are
  - a global namespace
  - there are more local namespaces, one for each Terminal Service
- object namespace's structure is similar to a file system
  - directories and sub-directories of objects
  - links (objects of SymbolicLink type)
- code audit: named objects are generally visible, though not necessarily accessible

Object Properties – Namespace Collisions

- also called **name squatting** attacks
  - application **opens an attacker created object**, instead of creating a new one
- Create*() functions supports both creation and opening
  - could lead to vulnerabilities
  - creation uses a SECURITY_ATTRIBUTES structure, which is ignored if object exists
  - creation flags provided to avoid opening an existing object
- code audit
  - understand semantic of each Create*() function individually
  - check if they correctly set flags and check for return values

Object Properties – Private Object Namespace

- **avoid name squatting attacks** on the global namespace
  - though, do not protect objects with weak access control
- private namespace **uniquely identified** by a **name** and a **boundary descriptor**
  - there could be namespaces with the same name, but with different boundary descriptor
  - the boundary contains at least one security identifier (SID)
- object name preceded by “namespace\”, like “NS0\MyMutex”
- a process can open an existing namespace even if it is not within the boundary
  - if access not restricted by the SECURITY_ATTRIBUTES parameter at creation
- functions
  - CreatePrivateNamespace(), OpenPrivateNamespace()
  - CreateBoundaryDescriptor(), AddSIDToBoundaryDescriptor()

Object Handles – Object Handle Manipulation

- at creation/opening, the object is referred by its name
  - return an object handle
- any subsequent operations are based on the object handle
- system maintains a list of open handles, categorized by the owning process
- duplicating a handle requires PROCESS_DUP_HANDLE permission for both the source and destination processes

Object Handles – INVALID_HANDLE_VALUES versus NULL

- Windows API functions are inconsistent
  - an error results in a NULL or an INVALID_HANDLE_VALUE (-1)
- examples
  - CreateFile() returns INVALID_HANDLE_VALUE when encounters errors
  - OpenProcess() returns NULL on errors
- code audit: each function documentation must be consulted

Object Handles – ex. Wrong way to check for return value

```
HANDLE lockUserSession(TCHAR *szUserPath)
{
    HANDLE hLock;
    hLock = CreateFile(szUserPath, GENERIC_ALL, 0, NULL, CREATE_ALWAYS,
                       FILE_FLAG_DELET_ON_CLOSE, 0);
    return hLock;
}

BOOL isUSerLoggedIn(TCHAR *szUserPath)
{
    HANDLE hLock;
    hLock = CreateFile(szUserPath, GENERIC_ALL, 0, NULL, CREATE_NEW,
                       FILE_FLAG_DELET_ON_CLOSE, 0);

    if (hLock == NULL)
      return TRUE;
    CloseHandle(hLock);
    return FALSE;
}
```

Object Handles – Handle Inheritance

- **no special default privileges** or shared object access **to a child process**
- **handles inherited only by explicit configurations**
  - set true the bInheritable parameter of the CreateProcess()
  - only handles marked as inheritable are duplicated in child process
- handle **inheritance configurations**
  1. set true the bInheritable field of the SECURITY_ATTRIBUTES structure at object creation
  2. use DuplicateHandle() with a true bInheritable argument
- **inherited handles could be a security issue**
  - for children run under another security context than their parent

Object Handles – Handle Inheritance (2)

- code audit
  - identify inheritable handles
  - identify overlaps of inheritable handles lifespan with creation of child process
  - risks: child processes run in a separate security context, which inherit handles
  - useful tool: Process Explorer
- good practice
  - never create inheritable handles at object instantiation
  - duplicate, if needed, just before child process creation
  - close the inheritable handle after child creation

Object Handles – Handle Inheritance, vulnerable example

```
int tclient(HANDLE io)
{
  int hr = 0;
  HANDLE hStdin, hStdout, hStderr;
  HANDLE hproc = GetCurrentProcess();

  // drop privileges
  if (!ImpersonateNamedPipeClient(io))
    return GetLastError();

  // create inheritable handles
  DuplicateHandle(hProc, io, hProc, &hStdin, GENERIC_READ, TRUE, 0);
  DuplicateHandle(hProc, io, hProc, &hStdout, GENERIC_WRITE, TRUE, 0);
  DuplicateHandle(hProc, io, hProc, &hStderr, GENERIC_WRITE, TRUE, 0);

  CloseHandle(io);

  // create a child process that inherits inheritable handles
  hProc = CreateRedirectedShell(hStdin, hStdout, hStderr);

  // close duplicated handles
  CloseHandle(hStdin);
  CloseHandle(hStdout);
  CloseHandle(hStderr);

  // regaing privileges
  hr = RevertToSelf();
```

Object Handles – Handle Inheritance, vulnerable example (2)

```
  // wait for child process' termination
  if (hProc != NULL)
    WaitForSingleObject(hProc, INFINITE);

  return hr;
}
```

- suffer a race condition vulnerability
- while in CreateRedirectedShell()
  - inheritable handles prepared for “client 1” (privileged)
  - could also be inherited by a concurrent child process for “client 2” (non-privileged)

Sessions – Handling of multiple logged-on users

- **each logged on user is associated a session**
- a session encapsulates data relevant to a logon instance
  - info for **governing process access rights**
  - data accessible to constituent processes in a session
  - selected behavioral characteristics for a process started in a session
- sessions **isolate users from each other**

Sessions – Security Identifiers (SID)

- **uniquely identifies an entity** (security “principal”)
  - e.g. users, service accounts, groups, machines
- used to **determine who has access to what**
- SID structure
  - revision level
  - identifier authority value
  - variable-length subauthority
  - relative ID (RID)
- often represented in text format

S-<revision>-<identifier authority>-<subauthority>-<RID>

- functions: ConvertStringSidToSid() and ConvertSidToStringSid()

Sessions – Ex. Of well-known SIDs

Administrator: S-1-5-<domain ID>-500

Administrators group: S-1-5-32-444

Users group: S-1-5-32-545

Everyone group: S-1-1-0

Local system acount: S-1-5-18

Local service account: S-1-5-19

Local network account: S-1-5-20

Sessions – Logon Rights

- determine whether
  - a user can establish a logon session on a machine and
  - what type of session is allowed
- can be viewed in “Local Security Policy” editor
  - “Local Policy” ! “User Rights Assignment”
- examples
  - SeNetworkLogonRight
  - SeRemoteInteractiveLoginRight
  - SeBatchLogonRight
  - SeInteractiveLogonRight

Sessions – Access Tokens

- system objects that **describe the security context for a process or thread**
  - used to to identify the user
  - when a thread interacts with a securable object or
  - tries to perform a system task that requires privileges
- **determine** if a process or thread
  - **can access a securable object** or
  - **perform a privileged system task**
- each process/thread can optionally change certain attributes in its access token
  - using functions like AdjustTokenGroups() and AdjustTokenPrivileges()

Sessions – Access Tokens Types

1. **primary access token**
   - **created when a user starts a new session**
   - assigned to all processes started in a session
   - a new copy created for each new process/thread
   - could be obtained using the OpenProcessToken() function
2. **impersonation token**
   - **associated to a thread that impersonate a client account**
   - allows the thread to interact with securable objects using the client's security context
   - an impersonation thread has both a primary token and an impersonation token
   - could be obtained using the OpenThreadToken() function

Sessions – Access Token Main Components

- security identifier (SID) of the associated user's account
- SID list of groups the user belongs to
- session SID
- privilege list
- owner SID
- SID of the primary group
- default DACL (used when a process creates a securable object
- without specifying a security descriptor)
- type: primary or impersonation
- restricting SID list

Sessions – Access Token Privileges

- SeAssignPrimaryTokenPrivilege: assign the primary access token for a process/thread
- SeAuditPrivilege: generate security logs
- SeBackupPrivilege: create backups
- SeChangeNotifyPrivilage: be notified when certain files or folders are changed
- SeDebugPrivilege: attach and debug processes
- SeIncreaseBasePriorityPrivilege: increase the scheduling priority of a process
- SeLoadDriverPrivilege
- SeShutdownPrivilege
- SeSystemTimePrivilege
- SeTakeOwnershipPrivilege

Sessions – Access Token Group List

- the list of SIDs for all the associated user's group membership
- **used to check access permission rights** of a process
  - when the process attempts to access an object
  - the object's DACL is checked against entries in the group list of the process' access token
- **generated at logon**
- **cannot be updated during a session**, though **can be altered**
  - by manipulating their group SID attributes
  - e.g.: disable, if not mandatory
- **group SID attributes**
  - SE_GROUP_ENABLED
  - SE_GROUP_ENABLED_BY_DEFAULT
  - SE_GROUP_LOGON_ID
  - SE_GROUP_MANDATORY
  - SE_GROUP_OWNER
  - SE_GROUP_RESOURCE
  - SE_GROUP_USE_FOR_DENY_ONLY

Sessions – Restricted Access Tokens

- an access token having a **subset of the privileges and access rights of its original token**
  - **has a nonempty restricted SID list**
- created with the CreateRestrictedToken() function
  1. establish deny-only group SIDs by turning
     - on their SE_GROUP_USE_FOR_DENY_ONLY attribute
     - off their SE_GROUP_ENABLED attribute
  2. revoke any privilege currently assigned
  3. add SIDs to the restricting SID list
- setting the SE_GROUP_USE_FOR_DENY_ONLY on mandatory group SIDs
  - prevent an account using its own SID for granting access to a resource

Sessions – Restricted Access Tokens (2)

- **access is granted** only if requested access rights allowed **by checking both**
  - the token's **enabled SIDs**
  - the list of **restricting SIDs**
- any process can create a restricted access token
- a restricted token prevents the token from being reset to its original (default) group list and privilege state

Sessions – Running Under Different Contexts

- the capability to change the current thread's token or create a new process under a different token
- **processes running in a new user session**
  - functions: CreateProcessWithLogonW() and LogonUser()
  - logon types: LOGON32_LOGON_BATCH, LOGON32_LOGON_INTERACTIVE, LOGON32_LOGON_NETWORK, LOGON32_LOGON_SERVICE
- **processes with restricted privileges**
  - functions: CreateProcessAsUser() or CreateProcessWithTokenW()
- **threads impersonating other users**
  - call SetThreadToken() with a restricted token
  - run with a privileges of a client (of a server) using functions like ImpersonateNamedPipeClient(), ImpersonateLoggedOnUser()

Security Descriptors - Definition

- provide granular access control for securable objects
- consists of
  - owner SID
  - group SID
  - discretionary access control list (DACL)
  - security access control list (SACL)

Security Descriptors - Access Control Entries (ACE)

- elements in ACLs
- consists of
  - SID (whom is applied)
  - type: allow and deny
  - access mask (what is allowed or denied)
  - inheritance related flags

Security Descriptors – Access Mask

- a bit field named ACCESS_MASK in the ACE structure
- divided into three categories
  - **generic** access rights
  - **standard** access rights
  - **specific** access rights

Security Descriptors – Generic Access Rights

- types
  - GENERIC_ALL
  - GENERIC_READ
  - GENERIC_WRITE
- GENERIC_EXECUTE
- translated into **a combination of**
  - **specific** and **standard** access rights
  - example for files: GENERIC_READ = READ_CONTROL, SYNCHRONIZE, FILE_READ_DATA, FILE_READ_EA, FILE_READ_ATTRIBUTES

Security Descriptors – Standard Access Rights

- **apply to any sort of object**
- define access to pieces of object control information rather than the object data itself
- composed by 8 bits, from which only 5 in use
  - DELETE: delete the object
  - READ_CONTROL: read security information
  - WRITE_DAC: write to the object's DACL
  - WRITE_OWNER: change the owner
  - SYNCHRONIZE: use object for synchronization
- constants of combined standard access rights
  - **STANDARD_RIGHTS_ALL**: DELETE, READ_CONTROL, WRITE_DAC, WRITE_OWNER, SYNCHRONIZE
  - **STANDARD_RIGHTS_EXECUTE**: READ_CONTROL
  - **STANDARD_RIGHTS_READ**: READ_CONTROL
  - **STANDARD_RIGHTS_REQUIRED**: DELETE, READ_CONTROL, WRITE_DAC, WRITE_OWNER
  - **STANDARD_RIGHTS_WRITE**: READ_CONTROL

Security Descriptors – Specific Access Rights

- bits 0-15 in ACCESS_MASK
- depends on the object

Security Descriptors - ACL Inheritance

- objects can be containers for other objects
- examples: directories and registry keys
- Windows defines permissions that apply to child objects
- types
  - CONTAINER_INHERIT_ACE
  - INHERIT_ONLY_ACE
  - INHERITED_ACE
  - NO_PROPAGATE_INHERIT_ACE
  - OBJECT_INHERIT_ACE

Security Descriptors - Low-Level ACL Control API

- AddAce(): add ACEs to an ACL

  BOOL AddAce (PACL pAcl, DWORD dwAceRevision, DWORD dwStartingAceIndex,
  LPVOID pAceList, DWORD nAceListLength);

- AddAccessAllowedAce(): appends an allow ACE to an ACL

  BOOL AddAccessAllowedAce(PACL pAcl, DWROD dwRevision, DWORD AccessMask, PSID pSid);

- AddAccessDeniedAce(): appends a deny ACE to an ACL

  BOOL AddAccessDeniedAce(PACL pAcl, DWROD dwRevision, DWORD AccessMask, PSID pSid);

- GetAce: gets an ACE from an ACL

  *BOOL GetAce(PACL pAcl, DWORD dwAceIndex, LPVOID *pAce);*

- SetSecurityDescriptorDacl(), SetEntriesInAcl(),
- GetNamedSecurityInfo(), SetNamedSecurityInfo()
- see a complete list at MSDN Low-level Access Control Functions

Security Descriptors - High-Level API: Security Descriptor Strings

- allow specifying security descriptors as **human understandable text strings**
  - **encoding its fields and attributes**
- based on the **security descriptor definition language** (SSDL)
  - see details on the MSDN page
- functions
  - ConvertSecurityDescriptorToStringSecurityDescriptor()
  - ConvertStringSecurityDescriptorToSecurityDescriptor()
- the security descriptor string format

  O:owner_sid

  G:group_sid

  D:dacl_flags(string_ace_1)...(string_ace_n)

  S:sacl_flags(string_ace_1)...(string_ace_n)

Security Descriptors - High-Level API: Security Descriptor Strings (2)

- the ACE string format

  ace_type;ace_flags;rights;object_guid;inherit_object_giud;sid

  - type: ’A’ (allow) and ’D’ (deny)
  - flags: indicate ACE’s properties
  - rights:
    - generic: ’GR’ (GENERIC_READ), ’GW’ (GENERIC_WRITE), ’GX’ (GENERIC_EXECUTE), ’GA’ (GENERIC_ALL_ACCESS)
    - standard: “RC” (READ_CONTROL), “SD” (DELETE), “WD” (WRITE_DAC), “WO” (WRITE_OWNER)
    - specific: object-specific encoding
  - sid: SID the ACE applies to
- example of an ACE string

  (A;;GR,GW;;;S-1-0-0)
- example of a DACL string

  D:P(D;OICI;GA;;;BG)(A;OICI;GA;;;SY)

  (A;OICI;GA;;;BA)(A;OICI;GRGWGX;;;IU)

Security Descriptors - Code Audit on ACLs

- examine the list of access control entries (ACE) in ACLs to identify permissions associated with a resource
  - account for every ACE in an ACL
  - if cannot determine why an ACE is in ACL, thet ACE should be removed
- determine both immediate and inherited permissions

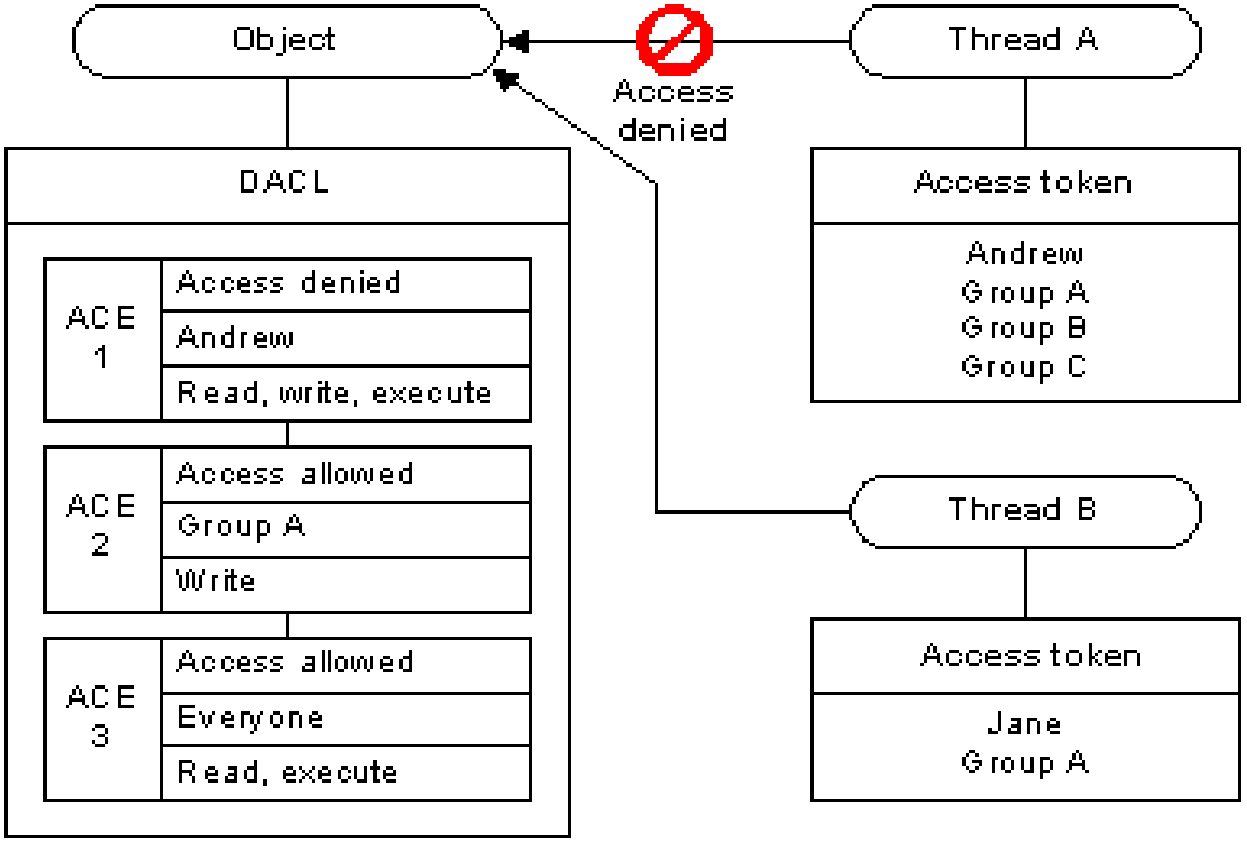
Security Descriptors – No Permissions

- **NULL DACL: allow any type of access to anyone**
  - exposed to interference by rogue applications
  - can lead to exposure of information, privilege escalation etc.
  - allow arbitrary change of the object's owner and ACLs
- **NON-NULL DACL: restrictive by default**
  - **an empty DACL allow no access**
  - until an allow ACE grants access
- difference between an empty and a NULL DACL
  - NULL: public, full access
  - EMPTY: restrict everyone
- providing a NULL pointer for a SECURITY_ATTRIBUTES structure at process creation
  - -> security descriptor with inherited and default attributes

Security Descriptors – ACE Order

- an ACL is an ordered list of ACEs
  - evaluated following that order
- **correct order**
  - **place deny entries before any allow entries**
- **access rights are evaluated only when an object is opened**, not when an existing handle is used
  - -> existing handles could be used even if objects permissions are changed
- DACL evaluation
  - current ACE's SID is compared against the token's SIDs
    - the ACE's access mask is used if SID is found
  - access is denied if matching ACE is a deny entry
  - access is allowed if the collection of ACEs contains all bits in the requested access mask
  - repeat on the next ACE if not decided yet
  - access is denied if end of the list is reached and the collection of matching ACEs does not contain all bits in the access mask

Security Descriptors – Example of DACL Evaluation



Processes and Thread Management - Definition

- just a container for threads
- described by attributes
- thread is the basic unit of execution
- all threads in a process share the same address space and security properties

Processes and Thread Management – Process Loading

- CreateProcess() is the common method to start a new process
- the second parameter is the command line
  - also contains the executable's path
- security issue: unquoted path containing spaces
  - leave the possibility for executing unintended programs
- example and the order in which executable is searched for
- CreateProcess(NULL, "C:\\Program Files\\My Applications\\my app.exe", …);
- C:\\Program.exe
- C:\\Program Files\\My.exe
- C:\\Program Files\\My Applications\\my.exe
- C:\\Program Files\\My Applications\\my app.exe
- correct form
- CreateProcess(NULL, "\"C:\\Program Files\\My Applications\\my app.exe\"", …);

Processes and Thread Management – Process Loading (2)

- a privilege program is vulnerable to this type of attack (privilege escalation) if the attacker is allowed to write in any directory in the path

Processes and Thread Management – ShellExecute() and ShellExecuteEx()

- also used to start new processes
- result in indirect use of CreateProcess()
- use Windows Explorer shell API (“open”, “edit”, “explore”, “search”)
- determine, based on file type, which application to launch
- code audit: take care that these functions to not necessarily (especially in case of no executable files) run the supplied file

DLL Loading – Security Issues

- result from the way Windows searches for a DLL during the loading process
- DLL search order
  - application load directory
  - current directory
  - “system32” directory
  - “Windows” directory
  - directories in PATH
- attack way: cause the run of an application in a directory where the attacker can write (DLL) files
  - creates a malicious DLL with the same name as a system DLL
  - makes a victim user to run a command in the attacker-controlled directory
  - the application will load the malicious DLL

DLL Loading – Security Issues (2)

- protection features (introduced from Windows XP)
  - SafeDllSearchMode changes the search order (current directory is searched only before those in PATH)
  - SetDllDirectory() places restrictions on a runtime-loaded DLL
  - LoadLibraryEx()

DLL Loading – DLL redirection

- address the common issues with DLL versioning (“DLL hell”)
- introduced security issue: a redirection file causes loading of an alternate set of libraries, even when a qualified path is provided in LoadLibrary()
- redirection file/directory
  - located in the same directory as the application
  - its name is the application’s name with “.local” extension
  - its contents is ignored
  - causes DLLs in current directory to be loaded in preference to any other locations
- redirection is superseded by an application manifest
  - an XML file
  - named as application with extension “.manifest”
  - includes a list of required libraries with specific version numbers

DLL Loading – DLL redirection (2)

- Windows XP and later prevent redirection of any DLLs listed in the registry key “HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs”
- vulnerabilities
  - the possibility of an attacker to write a file in the library load path that take precedence over the intended DLLs

Services - Definition

- a background process typically stared automatically during startup
- started by the Service Control Manager (SCM)
- can be configured to run under alternate accounts
- Windows applications handle privileged operations by creating a
- service that exposes an IPC interface for lower privileged process
- almost always run with some degree of elevated privileges
- typically expose some form of an attacker-facing interface
- most attacks on a Windows focus on compromising a service

Services – Service Control Permissions

- permissions for controlling a service can be granted to individual users and groups
- possible vulnerability: the ability to start a vulnerable service (e.g. “Network Dynamic Data Exchange”)
- during initialization services are often more vulnerable to a variety of attacks (e.g. object squatting and TOCTOU)
- code audit: identify any service that allow control commands to any non-administrative user
- useful tool: sdshow command of the sc.exe command-line utility

Services – Service Image Path

- it is the command-line used to run a service
- it is set when installing a service
- contains the executable path followed by arguments
- being started with CreateProcess() faces the same
- vulnerabilities like it (e.g. pathnames with unquoted spaces)
- could be seen using the qc command of the sc.exe utility

File Permissions

- files are treated as objects
- object permissions describe the permissions for the physical file
- some specific access rights
  - FILE_ADD_FILE, FILE_ADD_SUBDIRECTORY
  - FILE_ALL_ACCESS
  - FILE_APPEND_DATA
  - FILE_CREATE_PIPE_INSTANCE
  - FILE_DELETE_CHILD
  - FILE_EXECUTE, FILE_TRAVERSE
  - FILE_LIST_DIRECTORY
  - FILE_READ_ATTRIBUTES, FILE_WRITE_ATTRIBUTES
  - FILE_READ_DATA, FILE_WRITE_DATA
- specified at CreateFile()
- code audit: correlate permissions applied to a new file with what entities having that rights

File I/O API – the API Functions

- use file handles
- main functions: CreateFile(), ReadFile(), WriteFile(), CloseHandle()
- code auditing: the most important is CreateFile()

```
HANDLE CreateFile (LPCSTR lpFileNAme, DWORD dwDesiredAccess,
                   DWORD dwSharedMode,
                   LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                   DWORD dwCreationDisposition,
                   DWORD dwFlagsAndAttributes,
                   HANDLE hTemplateFile);
```

File I/O API – File Squatting

- if inappropriate parameters of CreateFile() are used
  - **an application could open an existing file instead of creating it**
  - specified **access rights are ignored** in case of opening an existing file
- conditions of vulnerabilities
  1. any setting of dwCreationDisposition excepting CREATE_NEW
  2. the location where file is to be created is writable by potential attackers
- example of vulnerable code

```
BOOL CreateWeeklyReport(PREPORT_DATA rData, LPCSTR filename)
{
  HANDLE hFile;
  hFile = CreateFile(filename, GENERIC_WRITE, 0, NULL,  CREATE_ALWAYS,
                                  FILE_ATTRIBUTE_ARCHIVE, NULL);
}
```

File I/O API –Canonicalization

- the process of turning a pathname into its simplest absolute form
- it is risky to use untrusted data to construct relative pathnames
- example of vulnerable code
  - let the user control the “beginning of” a filename
  - attacker could simply provide an absolute path

```
char *ProfileDirectory = "c:\\profiles\\";
BOOL LoadProfile (LPCSTR UserName) {
   HANDLE hFile;
   if (strstr(UserName, ".."))
      die("invalid username: %s\n", UserName);
   SetCurrentDirectory(ProfileDirectory);
   hFile = CreateFile(UserName, GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
}
```

File I/O API –Canonicalization (2)

- CreateFile() canonicalizes any directory traversal components before validating whether each path segment exists
  - nonexistent paths could be supplied in the filename argument as long as they are eliminated during canonicalization
    - “c:\nonexistent\path\..\..\file.txt” ! “c:\file.txt”
  - example of vulnerable code
    - allows for **directory traversal** using “\..\..\..\test”

```
char *ProfileDirectory = "c:\\profiles\\";
BOOL LoadProfile (LPCSTR UserName) {
   HANDLE hFile;
   char buf[MAX_PATH];
   if ((strlen(UserName) > MAX_PATH - strlen(ProfileDirectory) -12)
      die("invalid username: %s\n", UserName);
   _snprintf(buf, sizeof(buf), "%s\\prof_%s.txt", ProfileDirectory, UserName);
   hFile = CreateFile(UserName, GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
}
```

File I/O API – File-like Objects

- several non-file objects can be opened like files
  - pipes, mailslots, volumes, tape drives
- they do not appear in the file system, but only in the object namespace
- special filename format: “\\host\object”
  - local host is specified by “.”
- example for pipes: “\\.\pipe\pipename”
- attacking such objects requires control of the first segment of the pathname

File I/O API – Device Files

- special entities that
  - reside in the “file hierarchy”
  - give access to virtual of physical devices
- do not exist on the file system
- represented by file objects in the object namespace
- types
  - COM1-9
  - LPT1-9
  - CON
  - CONIN$
  - CONOUT$
  - PRN
  - AUX
  - CLOCK$
  - NUL

File I/O API – Device Files (2)

- pathnames are searched for such special names as filename and the rest of the pathname and extension are ignored
  - device file's names could be prepended by any pathname
  - device file's names could have any extension appended
  - vulnerable code: UserName could be a device file name

```
HANDLE OpenProfile(LPCSTR UserName) {
   HANDLE hFile;
   char path[MAX_PATH];
   if (strstr(UserName, ".."))
      die("bad username");
   _snprintf(path, sizeof(path), "%s\\profiles\\%s.txt", ConfigDir, UserName);
   hFile = CreateFile (path, GENERIC_READ, FILE_SHARE_READ, NULL,
                       OPEN_EXISTING, 0, NULL);
}
```

File I/O API – Check What is Open

1. check type: avoid opening special files as regular
   - functions: GetFileAttributes(), GetFileAttributesEx(), and GetFileType()
2. use Universal Naming Convention (UNC): starts name with "\\?\UNC\"
   - + avoiding opening a device file
   - + skips certain checks: if a DOS device file, special filename
   - +/- does not accept relative paths
   - - might create paths inaccessible via traditional DOS-style

File I/O API – File Streams

- alternate data streams (ADS)
- stream = a named unit of data
- default data stream is nameless
  - referred by default by the filename
- stream’s name format: “filename:stream_name[:stream_type]”
  - the only valid type: “$DATA”
  - example: “file:extra_info”

File I/O API – Extraneous Filename Characters

- trailing spaces (’ ’) and dots (’.’) are **striped out silently by CreateFile()**
- examples
  - “file ” ! “file”
  - “file.....” ! “file”
  - “file. ... ” ! “file”
- trailing spaced and dots are not removed if the filename is followed by an alternate name
  - "c:\test.txt....::$DATA.. ") "c:\test.txt...."
- possible vulnerabilities: could allow an attacker to choose arbitrary file extensions based on
  - path truncation
  - alternate file streams

File I/O API – Extraneous Filename Characters Attacks

- example 1: vulnerable code allowing creation of files with any extension

```
BOOL OpenUserProfile(LPCSTR UserName)
{
   HANDLE hProfile;
   char buf[MAX_PATH];
   if (strstr(UserName, ".."))
      return FALSE;
   _snprintf(buf, sizeof(buf), "%s\\%s.txt", ProfilesDir, UserName);
   buf[sizeof(buf) - 1] = '\0';
   hProfile = CreateFile (buf, GENERIC_ALL, FILE_SHARE_READ, NULL,
                                    CREATE_ALWAYS, 0, NULL);
}
```

- attack: a file name with any extension followed by a big number of spaces to cut off the intended ".txt"

File I/O API – Extraneous Filename Characters Attacks (2)

- example 2: vulnerable code allowing getting secret files

```
HANDLE GetRequestedFile(LPCSTR requestedFile)
{
   if (strstr(requestedFile, ".."))
      return INVALID_HANDLE_VALUE;
   if (! strcmp(requestedFile, ".config"))
      return INVALID_HANDLE_VALUE;
   return CreateFile(requestedFile, GENERIC_READ, FILE_SHARE_READ,
         NULL, OPEN_EXISTING, 0, NULL);
}
```

- attack ".config " or ".config::$DATA"