OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Secure Programming

Initialization

Prof. Dr. Christoph Skornia
christoph.skornia@hs-regensburg.de

Safe Initialization

System Environment:

- standard C runtime library defines a global variable, `environ` , as a NULL-terminated array of strings
- Linux is providing a declaration in *unistd.h*
- You can gain access to the variable by including the following extern statement in your code:

```
extern char **environ;
```

- Several functions defined in *stdlib.h*, such as `getenv( )` and `putenv( )`, provide access to environment variables, and they all operate on this variable.
- be aware of any environment variables that will be used by code you're using
- In particular, dynamic loaders on ELF-based Unix systems and most standard implementations of `malloc( )` all recognize a wide variety of environment variables that control their behavior.
- Where is the problem?

Safe Initialization

Example:

1. Program, which will run with admin/root right is using $HOME from users environment to create a file in the users directory with write access for the executing user
2. User is modifying $HOME to e.g. /bin (on Unix)
3. File is generated in /bin and user has write access to it !

Solution:

- Sanitize used variables:

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <pwd.h>

int main(int argc, char *argv[  ]) {
    uid_t          uid;
    struct passwd *pwd;

    uid = getuid(  );
    printf("User's UID is %d.\n", (int)uid);
    if (!(pwd = getpwuid(uid))) {
        printf("Unable to get user's password file record!\n");
        endpwent(  );
        return 1;
    }
    printf("User's home directory is %s\n", pwd->pw_dir);
    endpwent(  );

    return 0;
}
```

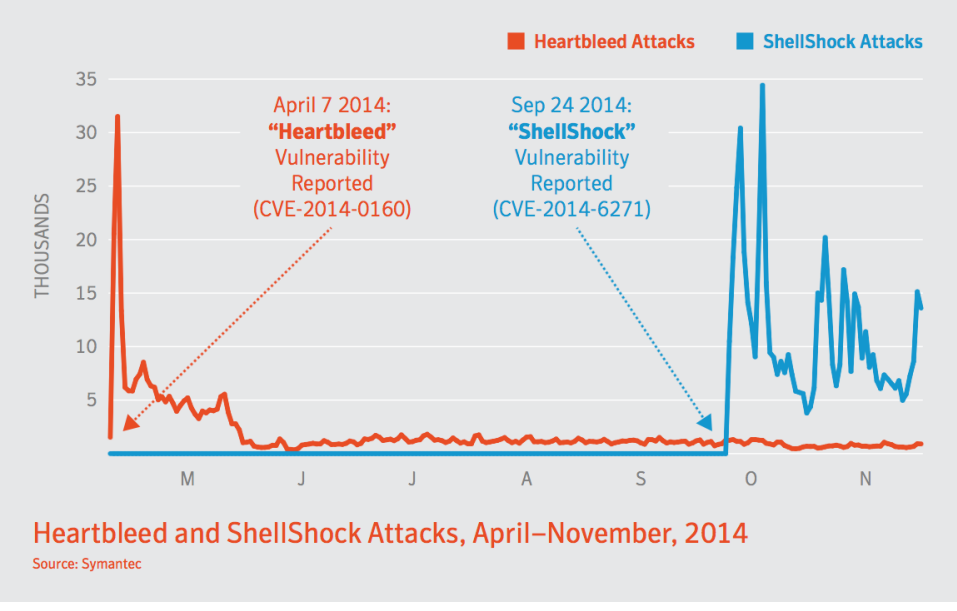
Safe Initialization

Advise: Decide carefully whether to use environment variables at all and make sure that you analyze any option of abuse.

Always sanitize:

- `PATH`
  - used by the shell and some of the exec*( ) family of standard C functions to locate an executable if a path is not explicitly specified.
  - should never include relative paths.
  - always force the setting of the PATH environment variable to `_PATH_STDPATH`, which is defined in paths.h.
- `IFS`
  - used by many shells to determine which character separates command-line arguments.
  - Modern Unix shells use a reasonable default value for IFS if it is not already set.
  - set it to something sane, such as a space, tab, and newline character.

Epic Fail: Shellshock



Heartbleed and ShellShock Attacks, April–November, 2014

Source: Symantec

```
[[env]] x='() { :;}; echo shellshockverwundbar' bash -c ""
```

Safe Initialization

Typical Sanitization

1. Create new environment e.g.

```
char    **new_environ…
```

2. Set needed values in `new_environ` to sane values
   - by setting to well controlled values (like `_PATH_STDPATH`) or
   - by copying from environ

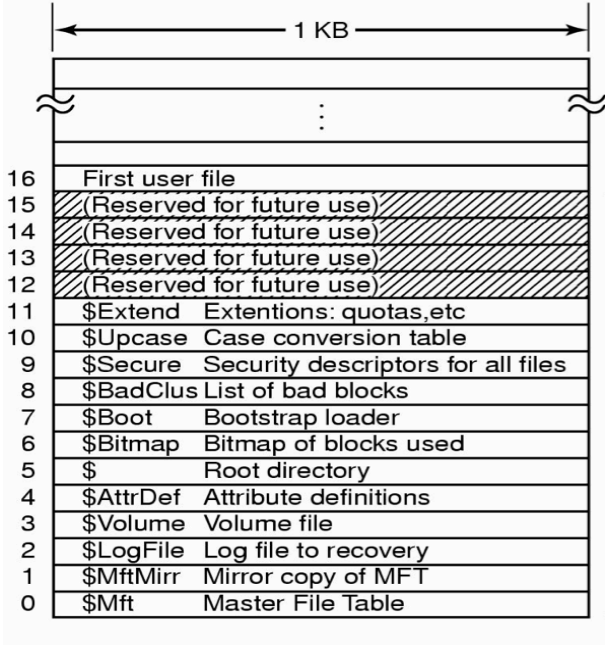
1. overwrite original environment

```
environ = new_environ;
```

Notice: also server-applications might pass environment- variables to modules started by them. E.g. cgi-scripts inherit environment from web-server…

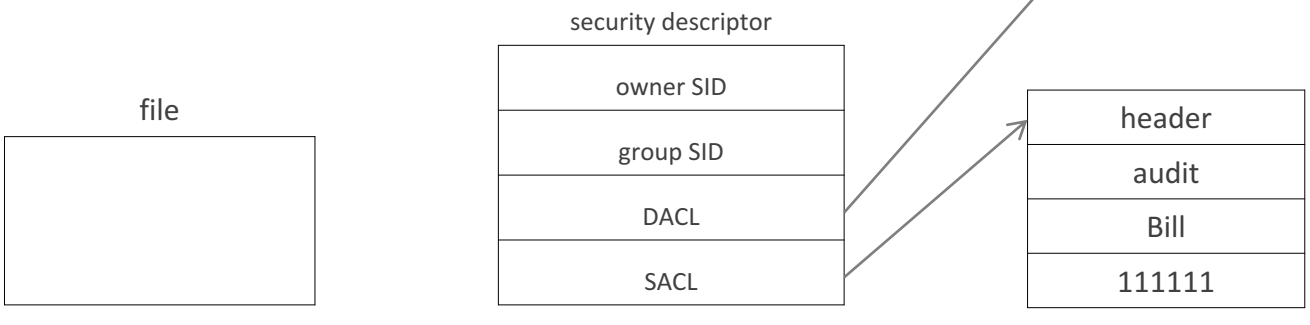
Authorization in Windows

- access management in Windows 2000/XP/Vista/7
  - ACL with rights for users and groups
  - saved in security-descriptors in the MFT
- access control:
  - AccessCheck-operation performed by the security-reference monitor
    - input:
      - security-descriptor of operations target
      - access-token of the process which wants to access the object Desired-Access-Mask
    - output:
      - allow, deny

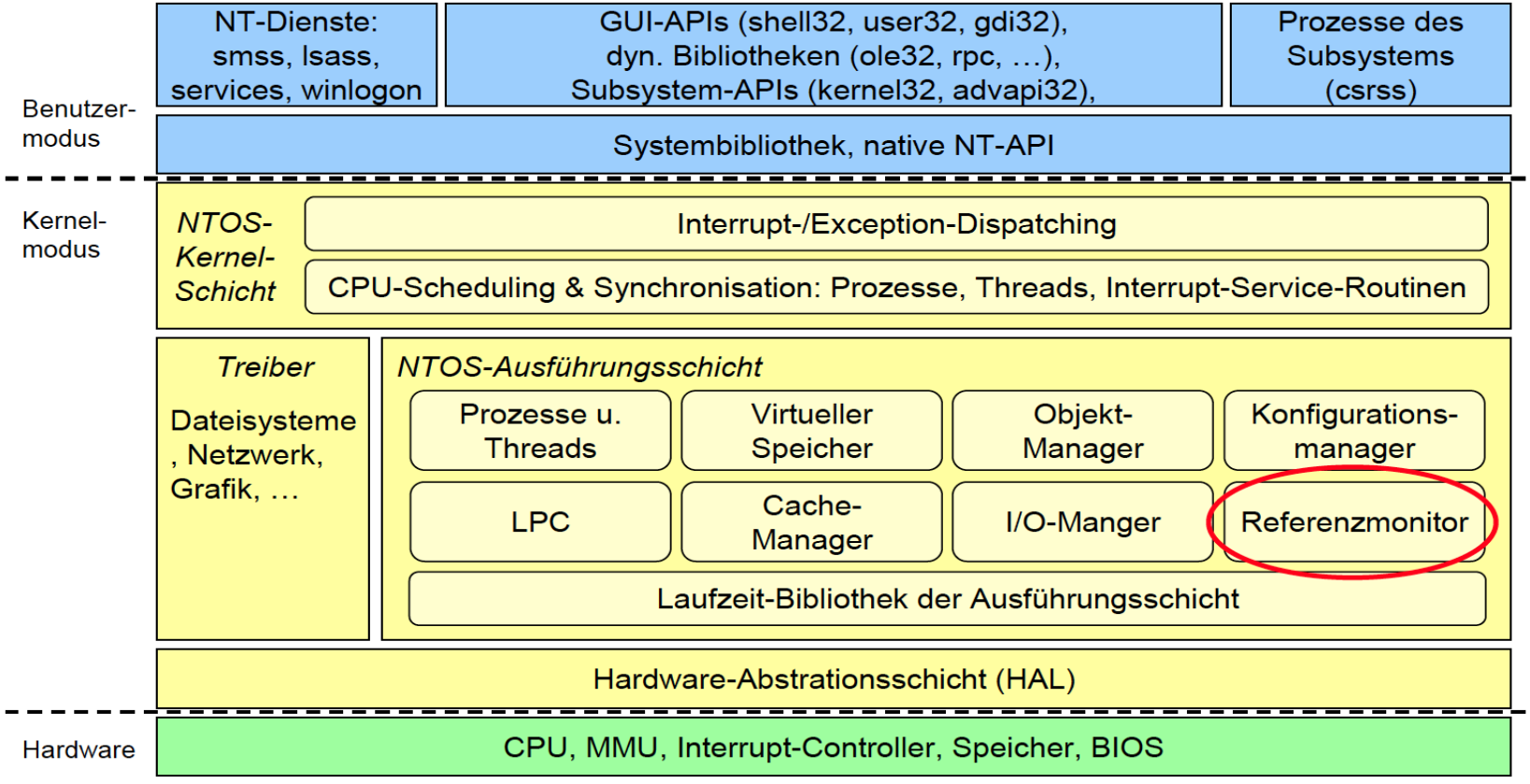


Authorization in Windows

- security-descriptor contains
  - SID = system wide unique security ID of owner and group of object
  - DACL = discretionary ACL: list of ACEs
    - ACE = access control element with allow/deny
  - SACL = system ACL, specifies operations to log and audit



Authorization in Windows



Privilege Restriction on Windows

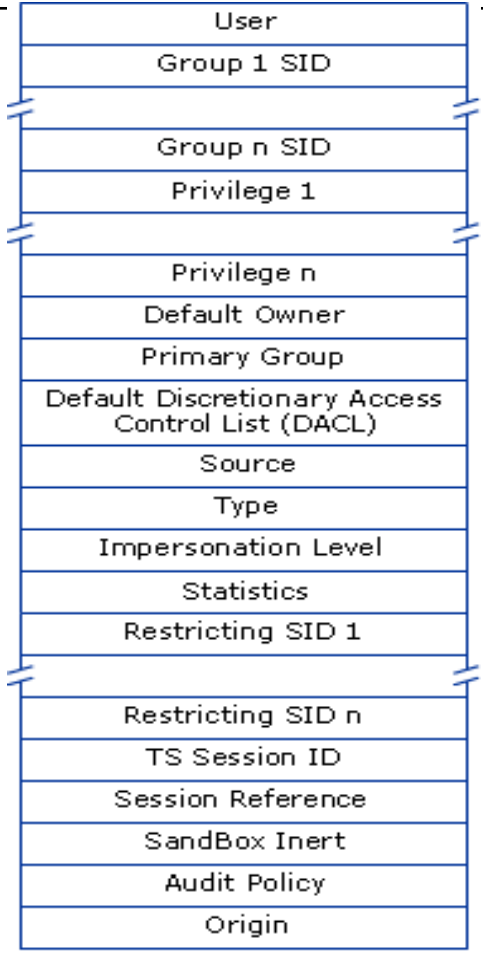
The security context of a process or thread on Windows is described by an access token (AT)

- created by the system after user login
- every process started in the users context has a copy of the AT
- used by the system to identify the user when a thread interacts with a securable object or tries to perform a system task that requires privileges

Privilege Restriction on Windows

access tokens contain the following information:

- The security identifier (SID) for the user's account
- SIDs for the groups of which the user is a member
- logon SID that identifies the current logon session
- list of the privileges held by either the user or the user's group
- owner SID
- TSID for the primary group
- default DACL that the system uses when the user creates a securable object without specifying a security descriptor
- source of the access token
- Whether the token is a primary or impersonation token
- optional list of restricting SIDs
- Current impersonation levels
- Other statistics



Privilege Restriction on Windows

selected privileges:

- SeTakeOwnershipPrivilege (Take ownership of files or other objects)
- SeSystemTimePrivilege (Change the system time)
- SeCreatePagefilePrivilege (Create a pagefile)
- SeDebugPrivilege (Debug programs)
- SeLoadDriverPrivilege (Load and unload device drivers)
- SeLockMemoryPrivilege (Lock pages in memory)
- SeShutdownPrivilege (Shut down the system)

For full list of rights and privileges:

http://technet.microsoft.com/en-us/library/dd277311.aspx

Privilege Restriction on Windows

- access tokens contain the sum of the rights which are granted
- single operations should not have all the users right (e.g. should a browser be able to change the users password?)
- solution?

- restrict rights of processes to needed rights
  - call process through CreateProcessAsUser( )
  - use restricted tokens

Privilege Restriction on Windows

A restricted token is a

- primary or impersonation access token that
  has been modified by the CreateRestrictedToken function.

A process or impersonating thread running in the security context of a restricted token is restricted in its ability to access securable objects or perform privileged operations.

The CreateRestrictedToken function can restrict a tokens in the following ways

- Remove privileges from the token.
- Apply the deny-only attribute to SIDs in the token so that they cannot be used to access secured objects. For more information about the deny-only attribute, see SID Attributes in an Access Token.
- Specify a list of restricting SIDs, which can limit access to securable objects.

Privilege Restriction on Windows

C++ signature:

```
BOOL CreateRestrictedToken(
  HANDLE ExistingTokenHandle,
  DWORD Flags,
  DWORD DisableSidCount,
  PSID_AND_ATTRIBUTES SidsToDisable,
  DWORD DeletePrivilegeCount,
  PLUID_AND_ATTRIBUTES
PrivilegesToDelete,
  DWORD RestrictedSidCount,
  PSID_AND_ATTRIBUTES SidsToRestrict,
  PHANDLE NewTokenHandle
);
```

- ExistingTokenHandle

  Handle to an existing token. An existing token handle can be obtained via a call to either OpenProcessToken( ) or OpenThreadToken( ). The token may be either a primary or a restricted token. In the latter case, the token may be obtained from an earlier call to CreateRestrictedToken( ). The existing token handle must have been opened or created with TOKEN_DUPLICATE access.

- Flags

  May be specified as 0 or as a combination of DISABLE_MAX_PRIVILEGE or SANDBOX_INERT. If DISABLE_MAX_PRIVILEGE is used, all privileges in the new token are disabled, and the two arguments DeletePrivilegeCount and PrivilegesToDelete are ignored. The SANDBOX_INERT has no special meaning other than it is stored in the token, and can be later queried using GetTokenInformation( ).

- DisableSidCount

  Number of elements in the list SidsToDisable. May be specified as 0 if there are no SIDs to be disabled. Disabling a SID is the same as enabling the SIDs "deny" attribute.

Privilege Restriction on Windows

C++ signature:

```
BOOL CreateRestrictedToken(
  HANDLE ExistingTokenHandle,
  DWORD Flags,
  DWORD DisableSidCount,
  PSID_AND_ATTRIBUTES SidsToDisable,
  DWORD DeletePrivilegeCount,
  PLUID_AND_ATTRIBUTES
PrivilegesToDelete,
  DWORD RestrictedSidCount,
  PSID_AND_ATTRIBUTES SidsToRestrict,
  PHANDLE NewTokenHandle
);
```

- SidsToDisable
  List of SIDs for which the "deny" attribute is to be enabled. May be specified as NULL if no SIDs are to have the "deny" attribute enabled. See below for information on the SID_AND_ATTRIBUTES structure.
- DeletePrivilegeCount
  Number of elements in the list PrivilegesToDelete. May be specified as 0 if there are no privileges to be deleted.
- PrivilegesToDelete
  List of privileges to be deleted from the token. May be specified as NULL if no privileges are to be deleted. See below for information on the LUID_AND_ATTRIBUTES structure.
- RestrictedSidCount
  Number of elements in the list SidsToRestrict. May be specified as 0 if there are no restricted SIDs to be added.
- SidsToRestrict
  List of SIDs to restrict. If the existing token is a restricted token that already has restricted SIDs, the resulting token will have a list of restricted SIDs that is the intersection of the existing token's list and this list. May be specified as NULL if no restricted SIDs are to be added to the new token.
- NewTokenHandle
  Pointer to a HANDLE that will receive the handle to the newly created token.

Privilege Restriction on Windows

So how to get the `ExistingTokenHandle`?

Solution:

```
BOOL OpenProcessToken(HANDLE hProcess,
            DWORD dwDesiredAccess,
            PHANDLE phToken);

BOOL OpenThreadToken(HANDLE hThread,
            DWORD dwDesiredAccess,
            BOOL bOpenAsSelf,
            PHANDLE phToken);
```

- hProcess
  Handle to the current process, which is normally obtained via a call to GetCurrentProcess( ).
- hThread
  Handle to the current thread, which is normally obtained via a call to GetCurrentThread( ).
- dwDesiredAccess
  Bit mask of the types of access desired for the returned token handle.
- bOpenAsSelf
  Boolean flag that determines how the access check for retrieving the thread's token is performed. If specified as FALSE, the access check uses the calling thread's permissions. If specified as TRUE, the access check uses the calling process's permissions.
- phToken
  Pointer to a HANDLE that will receive the handle to the process's primary token or the thread's impersonation token, depending on whether you're calling OpenProcessToken( ) or OpenThreadToken( ).

Privilege Restriction on Windows

Example Pseudo-Code

```
HANDLE hProcessToken, hRestrictedToken;

/* First get a handle to the current process's primary token */

OpenProcessToken(GetCurrentProcess(  ), TOKEN_DUPLICATE | TOKEN_ASSIGN_PRIMARY,
        &hProcessToken);

/* Create a restricted token with all privileges removed */

CreateRestrictedToken(hProcessToken, DISABLE_MAX_PRIVILEGE, 0, 0, 0, 0, 0, 0,
        &hRestrictedToken);

/* Create a new process using the restricted token */

CreateProcessAsUser(hRestrictedToken, ...);

/* Cleanup */

CloseHandle(hRestrictedToken);
CloseHandle(hProcessToken);
```

Privilege Restriction on Windows

Creating new processes with restricted rights works now, but how to adjust rights of the current process?

Solution:

- modify process's primary token with

```
BOOL AdjustTokenPrivileges(HANDLE TokenHandle,
            BOOL DisableAllPrivileges,
            PTOKEN_PRIVILEGES NewState,
            DWORD BufferLength,
            PTOKEN_PRIVILEGES PreviousState,
            PDWORD ReturnLength);
```

skipped the parameters listed here

- TokenHandle
  Handle to the token that is to have its privileges adjusted. The handle must have been opened with TOKEN_ADJUST_PRIVILEGES access; in addition, if PreviousState is to be filled in, it must have TOKEN_QUERY access.

- DisableAllPrivileges
  Boolean argument that specifies whether all privileges held by the token are to be disabled. If specified as TRUE, all privileges are disabled, and the NewState argument is ignored. If specified as FALSE, privileges are adjusted according to the information in the NewState argument.

Privilege Restriction on Windows

```
BOOL AdjustTokenPrivileges(HANDLE TokenHandle,
            BOOL DisableAllPrivileges,
            PTOKEN_PRIVILEGES NewState,
            DWORD BufferLength,
            PTOKEN_PRIVILEGES PreviousState,
            PDWORD ReturnLength);
```

- NewState
  List of privileges that are to be adjusted, along with the adjustment that is to be made for each. Privileges can be enabled, disabled, and removed. The TOKEN_PRIVILEGES structure contains two fields: PrivilegeCount and Privileges. PrivilegeCount is simply a DWORD that indicates how many elements are in the array that is the Privileges field. The Privileges field is an array of LUID_AND_ATTRIBUTES structures, for which the Attributes field of each element indicates how the privilege is to be adjusted. A value of 0 disables the privilege, SE_PRIVILEGE_ENABLED enables it, and SE_PRIVILEGE_REMOVED removes the privilege.

- BufferLength
  Length in bytes of the PreviousState buffer. May be 0 if PreviousState is NULL.

- PreviousState
  Buffer into which the state of the token's privileges prior to adjustment is stored. It may be specified as NULL if the information is not required. If the buffer is not specified as NULL, the token must have been opened with TOKEN_QUERY access.

- ReturnLength
  Pointer to an integer into which the number of bytes written into the PreviousState buffer will be placed. May be specified as NULL if PreviousState is also NULL.

Privilege Restriction on Windows

needed in case you want to manipulate

potentially uncommon data-structures:

- SID_AND_ATTRIBUTES
  - two fields: SID (type PSID) and Attributes
  - SID should never be directly manipulated
  - Attribute is ignored if structure is used for disabling SID
  - Atttribute must be 0 for restricting SID
  - *Note: anyway 0 is the secure choice for Attribute here*

- terrible to get into this topic very complex and unintuitive

- LUID_AND_ATTRIBUTES
  - two fields: LUID (type LUID) and Attributes
  - LUID should never be directly manipulated
  - Attribute is ignored if structure is used for deleting privileges
  - Attribute must be SE_PRIVILEGE_ENABLED to enable the privilege, SE_PRIVILEGE_REMOVED to remove the privilege, or 0 to disable the privilege
  - *Note: SE_PRIVILEGE_REMOVED attribute is not valid on Win NT, Win 2K or Win XP*

Privilege Restriction on Windows

code example as starting point for Windows stuff

```
#include <windows.h>

BOOL RemoveBackupAndRestorePrivileges(VOID) {
    BOOL              bResult;
    HANDLE            hProcess, hProcessToken;
    PTOKEN_PRIVILEGES pNewState;

    /* Allocate a TOKEN_PRIVILEGES buffer to hold the privilege change information.
     * Two privileges will be adjusted, so make sure there is room for two
     * LUID_AND_ATTRIBUTES elements in the Privileges field of TOKEN_PRIVILEGES.
     */
    pNewState = (PTOKEN_PRIVILEGES)LocalAlloc(LMEM_FIXED, sizeof(TOKEN_PRIVILEGES) +
                                              (sizeof(LUID_AND_ATTRIBUTES) * 2));
    if (!pNewState) return FALSE;

    /* Add the two privileges that will be removed to the allocated buffer */
    pNewState->PrivilegeCount = 2;
    if (!LookupPrivilegeValue(0, SE_BACKUP_NAME, &pNewState->Privileges[0].Luid) ||
        !LookupPrivilegeValue(0, SE_RESTORE_NAME, &pNewState->Privileges[1].Luid)) {
        LocalFree(pNewState);
        return FALSE;
    }
    pNewState->Privileges[0].Attributes = SE_PRIVILEGE_REMOVED;
    pNewState->Privileges[1].Attributes = SE_PRIVILEGE_REMOVED;

    /* Get a handle to the process's primary token.  Request TOKEN_ADJUST_PRIVILEGES
     * access so that we can adjust the privileges.  No other privileges are req'd
     * since we'll be removing the privileges and thus do not care about the previous
     * state.  TOKEN_QUERY access would be required in order to retrieve the previous
     * state information.
     */
    hProcess = GetCurrentProcess(  );
    if (!OpenProcessToken(hProcess, TOKEN_ADJUST_PRIVILEGES, &hProcessToken)) {
        LocalFree(pNewState);
        return FALSE;
    }
```

Privilege Restriction on Windows

```
    /* Adjust the privileges, specifying FALSE for DisableAllPrivileges so that the
     * NewState argument will be used instead.  Don't request information regarding
     * the token's previous state by specifying 0 for the last three arguments.
     */
    bResult = AdjustTokenPrivileges(hProcessToken, FALSE, pNewState, 0, 0, 0);

    /* Cleanup and return the success or failure of the adjustment */
    CloseHandle(hProcessToken);
    LocalFree(pNewState);
    return bResult;
}
```

Note: also the modification of tokens requires privileges:

- SeCreateTokenPrivilege (Create a token object)
- SeAssignPrimaryTokenPrivilege (Replace a process-level token)

thanks for your interest

to be continued