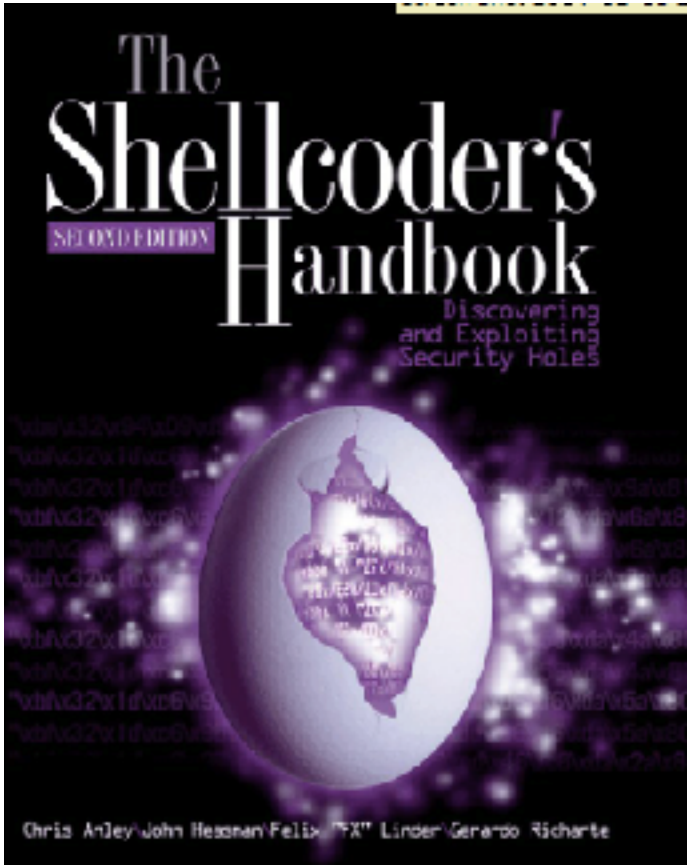
CNIT 127: Exploit Development
Ch 8: Windows Overflows
Part 2


The Shellcoder's Handbook
Second Edition
Discovering and Exploiting Security Holes
Chris Anley\John Heasman\Felix "FX" Linder\Gerardo Richarte

Topics

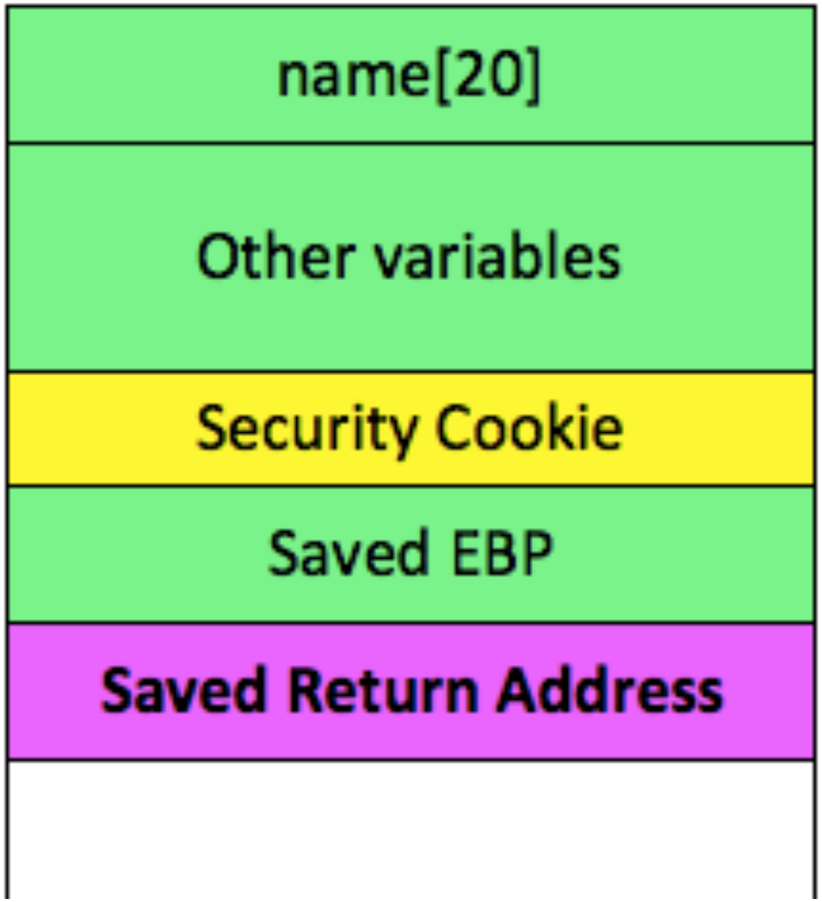
- Stack Protection
- Heap-Based Buffer Overflows
- Other Overflows

Stack Protection

Windows Stack Protections

- Microsoft Visual C++ .NET provides
  - /GS compiler flag is on by default
    - Tells compiler to place *security cookies* on the stack to guard the saved return address
    - Equivalent of a *canary*
  - 4-byte value (dword) placed on the stack after a procedure call
    - Checked before procedure return
    - Protects saved return address and EBP

Stack Protected by a Security Cookie



name[20]

Other variables

Security Cookie

Saved EBP

Saved Return Address

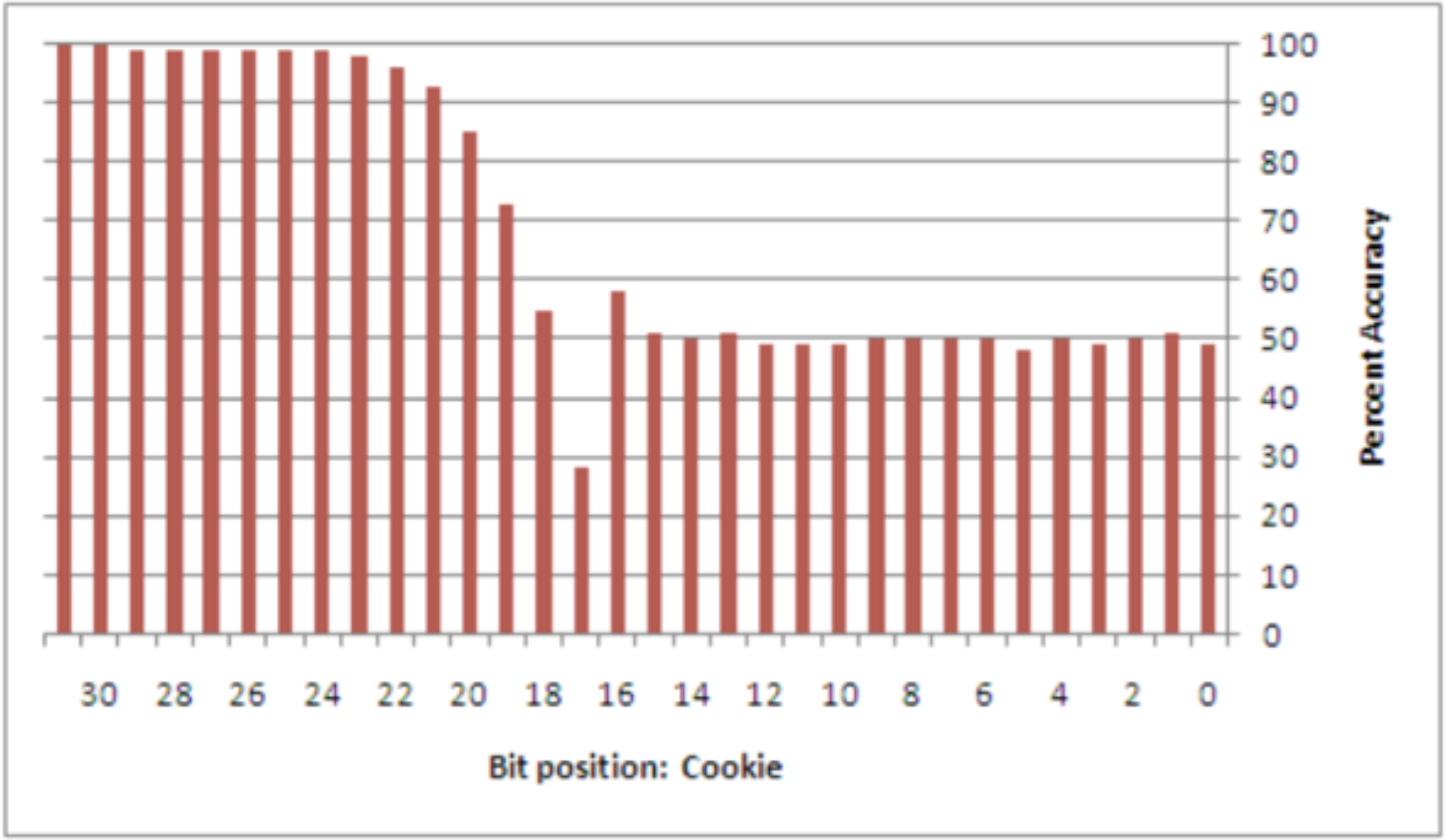
How is the Cookie Generated?

- When a process starts, Windows combines these values with XOR
  - DateTime (a 64-bit integer counting time intervals of 100 nanoseconds)
  - Process ID
  - Thread ID
  - TickCount (number of milliseconds since the system started up)
  - Performance Counter (number of CPU cycles)

Predicting the Cookie

- If an attacker can run a process on the target to get system time values
- Some bits of the cookie can be predicted

Effectively 17 bits of Randomness

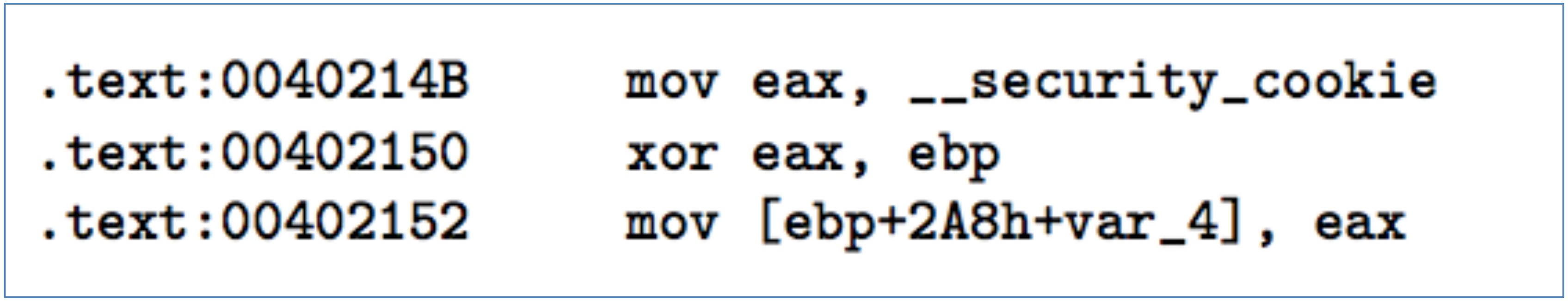


How Good is 17 Bits?

- 2^17 = 131,072
- So an attacker would have to run an attack 100,000 times or so to win by guessing the cookie

Prologue Modification

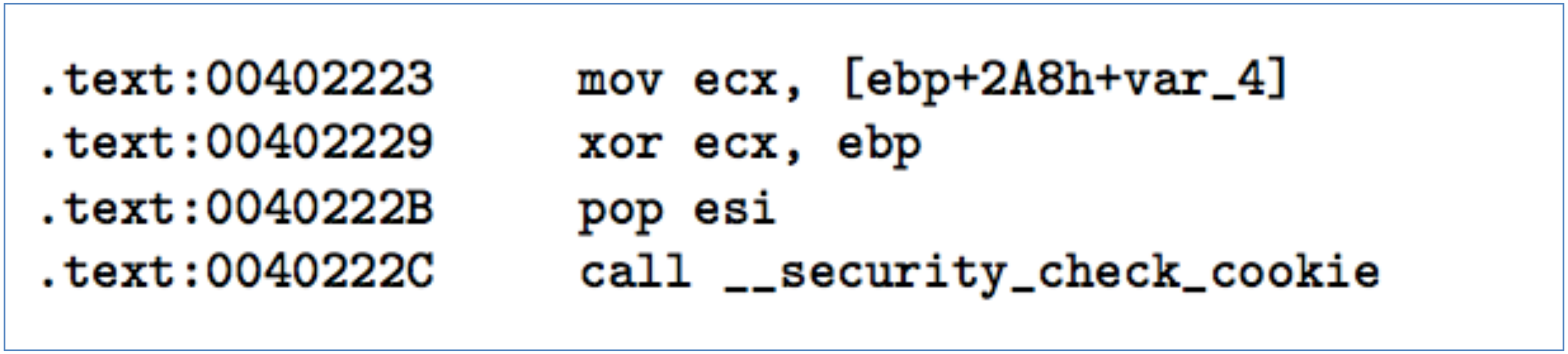
- ___security_cookie value placed in the stack at a carefully calculated position
- To protect the EBP and Return value
  - From link Ch 8m



```
.text:0040214B      mov eax, __security_cookie
.text:00402150      xor eax, ebp
.text:00402152      mov [ebp+2A8h+var_4], eax
```

Epilogue Modification

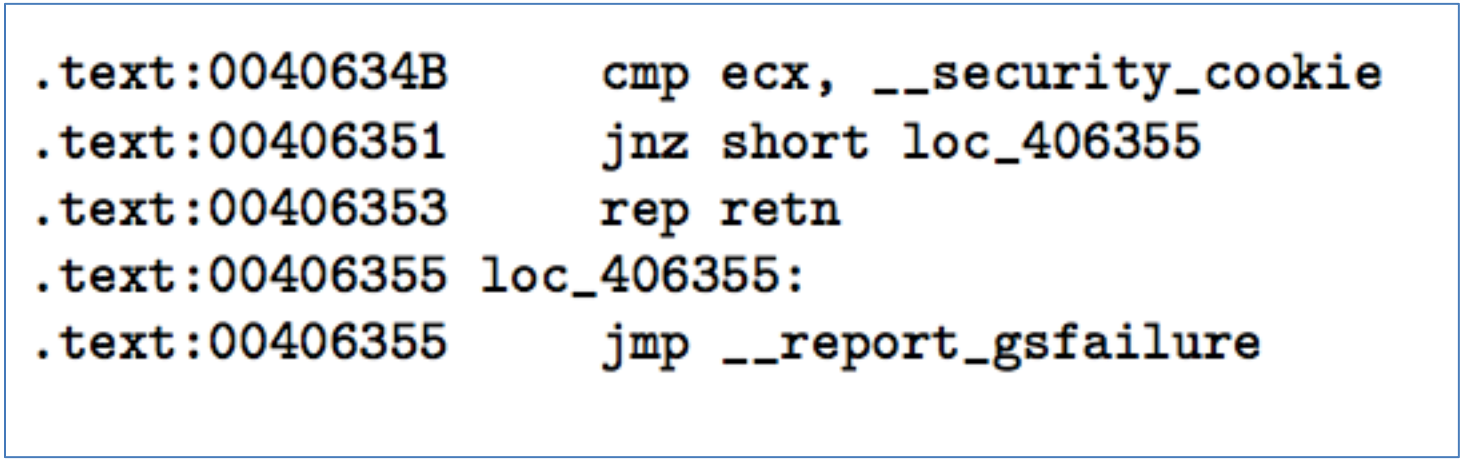
- Epilogue to a function now includes these instructions
  - From link Ch 8m



```
.text:00402223      mov ecx, [ebp+2A8h+var_4]
.text:00402229      xor ecx, ebp
.text:0040222B      pop esi
.text:0040222C      call __security_check_cookie
```

__security_check_cookie

- Current cookie value is in ecx
- Compared to authoritative value stored in the .data section of the image file of the procedure
- If the check fails, it calls a security handler, using a pointer stored in the .data section

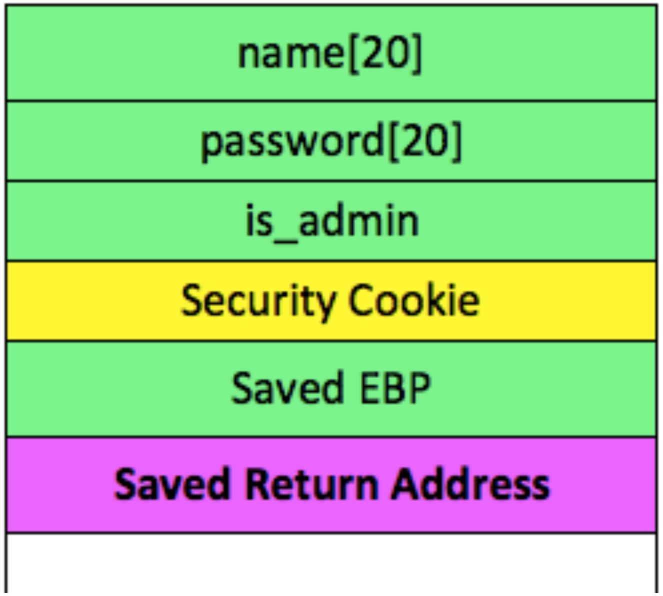


```
.text:0040634B     cmp ecx, __security_cookie
.text:00406351     jnz short loc_406355
.text:00406353     rep retn
.text:00406355 loc_406355:
.text:00406355     jmp __report_gsfailure
```

Parameter Order

- Before Windows Server 2003, local variables were placed on the stack in the order of their declaration in the C++ source code
- Now all arrays are moved to the bottom of the list, closest to the saved return address
- This prevents buffer overflows in the arrays from changing the non-array variables

Long password becomes admin



name[20]

password[20]

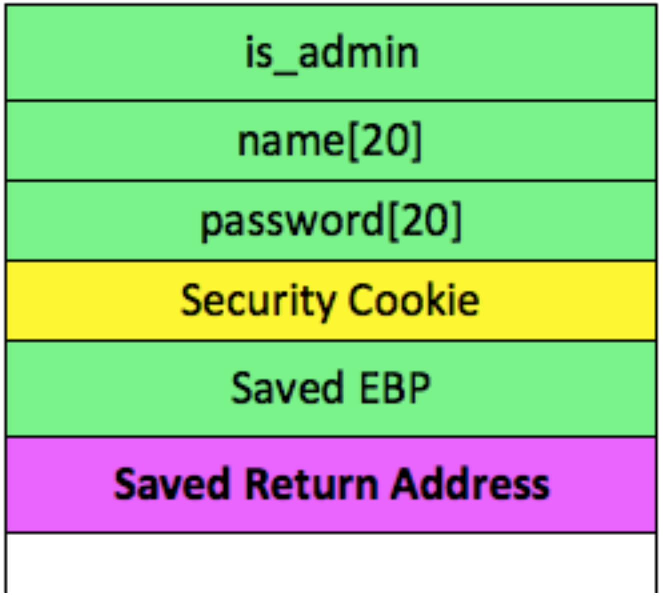
is_admin

Security Cookie

Saved EBP

Saved Return Address

Cannot overwrite is_admin



is_admin

name[20]

password[20]

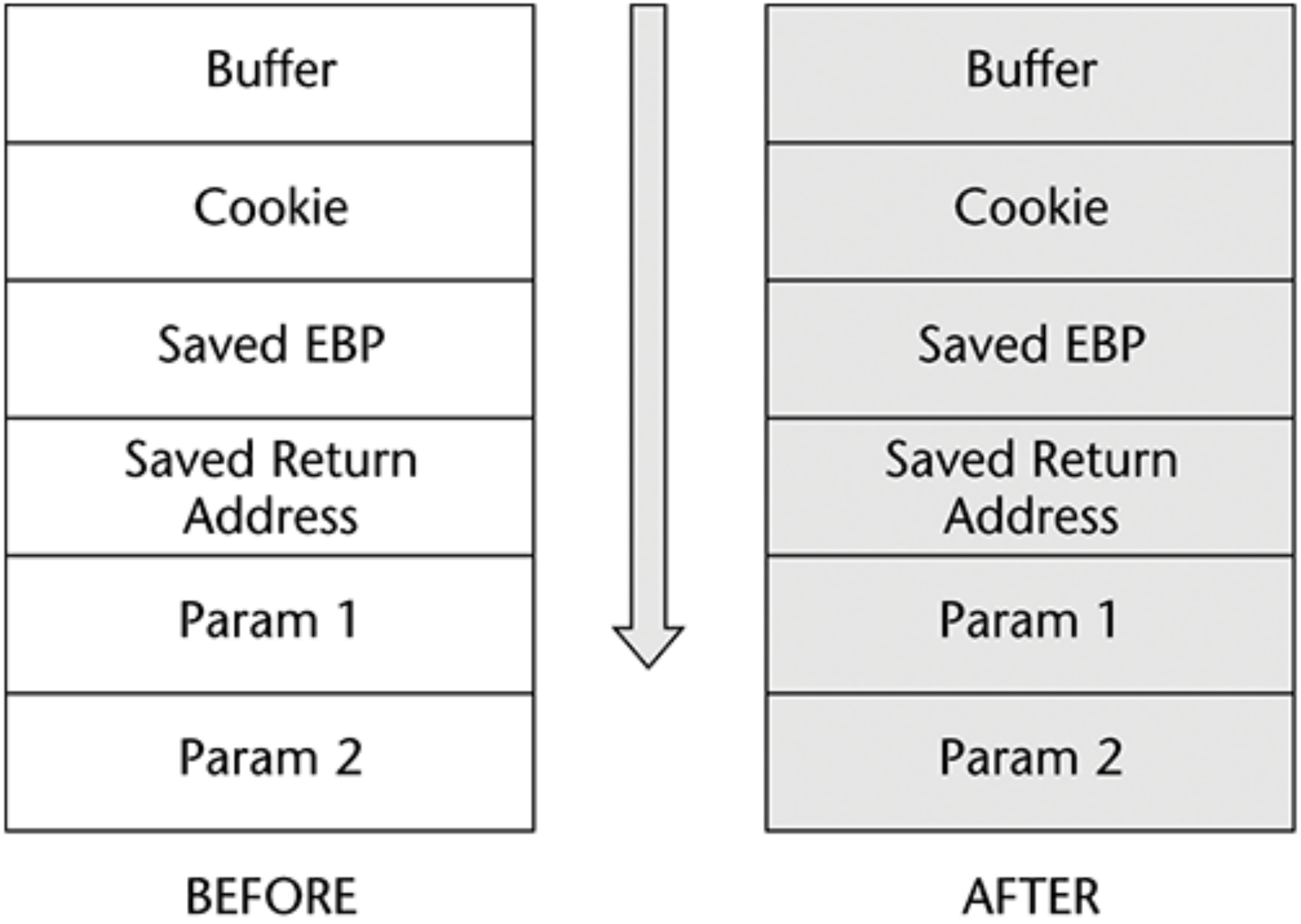
Security Cookie

Saved EBP

Saved Return Address

Overwriting Parameters

Figure 8-3: Before and after snapshots of the buffer



Overwriting Parameters

- We've changed the cookie, but if the parameters are used in a write operation before the function returns, we could
  - Overwrite the authoritative cookie value in the .data section, so the cookie check passes
  - Overwrite the handler pointer to the security handler, and let the cookie check fail
    - Handler could point to injected code
    - Or set handler to zero and overwrite the default exception handler value

Heap-Based Buffer Overflows

Purpose of the Heap

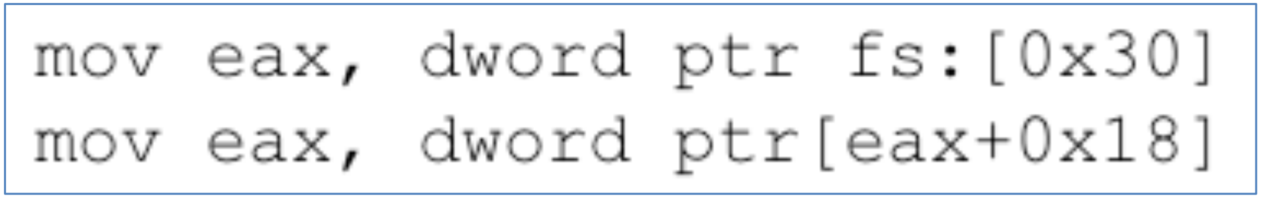
- Consider a Web server
- HTTP requests vary in length
- May vary from 20 to 20,000 bytes or longer (in principle)
- Once processed, the request can be discarded, freeing memory for re-use
- For efficiency, such data is best stored on the heap

The Process Heap

- Every process running on Win32 has a process heap
- The C function GetProcessHeap() returns a handle to the process heap
- A pointer to the process heap is also stored in the Process Environment Block

The Process Heap

- This code returns that pointer in eax

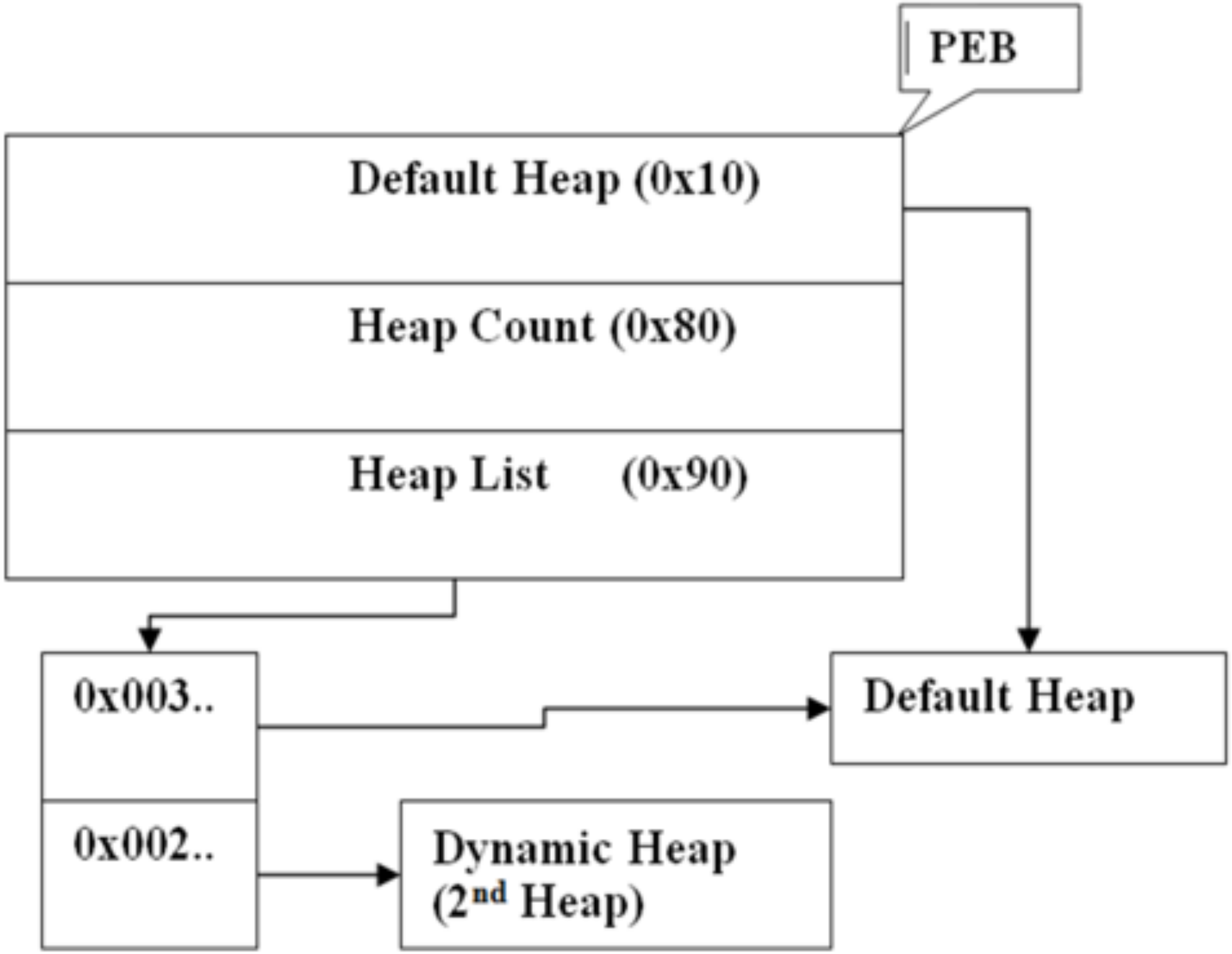


```
mov eax, dword ptr fs:[0x30]
mov eax, dword ptr[eax+0x18]
```

- Many of the underlying functions of the Windows API use this default process heap

Dynamic Heaps

- A process can create as many dynamic heaps as required
- All inside the default process heap
- Created with the HeapCreate() function



- From link Ch 8o

Working with the Heap

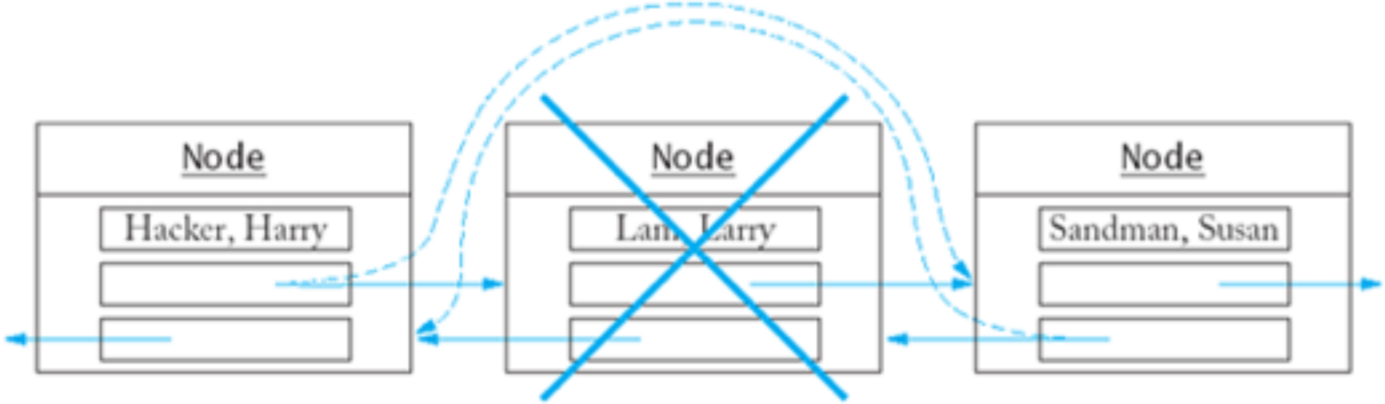
- Application uses HeapAllocate() to borrow a chunk of memory on the heap
  - Legacy functions left from Win16 are LocalAlloc() & GlobalAlloc(), but they do the same thing—there's no difference in Win32
- When the application is done with the memory, if calls HeapFree()
  - Or LocalFree() or GlobalFree()

How the Heap Works

- The stack grows downwards, towards address 0x00000000
- The heap grows upwards
- Heap starts with 128 LIST_ENTRY structures that keep track of free blocks

Vulnerable Heap Operations

- When a chunk is freed, forward and backward pointers must be updated
- This enables us to control a write operation, to write to arbitrary RAM locations
  - Image from mathyvanhoef.com, link Ch 5b



Details

- There is a lot more to it, involving these structures
  - Segment list
  - Virtual Allocation list
  - Free list
  - Lookaside list
- For details, see link Ch8o

Exploiting Heap-Based Overflows: Three Techniques

- Overwrite the pointer to the exception handler
- Overwrite the pointer to the Unhandled Exception Filter
- Overwrite a pointer in the PEB

Overwrite a Pointer in the PEB

- RtlEnterCriticalSection, called by RtlAcquirePebLock() and RtlReleasePebLock()
- Called whenever a process exits with ExitProcess()
- PEB location is fixed for all versions of Win NT
- Your code should restore this pointer, and you may also need to repair the heap

Win 2003 Server

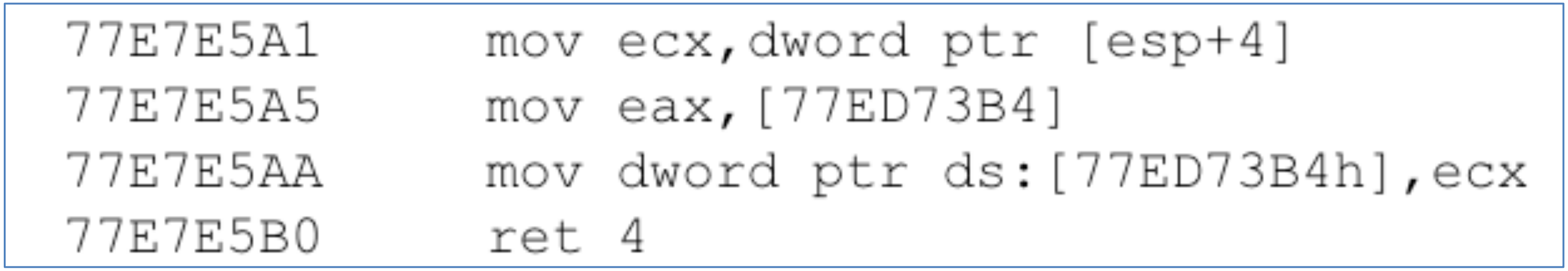
- Does not use these pointers in the PEB
- But there are Ldr* functions that call pointers we can control
  - Including LdrUnloadDll()

Vectored Exception Handling

- Introduced with Windows XP
- Traditional frame-based exception handling stores exception registration records on the stack
- Vectored exception handling stores information about handlers on the heap
- A heap overflow can change them

Overwrite a Pointer to the Unhandled Exception Filter

- First proposed at Blackhat Amsterdam (2001)
- An application can set this value using SetUnhandledExceptionFilter()
  - Disassemble that function to find the pointer



```
77E7E5A1        mov ecx,dword ptr [esp+4]
77E7E5A5        mov eax,[77ED73B4]
77E7E5AA        mov dword ptr ds:[77ED73B4h],ecx
77E7E5B0        ret 4
```

Repairing the Heap

- The overflow corrupts the heap
- Shellcode will probably cause an access violation
- Simplest repair process is to just make the heap look like a fresh, empty heap
  - With the one block we are using on it

Restore the Exception Handler you Abused

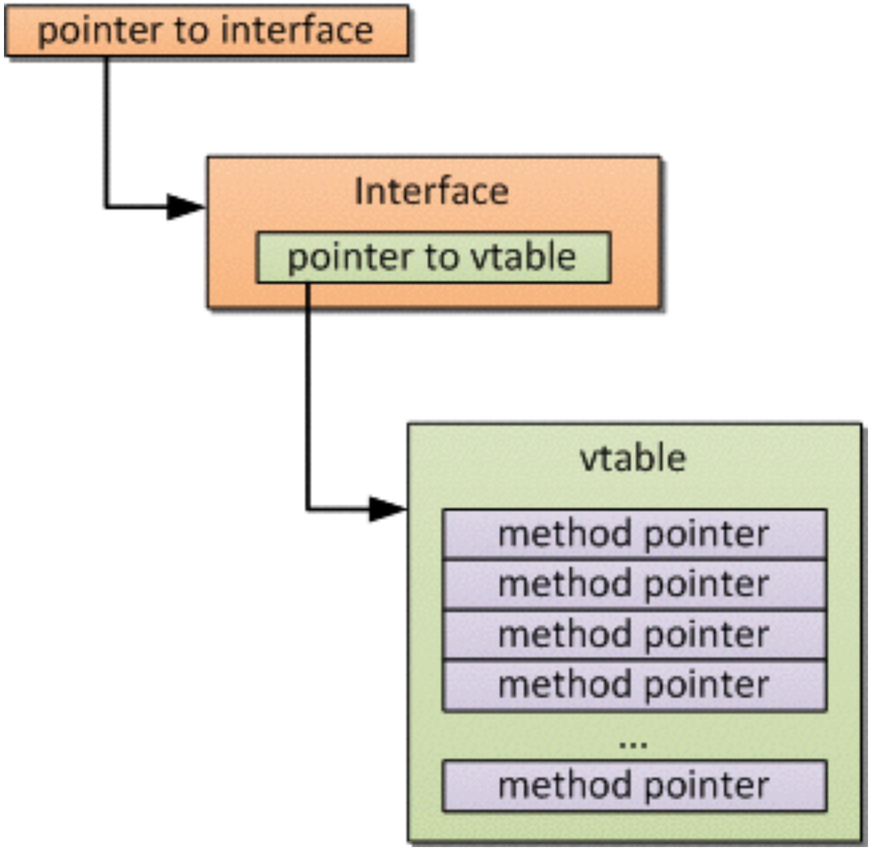
- Otherwise, you could create an endless loop
- If your shellcode causes an exception

COM Objects and the Heap

- Component Object Model (COM) Objects
  - An object that can be created when needed by another program
  - It has *methods* that can be called to perform a task
  - It also has *attributes* (stored data)
- COM objects are created on the heap

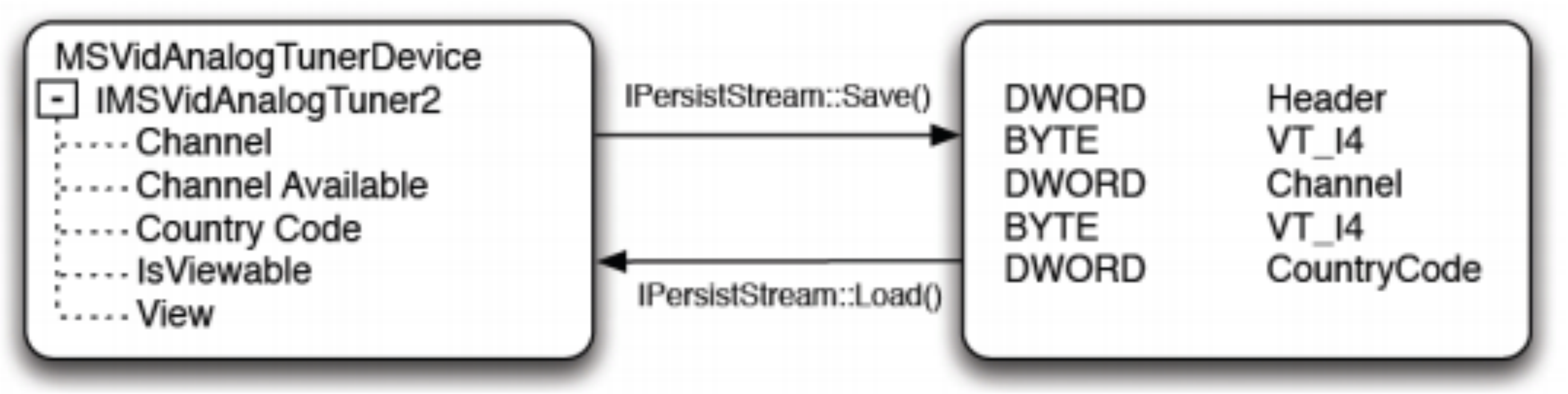
Vtable in Heap

- All COM classes have one or more interfaces, which are used to connect them to a program
  - Figure from link Ch 8p

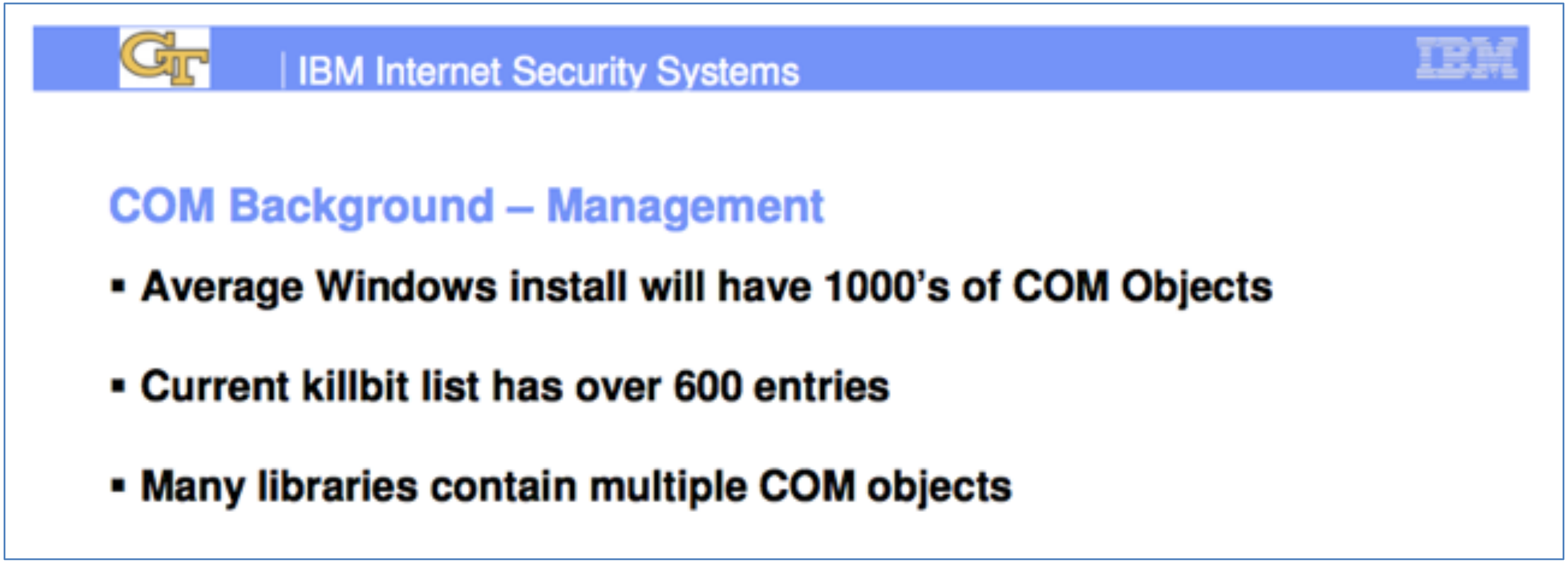


COM Objects Contain Data

- If the programmer doesn't check, these data fields could be overflowed, into the next object's *vtable*
  - Image from link Ch 8q



DWORD	Header
BYTE	VT_I4
DWORD	Channel
BYTE	VT_I4
DWORD	CountryCode

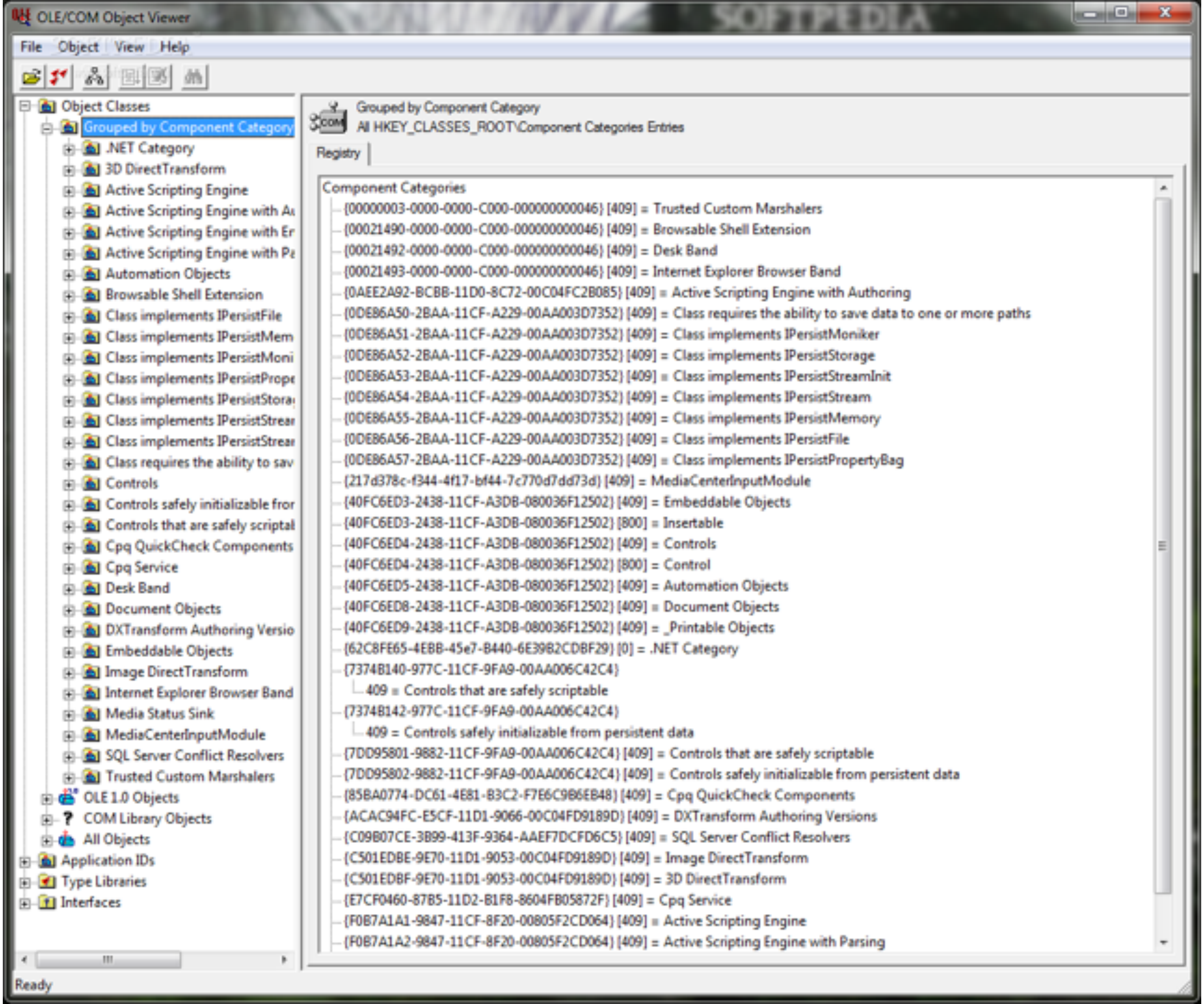


GT
IBM Internet Security Systems
IBM

COM Background – Management

- Average Windows install will have 1000's of COM Objects
- Current killbit list has over 600 entries
- Many libraries contain multiple COM objects

- Vunerable COM objects are often not fixed
  - Just added to the "killbit" list
  - Which can be circumvented
    - From link Ch 8qq; Image on next slide from link Ch 8r



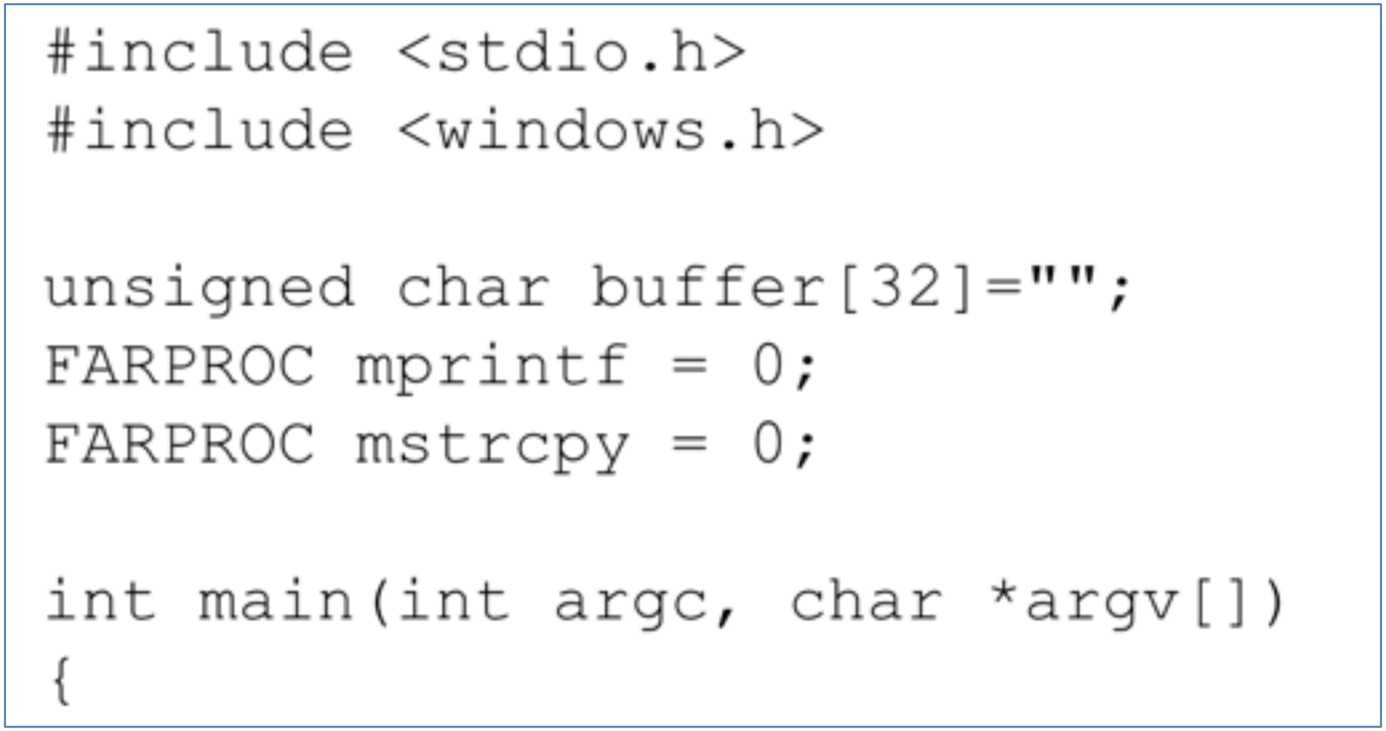
```
Object Classes
  Grouped by Component Category
    .NET Category
    3D DirectTransform
    Active Scripting Engine
    Active Scripting Engine with Au
    Active Scripting Engine with Er
    Active Scripting Engine with Pa
    Automation Objects
    Browsable Shell Extension
    Class implements IPersistFile
    Class implements IPersistMem
    Class implements IPersistMoni
    Class implements IPersistPrope
    Class implements IPersistStora
    Class implements IPersistStrear
    Class implements IPersistStrear
    Class requires the ability to sav
    Controls
    Controls safely initializable fror
    Controls that are safely scriptal
    Cpq QuickCheck Components
    Cpq Service
    Desk Band
    Document Objects
    DXTransform Authoring Versio
    Embeddable Objects
    Image DirectTransform
    Internet Explorer Browser Band
    Media Status Sink
    MediaCenterInputModule
    SQL Server Conflict Resolvers
    Trusted Custom Marshalers
  OLE 1.0 Objects
  COM Library Objects
  All Objects
Application IDs
Type Libraries
Interfaces
```

Grouped by Component Category
All HKEY_CLASSES_ROOT\Component Categories Entries

```
Component Categories
  {00000003-0000-0000-C000-000000000046} [409] = Trusted Custom Marshalers
  {00021490-0000-0000-C000-000000000046} [409] = Browsable Shell Extension
  {00021492-0000-0000-C000-000000000046} [409] = Desk Band
  {00021493-0000-0000-C000-000000000046} [409] = Internet Explorer Browser Band
  {0AEE2A92-BCBB-11D0-8C72-00C04FC2B085} [409] = Active Scripting Engine with Authoring
  {0DE86A50-2BAA-11CF-A229-00AA003D7352} [409] = Class requires the ability to save data to one or more paths
  {0DE86A51-2BAA-11CF-A229-00AA003D7352} [409] = Class implements IPersistMoniker
  {0DE86A52-2BAA-11CF-A229-00AA003D7352} [409] = Class implements IPersistStorage
  {0DE86A53-2BAA-11CF-A229-00AA003D7352} [409] = Class implements IPersistStreamInit
  {0DE86A54-2BAA-11CF-A229-00AA003D7352} [409] = Class implements IPersistStream
  {0DE86A55-2BAA-11CF-A229-00AA003D7352} [409] = Class implements IPersistMemory
  {0DE86A56-2BAA-11CF-A229-00AA003D7352} [409] = Class implements IPersistFile
  {0DE86A57-2BAA-11CF-A229-00AA003D7352} [409] = Class implements IPersistPropertyBag
  {217d378c-f344-4f17-bf44-7c770d7dd73d} [409] = MediaCenterInputModule
  {40FC6ED3-2438-11CF-A3DB-080036F12502} [409] = Embeddable Objects
  {40FC6ED3-2438-11CF-A3DB-080036F12502} [800] = Insertable
  {40FC6ED4-2438-11CF-A3DB-080036F12502} [409] = Controls
  {40FC6ED4-2438-11CF-A3DB-080036F12502} [800] = Control
  {40FC6ED5-2438-11CF-A3DB-080036F12502} [409] = Automation Objects
  {40FC6ED8-2438-11CF-A3DB-080036F12502} [409] = Document Objects
  {40FC6ED9-2438-11CF-A3DB-080036F12502} [409] = _Printable Objects
  {62C8FE65-4EBB-45e7-B440-6E39B2CDBF29} [0] = .NET Category
  {7374B140-977C-11CF-9FA9-00AA006C42C4}
    409 = Controls that are safely scriptable
  {7374B142-977C-11CF-9FA9-00AA006C42C4}
    409 = Controls safely initializable from persistent data
  {7DD95801-9882-11CF-9FA9-00AA006C42C4} [409] = Controls that are safely scriptable
  {7DD95802-9882-11CF-9FA9-00AA006C42C4} [409] = Controls safely initializable from persistent data
  {85BA0774-DC61-4E81-B3C2-F7E6C9B6EB48} [409] = Cpq QuickCheck Components
  {ACAC94FC-E5CF-11D1-9066-00C04FD9189D} [409] = DXTransform Authoring Versions
  {C09B07CE-3B99-413F-9364-AAEF7DCFD6C5} [409] = SQL Server Conflict Resolvers
  {C501EDBE-9E70-11D1-9053-00C04FD9189D} [409] = Image DirectTransform
  {C501EDBF-9E70-11D1-9053-00C04FD9189D} [409] = 3D DirectTransform
  {E7CF0460-87B5-11D2-B1F8-8604FB05872F} [409] = Cpq Service
  {F0B7A1A1-9847-11CF-8F20-00805F2CD064} [409] = Active Scripting Engine
  {F0B7A1A2-9847-11CF-8F20-00805F2CD064} [409] = Active Scripting Engine with Parsing
```

Other Overflows

Overflows in the .data Section



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

unsigned char buffer[32]="";
FARPROC mprintf = 0;
FARPROC mstrcpy = 0;

int main(int argc, char *argv[])
{
```

- If a buffer is placed before function pointers in the .data section
- Overflowing the buffer can change the pointers

TEB/PEB Overflows

- In principle, buffers in the TEB used for converting ASCII to Unicode could be overflowed
    - Changing pointers
- There are no public examples of this type of exploit