

Macro-Reliability In Win32 Exploits

Introduction

Nowadays, Windows platforms are widely localized. While Windows NT 4.0 only shipped in a few languages, Windows 2000 and XP have largely increased that number, one that is still increasing with Windows Vista.

Thus raises the issue of large scale exploitation: exploits targeting English Windows only are not sufficient anymore and methods have to be found to adapt to this new context. If you consider an embedded device such as Immunity's SILICA, with only 3 buttons to control exploitation, these methods should require as few user interaction as possible. Targets are heterogeneous, and the exploit has to adapt itself to give the best chances of exploitation.

Ideally, in order to have the best chances for a vulnerability to be exploited, one should gather the following information about the targeted Windows box:

- Major/Minor versions
- Service Packs
- Patches
- Configurations
- Language Packs
- Software version and configuration
- Networking conditions between the attacker and target
- Host protection mechanisms installed on the target

In rare cases, such as kernel exploits, even the number of CPU's on the target machine can affect the exploit's behavior.

In reality, all these are not that easy to get, and the attacker often has to satisfy themselves with a small subset. In this paper, we will discuss various new techniques for obtaining the high levels of reliability desired when attacking large numbers of hosts. This involves new techniques to gather more information from the remote targets as well as methods to harden exploits against target variation.

Common Return Addresses

One of the ideas that lead our reflection on the subject was to reduce the fingerprinting as much as possible, and to go only for reliable information. Indeed fingerprinting is usually noisy on the wire and we would rather avoid this. Moreover, if you look at the public fingerprinting techniques, such as the Service Pack fingerprinting through the MSRPC

interface enumeration, they are not that reliable across a wide range of heterogeneous hosts. For example, it is quite difficult to tell Windows 2000 SP3 from Windows 2000 SP4 from just MSRPC endpoints. There is currently no public information regarding remote fingerprinting of the localization of the Windows box.

The attacker has a long list of information they desire to get the best exploitation conditions. However, they can only count on the most basic information being available to the remote anonymous user: the major version of Windows (meaning NT 4.0, 2000 or XP). This can be done thanks to widely known techniques (like SMB negotiation for example) with an acceptably high success rate.

Some work has already been done on Service Pack independent return addresses — meaning that you can target a major version of Windows without having to know its actual service pack by using an address that is common to all service packs of that Windows version. But even with that, most "universal" exploits we can find in the wild will only target English install of Windows. Real "universal" exploits are rare.

Let's dig a bit more into this.

The Naïve Approach

The 1st thought one can have regarding this matter could be: would it be possible to try and find addresses as much independent as possible of the targets? Let's say look at the various binary components usually involved in running an executable:

- The DLLs: their image base address usually changes with language pack
- The EXEs: their image base doesn't change much, which is a good point
- The EXEs and DLLs: a different version number usually means different offsets relatively to image base. But well, versionning is definitely not a reliable science!

We can usually get cross-SP return addresses in DLLs that haven't changed, and cross-language return addresses in EXEs, but usually not both.

A naïve idea is that EXEs and DLLs with same version and same image base might provide common return addresses. The following small C program was written to try to corroborate this idea:

dllvers.c

```
/*
DLLVers v0.01 (c) 2005 Kostya Kortchinsky <kostya[at]immunityinc[dot]com>

Lists all the DLL and EXE in %WINDIR%\system32 and prints their version number and ImageBase. Quite useful to quickly track down unchanged binaries through service packs and localizations.

*/

#define _WIN32_WINNT 0x0501

#include <windows.h>
#include <string.h>
#include <string.h>
#include <dbghelp.h>

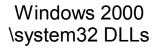
#pragma comment(lib, "version")
#pragma comment(lib, "dbghelp")

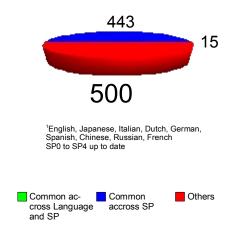
DWORD GetBaseAddress(char *FileName)
```

```
HMODULE hModule;
  HANDLE hFile, hFileMapping;
  PIMAGE NT HEADERS pNtHeader;
  DWORD dwBaseAddress;
  if(!(hFile = CreateFileA(FileName, GENERIC READ, 1, NULL, OPEN EXISTING, 0, (HANDLE)NULL))) {
   return (-1):
  if(!(hFileMapping = CreateFileMappingA(hFile, NULL, PAGE READONLY | SEC COMMIT, 0, 0, NULL))) {
    CloseHandle(hFile);
    return (-1);
  CloseHandle(hFile);
  if(!(hModule = (HMODULE)MapViewOfFile(hFileMapping, FILE MAP READ, 0, 0, 0))) {
    CloseHandle(hFileMapping);
    return (-1);
  CloseHandle(hFileMapping);
  pNtHeader = ImageNtHeader((PVOID)hModule);
#ifdef DEBUG
 printf("%d\n", pNtHeader->FileHeader.SizeOfOptionalHeader);
#endif
  dwBaseAddress = pNtHeader->OptionalHeader.ImageBase;
  UnmapViewOfFile((PVOID)hModule);
  return dwBaseAddress;
void PrintVersion(char *FileName)
  DWORD dwLen, dwHandle, dwBaseAddress;
  LPTSTR lpData, lpBuffer;
  UINT BufLen;
  VS FIXEDFILEINFO *pFileInfo;
  char *p = FileName + strlen(FileName);
  while (*p != '\\' && p >= FileName)
   p--;
  p++;
  p = _strlwr(p);
  if (strstr(FileName, ".dll") == NULL && strstr(FileName, ".exe") == NULL)
   return;
  if ((dwLen = GetFileVersionInfoSize(FileName, &dwHandle)) == 0) {
#ifdef DEBUG
   printf("%s (null) ", p);
#endif
   return;
 if ((lpData = (LPTSTR)malloc(dwLen)) == NULL) {
  printf("malloc() error\n");
    exit(-1);
  if (GetFileVersionInfo(FileName, dwHandle, dwLen, lpData) == FALSE) {
    free(lpData);
#ifdef DEBUG
    printf("%s (null) ", p);
#endif
   return;
  if (VerQueryValue(lpData, "\\", (LPVOID)&pFileInfo, (PUINT)&BufLen) == FALSE) {
#ifdef DEBUG
   printf ("%s (null) ", p);
#endif
  } else {
#ifdef DEBUG
    printf("%d.%d.%d.%d\n", HIWORD(pFileInfo->dwFileVersionMS), LOWORD(pFileInfo->dwFileVersionMS),
HIWORD(pFileInfo->dwFileVersionLS), LOWORD(pFileInfo->dwFileVersionLS));
#endif
if (HIWORD(pFileInfo->dwFileVersionMS) > 4 && (dwBaseAddress = GetBaseAddress(FileName)) != -1)
    printf("%s %d.%d.%d.%d %08x\n", p, HIWORD(pFileInfo->dwFileVersionMS), LOWORD(pFileInfo->dwFileVersionMS), HIWORD(pFileInfo->dwFileVersionLS),
dwBaseAddress):
  free (lpData);
int main(int argc, char *argv[])
  WIN32 FIND DATA FindFileData;
  HANDLE hFind = INVALID_HANDLE_VALUE;
  DWORD dwError;
  char Path[MAX_PATH + 1], SearchPath[MAX_PATH + 1], FileName[MAX_PATH + 1];
```

```
ExpandEnvironmentStrings("%WINDIR%", Path, sizeof(Path));
  strcat(Path, "\\system32\\");
 strcpy(SearchPath, Path);
strcat(SearchPath, "*");
hFind = FindFirstFile(SearchPath, &FindFileData);
  if (hFind == INVALID HANDLE VALUE) {
    printf("Invalid file handle. Error is %u\n", GetLastError());
    return (-1);
  } else {
    strcpy(FileName, Path);
    strcat(FileName, FindFileData.cFileName);
PrintVersion(FileName);
#ifdef DEBUG
    printf("First file name is %s\n", FindFileData.cFileName);
    while (FindNextFile(hFind, &FindFileData) != 0) {
#ifdef DEBUG
     printf ("Next file name is %s\n", FindFileData.cFileName);
#endif
      strcpy(FileName, Path);
      strcat(FileName, FindFileData.cFileName);
      PrintVersion(FileName);
    dwError = GetLastError();
    FindClose(hFind);
    if (dwError != ERROR_NO_MORE_FILES) {
#ifdef DEBUG
       printf("FindNextFile error. Error is %u\n", dwError);
#endif
       return (-1);
    }
 return (0);
```

The plan is now to launch this small program on as many Windows 2000 boxes as possible with different service packs, patches, and localization. The results are presented in the following graph:





And here is the list of the common DLLs to all these platforms:

These DLLs are pretty much unused in commonly exploited programs such as Windows services, with the exception of msvcirt.dll and msvcp50.dll that can be found from time to time (in the Message Queueing Service for example).

admparse.dll	5.0.2920.0	0x80000000
bootvid.dll	5.0.2172.1	0x80010000
dbmsadsn.dll	1999.10.20.0	0x42bd0000
dbmssocn.dll	1999.10.20.0	0x73330000
dbmsspxn.dll	1999.10.20.0	0x42be0000
gpkcsp.dll	5.0.2134.1	0x8000000
mcdsrv32.dll	5.0.2160.1	0x80010000
msvcirt.dll	6.1.8637.0	0x780a0000
msvcp50.dll	5.0.0.7051	0x780c0000
rtipxmib.dll	5.0.2168.1	0xd0000000
slbcsp.dll	5.0.2134.1	0x8000000
slbkygen.dll	5.0.2144.1	0x8000000
sqlwid.dll	1999.10.20.0	0x412f0000
vcdex.dll	5.0.2134.1	0x0ffb0000
vdmredir.dll	5.0.2134.1	0x0ffa0000

Looking Deeper in Memory

So nothing terribly exciting can be found in the default DLLs of Windows. Fortunately, there are several other memory areas allocated in a process containing data that can be of interest, among those are:

- Stacks
- Heaps
- File mappings
- PEB, TEBs
- Various different kinds of sections...

We can usually exclude stacks and heaps from our research since we can not rely on their content and location, nevertheless we should not restrain ourselves to the DLLs and EXEs of a process, but look into the whole process memory for opcodes of interest, which is what the small C program dumpop.c does.

```
DumpOp v0.21 (c) 2005 Kostya Kortchinsky <kostya[at]immunityinc[dot]com>
dumpop.exe [-d] [-i <InputFile>] [-o <OuputFile>] [-j <Register> | -s] [-p <PID>]
Exemple d'utilisation : esp dans services.exe de Windows 2000
1ere machine : dumpop.exe -p 212 -j esp -o services esp.dmp
=> Cela va sauvegarder toutes les adresses possibles d'un jmp esp ou equivalent
  dans le fichier services_esp.dmp. On peut afficher les resultats a l'ecran
  en rajoutant le flag -d
2eme machine : dumpop.exe -p 212 -j esp -i services esp.dmp -d
=> Cela charge les resultats precedemment trouves puis les compare a ceux que
   l'on trouve et on affiche les resultats a l'ecran. On peut aussi les
   sauvegarder via le flag -o.
Resultat : esp dans services.exe de Windows 2000 SP4 français vs. anglais
001f5028::ffe4 (sortkey.nls)
001ecad4::54c3 (sortkey.nls)
77f5e214::54c3 (gdi32.dll)
7800ccdd::54c3 (msvcrt.dll)
#include <windows.h>
#include <stdio.h>
#include "getopt.h"
enum { JmpEax = 0, JmpEcx, JmpEdx, JmpEbx, JmpEsp, JmpEsi, JmpEdi, PopPopRet };
static UCHAR Registers[][4] = { "eax", "ecx", "edx", "ebx", "esp", "ebp", "esi", "edi" };
typedef struct
```

```
UCHAR *Data;
USHORT Length;
   UCHAR Type;
} OPCODE;
static OPCODE Opcodes[] =
      "\xff\xd0", 2, JmpEax },
"\xff\xe0", 2, JmpEax },
"\x50\xc3", 2, JmpEax },
"\xff\xd1", 2, JmpEcx },
"\xff\xe1", 2, JmpEcx },
      "\x51\xc3", 2, JmpEcx
"\xff\xd2", 2, JmpEdx
       "\xff\xe2", 2, JmpEdx
"\x52\xc3", 2, JmpEdx
"\xff\xd3", 2, JmpEbx
       "\xff\xe3", 2, JmpEbx "\x53\xc3", 2, JmpEbx
     "\x53\xc3", 2,
"\xff\xe4", 2,
"\x54\xc3", 2,
"\xff\xd5", 2,
"\xff\xe5", 2,
"\xff\xd6", 2,
"\xff\xe6", 2,
"\xff\xe6", 2,
"\xff\xe3", 2,
"\xff\xe4", 2,
"\xff\xe4", 2,
"\xff\xd7", 2,
                                  JmpEsp
                                  JmpEsp
                                  JmpEbp
                                  JmpEbp
                                  JmpEbp
                                  JmpEsi
                                  JmpEsi
                                  JmpEsi },
      "\xff\xd7", 2, JmpEdi },
"\xff\xd7", 2, JmpEdi },
"\xff\xe7", 2, JmpEdi },
"\x57\xc3", 2, JmpEdi },
"\x58\x58\xc3", 3, PopPopRet },
"\x58\x58\x59\xc3", 3, PopPopRet },
       "\x58\x5a\xc3", 3, PopPopRet },
       "\x58\x5b\xc3", 3, PopPopRet },
       "\x58\x5c\xc3", 3, PopPopRet
       "\x58\x5d\xc3", 3,
                                          PopPopRet
       "\x58\x5e\xc3", 3,
                                         PopPopRet
       "\x58\x5f\xc3", 3,
                                          PopPopRet
       "\x59\x58\xc3", 3,
                                          PopPopRet
       "\x59\x59\xc3", 3,
"\x59\x5a\xc3", 3,
"\x59\x5b\xc3", 3,
                                          PopPopRet
                                          PopPopRet
                                          PopPopRet
       "\x59\x5c\xc3", 3,
"\x59\x5d\xc3", 3,
                                          PopPopRet
      "\x59\x5d\xc3", 3,
"\x59\x5e\xc3", 3,
"\x59\x5f\xc3", 3,
"\x5a\x58\xc3", 3,
"\x5a\x59\xc3", 3,
"\x5a\x5b\xc3", 3,
"\x5a\x5b\xc3", 3,
"\x5a\x5d\xc3", 3,
"\x5a\x5d\xc3", 3,
"\x5a\x5d\xc3", 3,
"\x5a\x5d\xc3", 3,
                                          PopPopRet
                                          PopPopRet
                                          PopPopRet
                                          PopPopRet
                                          PopPopRet },
                                          PopPopRet },
                                          PopPopRet },
                                          PopPopRet
                                          PopPopRet
       "\x5a\x5e\xc3", 3,
"\x5a\x5f\xc3", 3,
                                          PopPopRet
                                          PopPopRet
       "\x5b\x58\xc3", 3,
"\x5b\x59\xc3", 3,
                                          PopPopRet
                                          PopPopRet
       "\x5b\x5a\xc3", 3,
                                          PopPopRet
       "\x5b\x5b\xc3", 3,
                                          PopPopRet
       "\x5b\x5c\xc3", 3,
                                          PopPopRet
       "\x5b\x5d\xc3", 3,
                                          PopPopRet
       "\x5b\x5e\xc3", 3,
                                          PopPopRet
       "\x5b\x5f\xc3", 3,
                                          PopPopRet
       "\x5c\x58\xc3", 3,
"\x5c\x59\xc3", 3,
                                          PopPopRet
                                          PopPopRet
       "\x5c\x5a\xc3", 3,
"\x5c\x5b\xc3", 3,
"\x5c\x5b\xc3", 3,
"\x5c\x5c\xc3", 3,
"\x5c\x5c\xc3", 3,
"\x5c\x5c\xc3", 3,
                                          PopPopRet
                                          PopPopRet
                                          PopPopRet
                                          PopPopRet },
                                          PopPopRet },
       "\x5c\x5e\xc3", 3,
"\x5c\x5f\xc3", 3,
"\x5d\x58\xc3", 3,
"\x5d\x59\xc3", 3,
"\x5d\x5a\xc3", 3,
                                          PopPopRet
                                          PopPopRet
                                          PopPopRet },
                                          PopPopRet },
       "\x5d\x5b\xc3", 3,
"\x5d\x5c\xc3", 3,
"\x5d\x5d\xc3", 3,
                                          PopPopRet
                                          PopPopRet
                                          PopPopRet
       "\x5d\x5e\xc3", 3,
                                          PopPopRet
       "\x5d\x5f\xc3", 3,
                                          PopPopRet
       "\x5e\x58\xc3", 3,
                                          PopPopRet
      "\x5e\x59\xc3", 3,
"\x5e\x5a\xc3", 3,
                                          PopPopRet
                                         PopPopRet
       "\x5e\x5b\xc3", 3, PopPopRet },
"\x5e\x5c\xc3", 3, PopPopRet },
```

```
"\x5e\x5d\xc3", 3, PopPopRet },
"\x5e\x5d\xc3", 3, PopPopRet },
"\x5e\x5f\xc3", 3, PopPopRet },
"\x5f\x58\xc3", 3, PopPopRet },
"\x5f\x59\xc3", 3, PopPopRet },
"\x5f\x5b\xc3", 3, PopPopRet },
"\x5f\x5b\xc3", 3, PopPopRet },
"\x5f\x5b\xc3", 3, PopPopRet },
"\x5f\x5d\xc3", 3, PopPopRet },
"\x5f\x5d\xc3", 3, PopPopRet },
"\x5f\x5d\xc3", 3, PopPopRet },
"\x5f\x5e\xc3", 3, PopPopRet },
"\x5f\x5e\xc3", 3, PopPopRet },
"\x5f\x5e\xc3", 3, PopPopRet }
static ULONG NumberOfValidFirstBytes = 0;
static UCHAR ValidFirstBytes[256];
typedef struct
  LPVOID Address;
  ULONG OpcodeIndex;
} RETURN_ADDRESS, *PRETURN_ADDRESS;
void Usage(void)
  fprintf(stderr, "Usage:\ndumpop.exe [-d] [-i <InputFile>] [-o <OuputFile>] [-j <Register> | -s] [-p
<PID>]\n");
  exit(EXIT FAILURE);
ULONG DumpOpcode (ULONG Opcode, UCHAR *Buffer, ULONG Length)
  ULONG Index;
  static UCHAR HexString[] = "0123456789abcdef";
  memset (Buffer, 0, Length);
  for (Index = 0; (Index < Opcodes[Opcode].Length) && (Index < (Length >> 1)); Index++) {
     Buffer[Index << 1] = HexString[Opcodes[Opcode].Data[Index] >> 4];
    Buffer[(Index << 1) + 1] = HexString[Opcodes[Opcode].Data[Index] & Oxf];</pre>
  return (Index << 1);
BOOL ISINList (LPVOID Address, PRETURN ADDRESS List, ULONG ListSize)
  ULONG Index;
  for (Index = 0; Index < ListSize; Index++)</pre>
     if (List[Index].Address == Address)
       return TRUE:
  return FALSE;
void SearchMemoryForOpcode(LPVOID Address, ULONG Size, LPVOID Buffer, UCHAR Type, PRETURN ADDRESS
*pList, ULONG *pListSize)
  ULONG Index, Offset;
  PRETURN ADDRESS pReturnAddress;
  for (Offset = 0; Offset < Size; Offset++) {</pre>
     for (Index = 0; Index < NumberOfValidFirstBytes; Index++)</pre>
       if (((UCHAR *)Buffer)[Offset] == ValidFirstBytes[Index])
         break;
     if (Index == NumberOfValidFirstBytes)
       continue;
     for (Index = 0; Index < (sizeof(Opcodes) / sizeof(Opcodes[0])); Index++) {
  if (Type == Opcodes[Index].Type && Opcodes[Index].Data[0] == ((UCHAR *)Buffer)[Offset]) {</pre>
          if ((Offset + Opcodes[Index].Length) >= Size)
            continue;
          if (memcmp((LPVOID)((ULONG)Buffer + Offset), Opcodes[Index].Data, Opcodes[Index].Length) == 0)
{
            if (*pListSize == 0)
               *pList = LocalAlloc(LPTR, sizeof(RETURN ADDRESS));
            else
               *pList = LocalReAlloc(*pList, (*pListSize + 1) * sizeof(RETURN_ADDRESS), LMEM ZEROINIT);
            pReturnAddress = &((*pList)[*pListSize]);
             (*pListSize)++;
            pReturnAddress->Address = (LPVOID) ((ULONG)Address + Offset);
            pReturnAddress->OpcodeIndex = Index;
   }
```

```
int main(int argc, char *argv[])
  INT Option, Error = EXIT_FAILURE;
 ULONG Index, Offset, ProcessId = 0;
HANDLE hInputFile = NULL, hOutputFile = NULL, hProcess = NULL;
UCHAR Type = (UCHAR)-1, String[64], *Unused, *InputFileName = NULL, *OutputFileName = NULL;
BOOL bResult, bDumpToScreen = FALSE;
  DWORD NumberOfBytes;
  DWORD InputListSize = 0, OutputListSize = 0, Size = 0;
  PRETURN ADDRESS InputList = NULL, OutputList = NULL;
  SYSTEM INFO SystemInfo;
  LPVOID CurrentAddress, Buffer;
  MEMORY BASIC INFORMATION MemoryBasicInformation;
  while ((Option = getopt(argc, argv, "dsp:j:i:o:")) != EOF) {
    switch (Option) {
      case 'p':
        ProcessId = strtoul(optarg, &Unused, 10);
        break;
      case 'j':
        if (Type != (UCHAR)-1)
          Usage();
        for (Type = 0; Type < (sizeof(Registers) / sizeof(Registers[0])); Type++)</pre>
           if (stricmp(Registers[Type], optarg) == 0)
            break;
        if (Type == (sizeof(Registers) / sizeof(Registers[0])))
          Usage();
        break:
      case 's':
if (Type != (UCHAR)-1)
          Usage();
        Type = PopPopRet;
        break;
      case 'd':
        bDumpToScreen = TRUE;
        break;
      case 'i':
        InputFileName = optarg;
        break;
      case 'o':
        OutputFileName = optarg;
        break;
      default:
        Usage();
        break;
    }
  if (ProcessId == 0 \mid \mid Type == (UCHAR)-1)
    Usage();
  if ((hProcess = OpenProcess(PROCESS QUERY INFORMATION | PROCESS VM READ, FALSE, ProcessId)) == NULL)
    fprintf(stderr, "[-] Error opening process %d\n", ProcessId);
    return Error;
  if (InputFileName != NULL) {
    hInputFile = CreateFile(InputFileName, GENERIC READ, FILE SHARE READ, NULL, OPEN EXISTING,
FILE ATTRIBUTE_NORMAL, NULL);
    if (hInputFile == INVALID_HANDLE_VALUE) {
      fprintf(stderr, "[-] Error opening file %s\n", InputFileName);
      goto Cleanup;
    bResult = ReadFile(hInputFile, &InputListSize, sizeof(InputListSize), &NumberOfBytes, NULL);
    if (bResult == FALSE || NumberOfBytes != sizeof(InputListSize)) {
   fprintf(stderr, "[-] Error reading file %s\n", InputFileName);
      goto Cleanup;
    InputList = LocalAlloc(LPTR, InputListSize * sizeof(RETURN_ADDRESS));
    if (InputList == NULL)
      fprintf(stderr, "[-] Error allocating buffer\n");
      goto Cleanup;
    bResult = ReadFile(hInputFile, InputList, InputListSize * sizeof(RETURN_ADDRESS), &NumberOfBytes,
NULL);
    if (bResult == FALSE | | NumberOfBytes != InputListSize * sizeof(RETURN ADDRESS)) {
      fprintf(stderr, "[-] Error reading file %s\n", InputFileName);
      goto Cleanup;
    for (Index = 0; Index < InputListSize; Index++)
      if (InputList[Index].OpcodeIndex >= (sizeof(Opcodes) / sizeof(Opcodes[0]))) {
        fprintf(stderr, "[-] Invalid return addresses list\n");
```

```
goto Cleanup;
    }
    memset(ValidFirstBytes, 0, sizeof(ValidFirstBytes));
for (Index = 0; Index < (sizeof(Opcodes) / sizeof(Opcodes[0])); Index++) {</pre>
       if (Type == Opcodes[Index].Type) {
  for (Offset = 0; Offset < NumberOfValidFirstBytes; Offset++)</pre>
               if (Opcodes[Index].Data[0] == ValidFirstBytes[Offset])
                  break;
            if (Offset == NumberOfValidFirstBytes)
               ValidFirstBytes[NumberOfValidFirstBytes++] = Opcodes[Index].Data[0];
       }
    GetSystemInfo(&SystemInfo);
    CurrentAddress = SystemInfo.lpMinimumApplicationAddress;
    while (CurrentAddress <= SystemInfo.lpMaximumApplicationAddress) {</pre>
        if (VirtualQueryEx(hProcess, (LPVOID)CurrentAddress, &MemoryBasicInformation,
sizeof(MemoryBasicInformation)) == 0)
           break;
        if ((MemoryBasicInformation.State & MEM_COMMIT) != 0) {
           if ((Buffer = VirtualAlloc(NULL, MemoryBasicInformation.RegionSize, MEM_COMMIT, PAGE_READWRITE))
== NULL)
               return EXIT_FAILURE;
\label{lem:condition} \mbox{if (ReadProcessMemory(hProcess, MemoryBasicInformation.BaseAddress, Buffer, MemoryBasicInformation.RegionSize, NULL) != 0)}
               {\tt Search Memory For Opcode (Memory Basic Information. Base Address, Memory Basic Information. Region Size, and the state of the stat
Buffer, Type, &OutputList, &OutputListSize);
    VirtualFree(Buffer, 0, MEM_RELEASE);
       CurrentAddress = (LPVOID) ((ULONG)CurrentAddress + MemoryBasicInformation.RegionSize);
    if (OutputFileName != NULL) {
       hOutputFile = CreateFile(OutputFileName, GENERIC WRITE, 0, NULL, CREATE ALWAYS,
FILE ATTRIBUTE_NORMAL, NULL);
       if (hOutputFile == INVALID_HANDLE_VALUE) {
  fprintf(stderr, "[-] Error creating file %s\n", OutputFileName);
           goto Cleanup;
       if (InputListSize != 0) {
            for (Index = 0; Index < OutputListSize; Index++)
               if (IsInList(OutputList[Index].Address, InputList, InputListSize) == TRUE)
                   Size++;
       else
           Size = OutputListSize;
       bResult = WriteFile(hOutputFile, &Size, sizeof(Size), &NumberOfBytes, NULL); if (bResult == FALSE | | NumberOfBytes != sizeof(Size)) {
           fprintf(stderr, "[-] Error writing to file %s\n", OutputFileName);
           goto Cleanup;
    for (Index = 0; Index < OutputListSize; Index++) {</pre>
        if (InputListSize == 0 || IsInList(OutputList[Index].Address, InputList, InputListSize) == TRUE) {
            if (bDumpToScreen == TRUE) {
               DumpOpcode(OutputList[Index].OpcodeIndex, String, sizeof(String));
               fprintf(stdout, "%08x::%s\n", OutputList[Index].Address, String);
            if (hOutputFile != NULL) {
               bResult = WriteFile(hOutputFile, &(OutputList[Index]), sizeof(RETURN ADDRESS), &NumberOfBytes,
NULL);
               if (bResult == FALSE || NumberOfBytes != sizeof(RETURN_ADDRESS)) {
                   fprintf(stderr, "[-] Error writing to file %s\n", OutputFileName);
                   goto Cleanup;
          }
       }
    Error = EXIT SUCCESS;
Cleanup:
    if (hInputFile != NULL && hInputFile != INVALID HANDLE VALUE)
       CloseHandle(hInputFile);
    if (hOutputFile != NULL && hOutputFile != INVALID HANDLE VALUE)
       CloseHandle (hOutputFile);
    if (hProcess != NULL)
       CloseHandle (hProcess);
    if (InputList != NULL)
       LocalFree(InputList);
```

```
if (OutputList != NULL)
   LocalFree(OutputList);
return Error;
```

Using this program to compare retrieve common return addresses for standard Windows services on various platforms, we could notice some that were common to all of them, located in some file mapping memory area.

NLS File Mapping

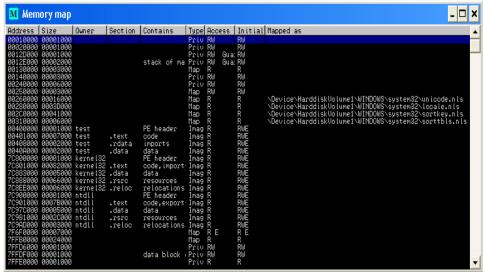
Several NLS files are mapped by default by Windows before the process even starts, those are:

- unicode.nls
- locale.nls
- sortkey.nls
- sorttbls.nls

Some others can be loaded at runtime depending on the locale used, as well as the DLLs initialized (ctype.nls for example). One of the advantages of these mapping is that even if they have no such thing as an image base address – like DLLs have – the mapping algorithm is deterministic and their mapping base address is (almost) fixed for a given binary on the same major version of Windows.

Indeed, their mapping base address will depend on previously allocated pages during the binary loading process: the stack of the main thread (based on the SizeOfStackReserve parameter in PE header – 0x100000 for Microsoft compiled binary, can be changed with /F for cl.exe), the imported DLLs (based on their image base address). Meaning that for standard Windows services, you will always find them at the same address for a given major version of Windows, which might not be the case for some other binaries. Among the advantages of these file mappings is that they contain a lot of jmp reg, call reg or push reg & ret opcodes, making them very useful for stack overflow, and that they haven't changed since Windows NT 4.0! Among their disadvantages: the addresses contain one NULL byte (the high-order byte) and are not marked as executable. In Windows NT up to and including XP SP1, the executable bit is not important for the memory section. So if you can use the null byte, you can use the address in the NLS section.

As a side note, the Blaster worm was using a return address in one of the NLS file mappings for its Windows 2000 targets, making it quite "universal".



Drawing 1: Immunity Debugger View of NLS Sections

Remote Language Fingerprinting

As mentioned above, we can usually get either cross-SP return addresses or cross-language return addresses but (except for NLS file mappings) not both. Since SP fingerprinting is painful and not very reliable, what about language fingerprinting? Microsoft Windows does not offer a documented remote and anonymous way to correctly determine the language pack of a Windows install, but it would be really useful since:

- The applied language pack changes offsets and base addresses within DLLs which affect our exploits
- Some vulnerabilities and/or exploits are only effective on certain languages:
 - o MS06-009: Korean Input Method Editor
 - o MS07-001: Brazilian Portuguese Grammar Checker

Remote language fingerprinting seems to be a promising approach for massively reliable exploitation. One of the aspects to consider is the Same Path Principle: when exploiting a flaw, we preferably want all the fingerprinting to be done using the same ports/services that the exploit is targeting, to reduce the risks of failure due to firewalling or services being disabled. Thus, we had to find as many ways as possible to remotely (and anonymously if possible) fingerprint a Windows install language, which include:

- MSRPC: using different interfaces and endpoints;
- The Windows SNMP service:
- IIS
- IE

Using MSRPC

MSRPC has been under particular attention during the last years, having carried the "best" anonymously and remotely exploitable flaws in Windows systems. I will not go

into the details of MSRPC, interfaces, endpoints, functions: you will be able to find a lot of information about this on the Interweb. MSRPC also happens to be one of the easiest and most reliable ways to fingerprint a Windows language install.

Shares

The MSRPC language fingerprint technique using share enumeration works by matching the "remark" unicode field of a SHARE_INFO_1 structure returned by the NetShareEnum() API. This function uses the interface 4b324fc8-1670-01d3-1278-5a47bf6ee188 v3.0, opnum 15 in services.exe (under 2000). This interface is usually accessible through endpoints on ncacn_np, ncadg_ip_udp (in old SP – was patched with the Windows Messenger Service vulnerability).

If the default IPC\$ or the C\$ share exists (or any other disk\$ share in fact), it will come with a remark field that is different depending on the Windows language. Anyway, if we are exploiting a RPC bug over ncacn_np, the IPC\$ share would better exist:)

This allows remote and anonymous language fingerprinting against NT 4.0, 2000, XP up to SP1a and 2003 SP0. It no longer works in XP SP2 and 2003 SP1.

The shares will allow to match uniquely the following languages:

- French
- Spanish
- Russian
- German
- Dutch
- Polish
- Simplified Chinese
- Traditional Chinese
- Turkish
- Hungarian
- Czech
- Norwegian
- Swedish
- Greek
- Danish
- Finnish

Unfortunately, for 5 languages, the remark field has not been localized, these are:

- Japanese
- Korean
- Hebrew
- Arabic
- English

These five languages all have the same remark field as an English install of Windows. Moreover, a few "collisions" exists in the remark field with the following languages using the IPC\$ share:

- Portuguese
- Italian
- Brazilian

This is reduced to Portuguese and Brazilian if using the C\$ share (or any other disk). The following program can be used to illustrate this technique:

fplang.c

```
FpLang v0.16 (c) 2005 Kostya Kortchinsky <kostya[at]immunityinc[dot]com>
Remotely fingerprints Windows localization. It uses the NetShareEnum RPC
function (NULL session needed - or credentials of course) to fetch the shares
and matches the "Remark" field of IPC$ and C$ shares to identify the language.
If we managed to connect, IPC$ should be listed, but there are a few collisions on some of the strings (like Italian & Portuguese). So if C$ is listed, we will
prefer this result.
It was tested against Windows 2000 platforms, but works fairly well against XP
and 2003 ... XP SP2 and 2003 SP1 will not allow the remote call anonymously
though.
#ifndef UNICODE
#define UNICODE
#endif
#pragma comment(lib, "netapi32")
#pragma comment(lib, "mpr")
#include <windows.h>
#include <stdio.h>
#include <lm.h>
typedef struct
  UCHAR *Remark;
  USHORT Size;
  UCHAR *Language;
} SHARE INFO;
SHARE INFO IpcInfoArray[] =
  Arabic, Hebrew, Japanese or Korean" }, { "\x52\x00\x65\x00\x66\x00\x6f\x00\x74\x00\x65\x00\x2d\x00\x49\x00\x50\x00\x43\x00", 20,
"German" },
  "French" },
  { "\x49\x00\x50\x00\x43\x00\x20\x00\x72\x00\x65\x00\x6d\x00\x6f\x00\x74\x00\x61\x00", 20,
"Spanish" },
  { "\x23\x04\x34\x04\x30\x04\x3b\x04\x35\x04\x3d\x04\x3d\x04\x4b\x04\x39\x04\x20\x00\x49\x00\x50\x00\
x43\x00", 26, "Russian" },
{ "\x49\x00\x50\x00\x43\x00\x20\x00\x72\x00\x65\x00\x6d\x00\x6f\x00\x74\x00\x6f\x00", 20, "Italian,
Portuguese or Brazilian" },
    "\x5a\x00\x64\x00\x61\x00\x6c\x00\x6e\x00\x65\x00\x20\x00\x49\x00\x50\x00\x43\x00", 20,
"Polish" },
    "x46\x00\x6a\x00\x6a\x00\x72\x00\x72\x00\x2d\x00\x49\x00\x50\x00\x43\x00", 18, "Swedish"},
   "\x45\x00\x74\x00\x65\x00\x72\x00\x66\x00\x65\x00\x20\x00\x49\x00\x50\x00\x43\x00", 22,
"Dutch" },
   "\x45\x00\x6b\x00\x73\x00\x74\x00\x65\x00\x72\x00\x6e\x00\x20\x00\x49\x00\x50\x00\x43\x00", 22,
"Norvegian" },
  { "\x54\x00\xe1\x00\x76\x00\x6f\x00\x6c\x00\x69\x00\x20\x00\x49\x00\x50\x00\x43\x00", 20,
"Hungarian" },
    "\x46\x00\x6a\x00\x65\x00\x72\x00\x6e\x00\x2d\x00\x49\x00\x50\x04\x00", 18, "Danish" },
   "\x45\x00\x74\x00\x2d\x00\x2d\x00\x49\x00\x50\x00\x43\x00", 14, "Finnish" },  
"\xdc\x8f\x0b\x7a\x20\x00\x49\x00\x50\x00\x43\x00", 12, "Simplified Chinese" },  
"\x60\x90\xef\x7a\x20\x00\x49\x00\x50\x00\x43\x00", 12, "Traditional Chinese" },
   "\x91\x03\xc0\x03\xbf\x03\xbc\x03\xb1\x03\xc1\x03\xc5\x03\xc3\x03\xbc\x03\xad\x03\xbd\x03\
"\x56\x00\x7a\x00\x64\x00\xe1\x00\x6c\x00\x65\x00\x6e\x00\xfd\x00\x20\x00\x49\x00\x30\x00"
 24, "Czech" }
SHARE_INFO DiskInfoArray[] =
```

```
x65\x00", 26, "English, Arabic, Hebrew, Japanese or Korean" },
                 x64\x00\xe9\x00\x66\x00\x61\x00\x75\x00\x72\x00\x73\x00\x26\x00\x20\x70\x00\x72\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x76\x00\x72\x00\x73\x00\x65\x00\x70\x00\x72\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x65\x00\x6
x65\x00\x74\x00\x65\x00\x72\x00\x6d\x00\x69\x00\x6e\x00\x61\x00\x64\x00\x6f\x00", 44, "Spanish" },
                 "\x21\x04\x42\x04\x30\x04\x3d\x04\x34\x04\x30\x04\x40\x04\x42\x04\x3d\x04\x4b\x04\x39\x04\x20\x00\
'Russian" },
                 "\x43\x00\x6f\x00\x6e\x00\x64\x00\x69\x00\x76\x00\x69\x00\x73\x00\x6f\x00\x6f\x00\x66\x00\x65\x00\
x20\x00\x70\x00\x72\x00\x65\x00\x64\x00\x65\x00\x66\x00\x69\x00\x6e\x00\x69\x00\x74\x00\x61\x00", 48,
"Italian" },
                "\x50\x00\x61\x00\x72\x00\x74\x00\x65\x00\x66\x00\x61\x00\x20\x00\x70\x00\x72\x00\x65\x00\
x69\x00\x61\x00\x42\x01", 30, "Polish" },
              x72\x00\x73\x00", 28, "Swedish" },
               x72\x00\x65\x00", 28, "Dutch" },
          { "\x44\x00\x65\x00\x6c\x00\x74\x00\x20\x00\x73\x00\x74\x00\x61\x00\x64\x00\x61\x00\x61\x00\x72\x00\
x64\x00\x72\x00\x73\x00\x73\x00\x75\x00\x72\x00\x73\x00", 40, "Norvegian" }, { "\x41\x00\x6c\x00\x61\x00\x70\x00\x22\x00\x74\x00\x6c\x00\x6c\x00\x65\x00\x7a\x00\x74\x00\x74\x00\x6c\x00\x6c\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x74\x00\x
x65 \times x00 \times 74 \times x00 \times x74 \times x00 \times x20 \times x00 \times x61 \times x00 \times x67 \times x00 \times x74 \times x00 \times
0", 50, "Hungarian" },

{ "\x53\x00\x74\x00\x61\x00\x64\x00\x61\x00\x64\x00\x72\x00\x64\x00\x73\x00\x68\x00\x61\x00\x72\x00\x64\x00\x73\x00\x68\x00\x72\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x72\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x00\x73\x0
x73\x00\x69\x00\x20\x00\x28\x00\x6a\x00\x61\x00\x65\x00\x74\x00\x75\x00\x29\x00", 46,
"Finnish" },
                "\xa0\x03\xc1\x03\xbf\x03\xb5\x03\xc0\x03\xb9\x03\xbb\x03\xb5\x03\xb3\x03\xbc\x03\xad\x03\xbd\x03\
xb7\x03\x20\x00\xba\x03\xbf\x03\xb9\x03\xbd\x03\xae\x03\x20\x00\xc7\x03\xc1\x03\xae\x03\xc3\x03\xb7\x0
3", 50, "Greek" },
               "\x56\x00\x61\x00\x72\x00\x73\x00\x61\x00\x79\x00\x31\x01\x6c\x00\x6e\x00\x6e\x00\x64\x00\x64\x00\x
x65\x00\x1f\x01\x65\x00\x72\x00", 32, "Turkish" },
              "\x56\x00\xfd\x00\x63\x00\x68\x00\x6f\x00\x7a\x00\xed\x00\x20\x00\x73\x00\x64\x00\xed\x00\x6c\x00\
x65\x00\x6e\x00\xe1\x00\x20\x00\x70\x00\x6f\x00\x6f\x00\x6f\x00\x7e\x01\x6b\x00\x,46,
"Czech" },
        x61 \times x00 \times x72 \times x00 \times x74 \times x00 \times x69 \times x00 \times x62 \times x00 \times x61 \times x00 \times x61 \times x00 \times x64 \times x00 \times x61 \times x00 \times x70 \times x00 \times x61 \times x00 
0\x72\x00\xe3\x00\x6f\x00", 56, "Brazilian" }
int wmain(int argc, TCHAR *argv[])
        PSHARE_INFO_1 pShareInfoBuf, pTmpBuf;
        NET API STATUS nStatus;
        LPTSTR pszServerName = NULL;
        DWORD i, j, dwEntriesRead = 0, dwTotalEntries = 0, dwResumeHandle = 0, dwError = 0;
        TCHAR pszRemoteName[MAX PATH];
        UCHAR pszLanguage[64];
        NETRESOURCE NetResource;
        memset(pszLanguage, 0, sizeof(pszLanguage));
        memset(pszRemoteName, 0, sizeof(pszRemoteName));
        strncpy(pszLanguage, "Unknown", sizeof(pszLanguage) - 1);
                fwprintf(stderr, L"[?] Usage: %s \\\ServerName\n", argv[0]);
        pszServerName = argv[1];
if (pszServerName[0] != '\\' || pszServerName[1] != '\\') {
  fprintf(stderr, "[-] Invalid ServerName (must start with \\\\)\n");
           _snwprintf(pszRemoteName, sizeof(pszRemoteName) - 1, L"%s\\IPC$", pszServerName);
        memset(&NetResource, 0, sizeof(NetResource));
        NetResource.lpLocalName = NULL;
        NetResource.lpProvider = NULL;
        NetResource.dwType = RESOURCETYPE ANY;
        NetResource.lpRemoteName = pszRemoteName;
         // Establish a NULL session
        if ((dwError = WNetAddConnection2(&NetResource, L"", L"", 0)) != NO ERROR) {
                fprintf(stderr, "[-] WNetAddConnection2() failed (%d)\n", dwError);
        fprintf(stderr, "[+] WNetAddConnection2() succeeded\n");
         // Enumerate the shares
        do {
```

```
nStatus = NetShareEnum(pszServerName, 1, (LPBYTE *)&pShareInfoBuf, MAX_PREFERRED_LENGTH,
&dwEntriesRead, &dwTotalEntries, &dwResumeHandle);
    if (nStatus == ERROR_SUCCESS || nStatus == ERROR_MORE_DATA) {
      fprintf(stderr, "[+] NetShareEnum() succeeded\n");
      // Let's first look for the IPC$ share \dots
      pTmpBuf = pShareInfoBuf;
      for(i = 0; i < dwEntriesRead; i++) {</pre>
        if (wcscmp((LPTSTR)(pTmpBuf->shi1_netname), L"IPC$") == 0) {
          for (j = 0; j < (sizeof(IpcInfoArray) / sizeof(IpcInfoArray[0])); j++)</pre>
            if ((wcslen((LPTSTR)pTmpBuf->shi1_remark) << 1) >= IpcInfoArray[j].Size
              && memcmp(pTmpBuf->shi1_remark, IpcInfoArray[j].Remark, IpcInfoArray[j].Size) == 0) {
              strncpy(pszLanguage, IpcInfoArray[j].Language, sizeof(pszLanguage) - 1);
          if (j == (sizeof(IpcInfoArray) / sizeof(IpcInfoArray[0]))) {
            for (j = 0; j < wcslen((LPTSTR)pTmpBuf->shi1 remark); j++)
              printf("\\x%02x\\x%02x", (UCHAR)(((LPTSTR) pTmpBuf->shil_remark))[j] & 0xff),
(UCHAR) (((LPTSTR) (pTmpBuf->shi1 remark))[j] >> 8));
            printf("\", %d, \"Unknown\" },\n", j << 1);
wprintf(L"%s\n", (LPTSTR)pTmpBuf->shi1_remark);
          break;
        pTmpBuf++;
      if (i == dwEntriesRead)
        printf("[!] No IPC\$ share\n"); // How would that be possible ?
      printf("[!] 1st guess : %s\n", pszLanguage);
strncpy(pszLanguage, "Unknown", sizeof(pszLanguage) - 1);
      // Now the C$ share ...
      pTmpBuf = pShareInfoBuf;
      for(i = 0; i < dwEntriesRead; i++) {</pre>
        if (wcscmp((LPTSTR)(pTmpBuf->shil_netname), L"C$") == 0) {
  for (j = 0; j < (sizeof(DiskInfoArray) / sizeof(DiskInfoArray[0])); j++)</pre>
            if ((wcslen((LPTSTR)pTmpBuf->shil_remark) << 1) >= DiskInfoArray[j].Size
   && memcmp(pTmpBuf->shil_remark, DiskInfoArray[j].Remark, DiskInfoArray[j].Size) == 0) {
              strncpy(pszLanguage, DiskInfoArray[j].Language, sizeof(pszLanguage) - 1);
          if (j == (sizeof(DiskInfoArray) / sizeof(DiskInfoArray[0]))) {
            printf("{ \"");
            for (j = 0; j < wcslen((LPTSTR)pTmpBuf->shi1_remark); j++)
              break;
        pTmpBuf++;
      if (i == dwEntriesRead)
        printf("[!] No C$ share\n");
      else
        printf("[!] 2nd guess : %s\n", pszLanguage);
      NetApiBufferFree(pShareInfoBuf);
      fprintf(stderr, "[-] NetShareEnum() failed (%d)\n", nStatus);
  } while (nStatus == ERROR MORE DATA);
  // Cancel the NULL session
 WNetCancelConnection2(pszRemoteName, 0, TRUE);
 return 0;
```

Users

Windows allows remote listing of the users on a system using LsaLookupSids() API by brute forcing SIDs. This technique matches the default ones that are localization dependent: for example the "None" group. The function is accessible through interface 12345778-1234-abcd-ef00-0123456789ab v0.0, being opnum 57, usually over ncacn_np.

It will work anonymously against NT 4.0 and 2000, and against XP up to SP1a with fake

credentials if a Share has been setup. It is quite useful in some case to refine a previous techniques results. In Immunity CANVAS, each technique is tried in turn, further reducing the set of possible languages until one has been reached.

Print Providers

The "Print Providers" method is the best of the RPC methods, and is unique to CANVAS. It works by matching the "comment" unicode field of a PRINTER_INFO_1 structure returned by the EnumPrinters() API. The API itself doesn't support remote listing of Print Providers, only local providers, so rather than using the Windows API, the attacker has to build their own RPC request to get it done. It need access to the Windows Spooler service (spoolsv.exe) interface 12345678-1234-abcd-ef00-0123456789ab v1.0, using opnum 0. This is usually accomplished through ncacn_np:\PIPE\spoolss which allows anonymous access even in Windows XP SP2!

So this method works remotely and anonymously against NT 4.0, 2000, XP up to and including SP2, but won't work against 2003 unless it is configured as a print server with \spoolss pipe enabled.

To get into the details, Windows based clients and servers have 3 print providers by default: localspl.dll, win32spl.dll and inetpp.dll. The comment string for win32spl.dll is localized and can be used for matching. In addition, 3rd party software packages can install their own print provider. This risk has been forced into the public eye with multiple vulnerabilities in print providers in the recent past:

- MS05-043: Heap overflow in win32spl.dll
- Novell TID #3125538: Stack overflow in nwspool.dll
- CTX111686: Stack overflow in cpprov.dll
- Etc.,

The printer provider localization detection method allows the attacker to uniquely match the following languages of Windows:

- French
- Spanish
- Russian
- German
- Dutch
- Polish
- Simplified Chinese
- Traditional Chinese
- Turkish
- Hungarian
- Czech
- Norwegian
- Swedish
- Greek

- Danish
- Finnish
- Japanese
- Korean
- Portuguese
- Italian
- Brazilian

Unfortunately, the comment string has not been translated for the following languages:

- Hebrew
- Arabic
- English

meaning that Windows install using these localizations will always be matched as English.

Regardless, this new method of using the printer providers gives the attacker the most reliable and widest range of results that can be recovered remotely and anonymously using MSRPC.

Using SNMP

SNMP has some advantages: it is widely used in corporate environments, and travels over UDP, making it an interesting target for attackers. Unfortunately, there is no such thing as a Windows Language OID that would allow an attacker to easily fetch the localization of the Windows install (well at least I haven't found one). There is a "Location" field (SNMPv2-MIB::sysLocation.0): it is a string that can be filled from SNMP services options by an Administrator, but since almost anything can be put in there, it is not useful for a reliable remote fingerprinting methodology.

Among the information that can be fetched from the SNMP service with read-access credentials, is a list of installed software (HOST-RESOURCES-

MIB::hrSWInstalledName.*), which also includes the hotfixes and patches installed on the machine. Luckily for the attacker, the term "Hotfix" is localized based on the Windows install version:

- "Correctif" in French
- "Revisión" in Spanish
- Etc.,

For this technique to work, the attacker will need a public community name and at least some hotfixes installed on the target machine. This is usually not a problem since no hotfix usually means a straight-forward exploitation of old "universal" bugs.

Using IIS and IE

When exploiting an IIS bug or an ISAPI filter, it is highly probable that no other port than 80 and/or 443 will be opened, and we will have to use that to try and retrieve some

information about the underlying Windows language. As expected IIS is not very talkative about its localization, the technique we have been using is based on the fact that the 40x error pages are localized, as well as some 40x error status strings (like the 404 one). Even if the 404 page is customized, there still are some other requests we can send to trigger errors for which pages are rarely modified.

Client-side exploits might also require a knowledge of the Windows language install, that can only be done through IE itself. The Accept-Language header can give an hint on the base language itself, but is not that reliable. Nowadays heap-spray provides a mean to disregard this, allowing exploitation regardless of fine-grained fingerprinting.

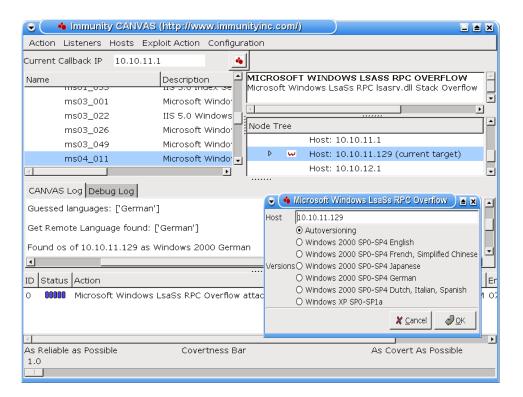
CANVAS Implementation

Based on the previously described research, CANVAS (and SILICA of course) implements these techniques to allow mass exploitation of Windows boxes without user interaction by fingerprinting the OS (major version, language pack) and launching the exploit based on the results. Some configuration options can be used to force some behaviors if no language could be found for a given host:

- Always assume a language: English for example;
- Don't run the exploit if not language was found;
- Assume the language to be the one of the nearest neighbor;

Common return addresses are used as often as possible to prevent language detection issues (mostly due to collisions).

Here is a example of a CANVAS run against a Windows 2000 German box, correctly identified:



Here is a list of some of the CANVAS exploits and the method chosen for reliability:

Exploit	Vulnerability	Method	Target
ms01_023	IPP ISAPI Overflow	NLS mapping	2000 SP0-SP1
ms01_033	Index Server ISAPI Overflow	NLS mapping	2000 SP0-SP1
ms03_001	RPC Locator Overflow	NLS mapping	NT 4.0 SP6a, 2000 SP0-SP3
ms03_022	Media Services ISAPI Overflow	NLS mapping	2000 SP0-SP4
ms03_026	RPC Interface Overflow	NLS mapping	NT 4.0 SP6a, 2000 SP0-SP4, XP SP0-SP1a, 2003 SP0
ms03_049	WksSvc Overflow	ws2help.dll address based on localization ws2help.dll address	2000 SP0-SP4, XP SP0-SP1a
ms04_011	LsaSs Overflow	based on localization	2000 SP0-SP4, XP SP0-SP1a
ms04_031	NetDDE RPC Overflow	NLS mapping	2000 SP0-SP4, XP SP0-SP1a
ms05_039	UPNP RPC Overflow	NLS mapping	NT 4.0 SP6a, 2000 SP0-SP4, XP SP0-SP1a
ms06_066	Netware Service Overflow	NLS mapping	2000 SP0-SP4, XP SP0-SP1a
ms06_070	WksSvc Overflow	NLS mapping	2000 SP0-SP4

Heap Overflows

Heap overflows are a different problem than stack overflows for reliability. Up to the point where we get a write-4, the Windows language pack should not be an issue. One notable exception is the Windows Messenger service overflow where the way the buffer is aligned depends on a string that is localized. Once we get the write-4, we usually need a function pointer overwritten to get control over the execution flow:

- The Unhandled Exception Filter (UEF) should be considered last resort since its location depends on both SP and language
- PEB lock functions are at a fixed location (independent of language pack and service pack) but may take as long as ten minutes to trigger, which is suboptimal
- To avoid an exception, the exploit writer often wants to find a writable location, which can be in .data section of a binary. This location depends on the binary's image base address and version.

For creating a reliable exploit despite these issues a memory leaks is often extremely useful. (In this context, a "memory leak" is a way to get the program to send contents of memory back to the attacker).

MSRPC Memory Leak

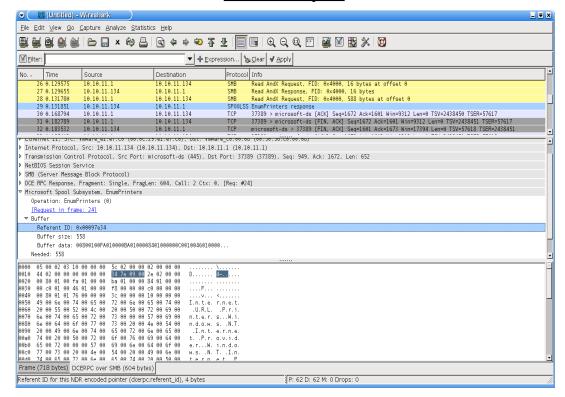
When dealing with MSRPC interfaces (once again), one has to know that the MIDL [unique] attribute leaks a pointer in the target process memory space on the wire if combined with [out]

• http://msdn2.microsoft.com/en-us/library/aa367294.aspx

Example

```
long _RpcEnumPrinters (
  [in] long arg_1,
  [in] [unique] [string] wchar_t * arg_2,
  [in] long arg_3,
  [in, out] [unique] [size_is(arg_5)] char * arg_4,
  [in] long arg_5,
  [out] long * arg_6,
  [out] long * arg_7);
```

Wireshark Capture



An ideal use of this memory leak would be the following:

- Populate target memory with an entry of your own using a 1st RPC function;
- Retrieve the entry using a 2nd RPC function with the MSRPC Pointer Leak
- You have the pointer to your entry!

This case doesn't happen that often, it could successfully be used for MS05-010: License Logging Service overflow. Even if this address is not directly used. it will give a good idea of the base address of the heap, which will turn out to be quite useful most of the time.

Example: Making MS06-074 Reliable

HEROES is the codename for the Windows SNMP Service arbitrary GlobalFree() vulnerability that was patched in Microsoft bulletin MS06-070. It is a great example of the heavy process that has to be carried out to make a vulnerability reliably work on several platforms.

The Windows SNMP Service (SNMP.exe listening on UDP/161) is vulnerable to a flaw that allows an attacker to trigger an arbitrary GlobalFree() operation. In order to trigger this vulnerability the attacker must know a read enabled community name.

Description

Consider the following pseudo-code:

```
array=(unsigned int *)malloc(n*sizeof(unsigned int *))
//initialization and various operations on array
...
i=0;
while (condition) {
   free(array[i++]);
   //process some more, update condition
   ...
}
```

If the while-condition is erroneously evaluated the loop will continue freeing memory referenced by pointer values taken from beyond the boundaries of the pointer array. Meaning you are able to trigger arbitrary calls to free(). This basic bug primitive exists within the Windows SNMP Service.

The actual impact of this flaw is largely dependent on being able to control the contents of the memory after the pointer array (which is located on a heap). Following an in-depth analysis of the vectors involved in exploiting this flaw Immunity has successfully developed a working and reliable exploit for this vulnerability.

Technical Details

The vulnerability is located within the UpdateVarBindsFromResolvedVbl() function which lives in the snmp.exe module. It can be reached by sending a SNMPv2c GetBulkRequest (see <u>RFC1905</u> for details).

GetBulk requests take several parameters such as 'non_repeaters' and 'max_repetitions'. They also take a list of variable bindings consisting of Object Instance Identifiers (OID) and actual variable values. The variable values are mostly ignored and usually set to NULL.

By carefully crafting the parameters to a SNMPv2c GetBulk request, an attacker can gain control over the exit-condition of the SnmpUtilVarBindCpy() & SnmpUtilVarBindFree() loop. This in turn leads to triggering an arbitrary GlobalFree() and reflects the previously discussed bug primitive.

The main reason we are able to trigger this condition is that, in certain cases, the SNMP service fails to properly initialize the repetition counter of the variable bindings. When the call to FindSupportedRegion() on a variable binding fails, the repetition counter will be left at 0 instead of being initialized to 1. This initiates the chain of events that allows for a full blown remote exploitation of the Windows SNMP service.

Exploitability

Several issues arise when attempting to exploit this vulnerability:

- How can we control the pointer that will be freed?
- Given pointer control, what do we actually want to free?
- Once we get our Write4 primitive, what will we overwrite?
- How do we leverage our Write4 primitive into full blown code execution?

To answer the first question consider the following; Considering all dynamic allocations take place on the same heap, we have to make sure we end up controlling the memory located after the initialized array. This in turn allows a SnmpUtilVarBindFree() to occur on data we control.

We can achieve this goal by playing with the size of the data we submit to the SNMP service. We control this size through the number of variable bindings and the actual size of the OIDs in our request. Note that the OIDs are decoded as an array of DWORDs.

The second question of what we actually want to free is a bit tricky. The heap base will change with every run of the SNMP service. That means the exact heap layout will depend on the amount of client interaction with the service at any given time. That means we can not rely on a specific static heap layout.

Fortunately for us, the exact same bug that triggers the arbitrary GlobalFree() can also be used to leak memory via the SnmpUtilVarBindCpy() operation.

SnmpUtilVarBindCpy enables us to overwrite data the SNMP service will send back to us. This lets us receive the variable bindings we originally sent, but with OIDs that now contain portions of heap memory. This leaked memory in turn contains the address of where our sent OIDs are located on the heap.

Considering we are dealing with a LookAside enabled heap, it is likely that this same chunk of OID memory will be reused for a later allocation of the same size. This means we can now reliably determine an address that points to data we control.

Since we can leak an address pointing to memory we control, we are able to trigger a GlobalFree() on this very memory. Thus, abusing the heap coalescing algorithm, we answer the first part of the third question and obtain our Write4 primitive. This means we can reliably write any 4 bytes anywhere in memory. Now we just have to figure out what to overwrite.

Preferably we only rely on write targets related to the SNMP service itself. Doing this makes us more generic than using write targets related to the OS core. The usual PEB RtlEnterCriticalSection pointer is a potential candidate, but due to timing reasons, has not proven itself to be a very reliable option. An alternative resides in the .data section of the snmpapi.dll module. This .data section contains several function pointers, some which are called in the SnmpSvcGetUptime() function. This function is in turn called when an authentication trap occurs. Authentication traps are raised if a request was made with a wrong community name and if the

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\SNMP\Parameters\E nableAuthenticationTraps registry key is set. It just so happens to be that this key is set to 1 by default.

So by overwriting one of the snmpapi Auth-trap triggered function pointers and sending a request with an invalid community name, we can reliably redirect execution. This answers question number four.

In order to fully exploit all of the above we redirect execution to an address that points just after our fake heap chunks and into our payload. This way everything needed for our exploit can be calculated relative to a single, leaked, address. Meaning we end up being quite reliable.

In order to finally make the exploitation language independent, we looked for an OID that would trigger a call to the PEB lock routines. Once an OID found – and triggering reliably the call – we only had to use the fixed PEB lock routine address and trigger the call with a SNMPv2 Get request for our OID.

Conclusion

Given the fact that all public DEP disabling techniques under XP SP2 require knowledge about addresses in ntdll.dll or kernel32.dll, and that the image base address of those are language dependent, the language fingerprinting techniques are definitely needed in the near future for large scale reliable exploitation of bugs.

Moreover, Vista introduces more languages, thus more targets and reliability issues. Likewise, expanding these techniques to other operating systems, such as OS X and Linux can require similar protocol analysis.

From the Immunity point of view, attacking large scale global networks can be done effectively by spending a fairly reasonable amount of time doing reliable fingerprinting.