Reverse Engineering Microsoft Binaries

Alexander Sotirov

asotirov@determina.com

Recon 2006

Overview

In the next one hour, we will cover:

- Setting up a scalable reverse engineering environment
  - getting binaries and symbols
  - building a DLL database
- Common features of Microsoft binaries
  - compiler optimizations
  - data directories
  - exception handling
  - hotpatching
- Improving IDA
  - IDA autoanalysis problems
  - loading debugging symbols
  - improving the analysis with IDA plugins

Why Microsoft Binaries?

Most reverse engineering talks focus on reversing malware, but Microsoft binaries present a very different challenge:

- Bigger than most malware
- No code obfuscation (with the exception of licensing, DRM and PatchGuard code)
- Debugging symbols are usually available
- Compiled with the Microsoft Visual C++ compiler
- Most of the code is written in object oriented C++
- Heavy use of COM and RPC

Part I

Setting Up a Scalable Reverse Engineering Environment

Getting the Binaries

Most Microsoft software, including older versions and service packs, is available for download from MSDN. We download these manually.

To download security updates automatically, we used the information in mssecure.xml, which is automatically downloaded by MBSA 1.2.1. The XML file contains a list of security bulletins for Windows, Office, Exchange, SQL Server and other products. It also provides direct download links to the patch files.

Unfortunately MBSA 1.2.1 was retired at the end of March 2006. The XML schema used by MBSA 2.0 is different, and our scripts don't support it yet.

Once you have all old security updates, downloading the new ones every month can be done manually.

Extracting the Binaries

- CAB and most EXE files can be unpacked with `cabextract`
- MSI and MSP files are difficult to unpack. Usually they contain CAB archives that can be extracted, but the files in them have mangled names. Still working on a solution.
- An administrative installation is our temporary solution for dealing with Microsoft Office.
- Some IIS files are renamed during the installation. For example smtpsvc.dll is distributed as smtp_smtpsvc.dll in IMS.CAB on the Windows 2000 installation CD.
- Recent patches use intra-package delta compression (US patent application 20050022175). Unpacking them with cabextract gives you files with names like `_sfx_0000._p`. To unpack these patches, you have to run them with the /x command line option.

DLL Database

We have an internal database of binaries indexed by the name and SHA1 hash of the file. We store the following file metadata:

• name	*ntdll.dll*
• size	*654336 bytes*
• modification date	*May 01, 2003, 3:56:12 PM*
• SHA1 hash	*9c3102ea1d30c8533dbf5d9da2a47…*
• DBG and PDB path	*Sym/ntdll.pdb/3E7B64D65/ntdll.pdb*
• source of the file	
○ product	*Windows XP*
○ version	*SP1*
○ security update	*MS03-007*
○ build	*qfe*
○ comment	

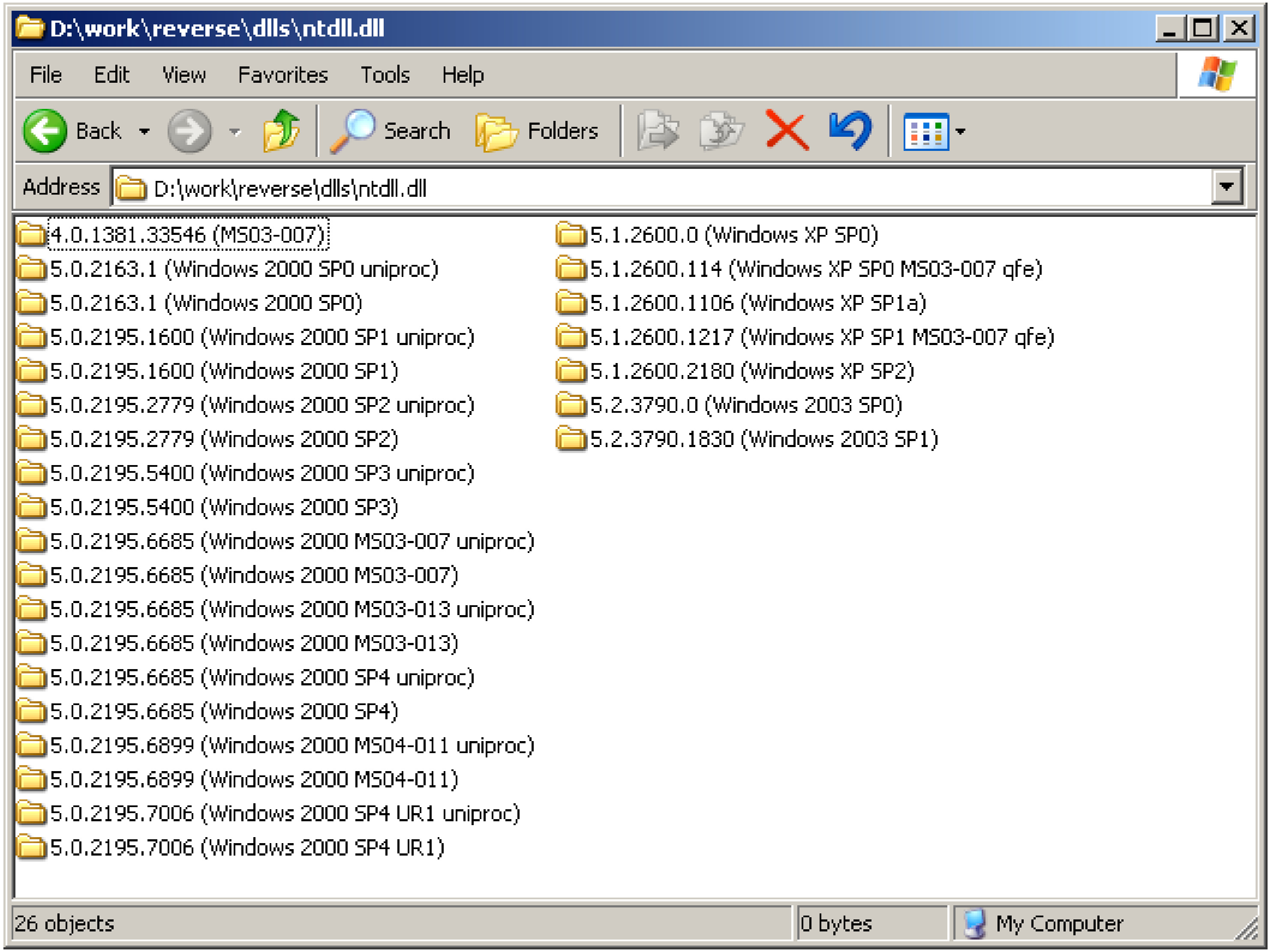
DLL Database

Current size of our database, including all service packs and security updates for Windows 2000, XP and 2003:

- 30GB of files
- 7GB of symbols
- 7500 different file names
- 28800 files total

and growing…

DLL Database



D:\work\reverse\dlls\ntdll.dll
File Edit View Favorites Tools Help
Back Search Folders
Address D:\work\reverse\dlls\ntdll.dll

4.0.1381.33546 (MS03-007)
5.0.2163.1 (Windows 2000 SP0 uniproc)
5.0.2163.1 (Windows 2000 SP0)
5.0.2195.1600 (Windows 2000 SP1 uniproc)
5.0.2195.1600 (Windows 2000 SP1)
5.0.2195.2779 (Windows 2000 SP2 uniproc)
5.0.2195.2779 (Windows 2000 SP2)
5.0.2195.5400 (Windows 2000 SP3 uniproc)
5.0.2195.5400 (Windows 2000 SP3)
5.0.2195.6685 (Windows 2000 MS03-007 uniproc)
5.0.2195.6685 (Windows 2000 MS03-007)
5.0.2195.6685 (Windows 2000 MS03-013 uniproc)
5.0.2195.6685 (Windows 2000 MS03-013)
5.0.2195.6685 (Windows 2000 SP4 uniproc)
5.0.2195.6685 (Windows 2000 SP4)
5.0.2195.6899 (Windows 2000 MS04-011 uniproc)
5.0.2195.6899 (Windows 2000 MS04-011)
5.0.2195.7006 (Windows 2000 SP4 UR1 uniproc)
5.0.2195.7006 (Windows 2000 SP4 UR1)
5.1.2600.0 (Windows XP SP0)
5.1.2600.114 (Windows XP SP0 MS03-007 qfe)
5.1.2600.1106 (Windows XP SP1a)
5.1.2600.1217 (Windows XP SP1 MS03-007 qfe)
5.1.2600.2180 (Windows XP SP2)
5.2.3790.0 (Windows 2003 SP0)
5.2.3790.1830 (Windows 2003 SP1)

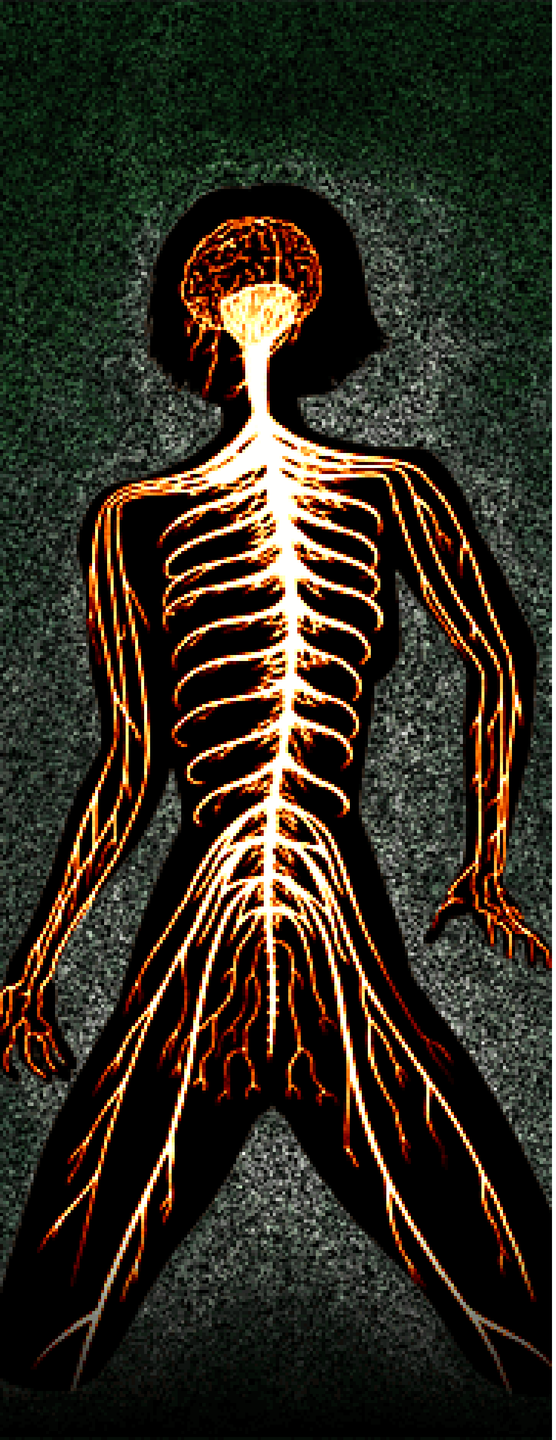
Getting Symbols

Microsoft provides symbols for most Windows binaries. They can be downloaded from their public symbol server by including it in your symbol path. See the Debugging Tools for Windows help for more information.

Use symchk.exe to download symbols for a binary and store them in a local symbol store.

We have scripts that automatically run symchk.exe for all new files that are added to the binary database.

Most Windows binaries have symbols, with the exception of some older Windows patches. In this case BinDiff can be used to compare the binaries and port the function names from another version that has symbols. Unfortunately symbols are not available for Office and most versions of Exchange.



Part II

Common Features of Microsoft Binaries

Common Features of Microsoft Binaries

- Visual C++ compiler optimizations
  - function chunking
  - function fall-through
  - array reference to the body of a function
  - reuse of stack frame slots
  - sbb comparison optimization
  - shr comparison optimization
  - switch optimization
- Data directories
- Exception handling
- Microsoft hotpatching

Function Chunking

Function chunking is a compiler optimization for improving code locality. Profiling information is used to move rarely executed code outside of the main function body. This allows pages with rarely executed code to be swapped out.

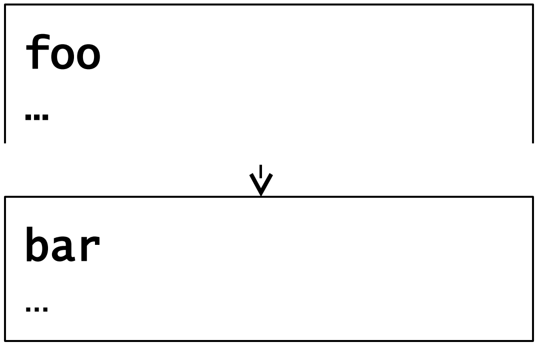
It completely breaks tools that assume that a function is a contiguous block of code. IDA has supported chunked functions since version 4.7, but its function detection algorithm still has problems in some cases.

This optimization leaks profiling information into the binary. We know that the code in the main function body is executed more often than the function chunks. For code auditing purposes, we can focus on the function chunks, since they are more likely to contain rarely executed and insufficiently tested code.

Function Fall-through

If foo is a wrapper around bar, the compiler can put the two functions next to each other and let foo fall through to bar.

```
void foo(a)
{
    if (a == 0)
        return;
    else
        bar(a);
}
```



Array Reference to the Body of a Function

Given the array reference A [eax-1] and the constraint eax >= 1, the compiler will convert the reference from

```
dec eax
mov ebx, A[eax*4]
```

to

```
mov ebx, B[eax*4]
```

where B is the address of A-4

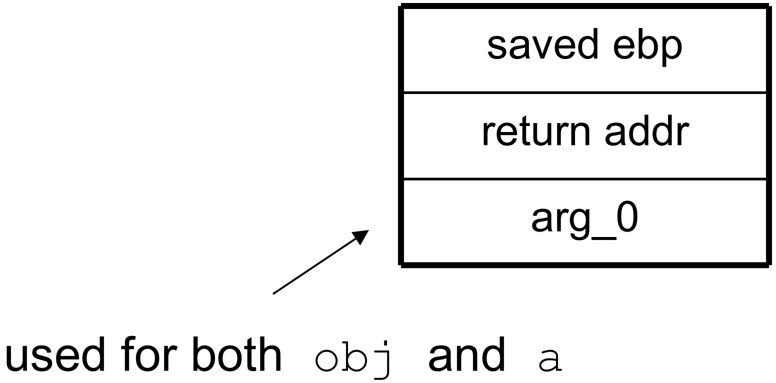
If the array is located right after a function, the address of A-4 will be inside the function and might be disassembled as data, even though the first 4 bytes are never referenced.

Reuse of Stack Frame Slots

In non-optimized code, there is a one-to-one correspondence between local variables and the stack slots where they are stored. In optimized code, the stack slots are reused if there are multiple variables with non-overlapping live ranges.

For example:

```
int foo(Object* obj)
{
      int a = obj->bar();
      return a;
}
```



saved ebp

return addr

arg_0

used for both `obj` and `a`

The live ranges of `obj` and `a` don't overlap, so they can be stored in same slot on the stack. The argument slot for `obj` is used for storing both variables.

SBB Comparison Optimization

The SBB instruction adds the second operand and the carry flag, and subtracts the result from the first operand.

- `sbb eax, ebx`

  eax = eax - (ebx + CF)

- `sbb eax, eax`

  eax = eax - (eax + CF)

  eax = - CF

SBB Comparison Optimization

The SBB instruction can be used to avoid branching in an `if` statement.

in assembly:

	ebx < ecx	ebx >= ecx
`cmp ebx, ecx`	CF = 1	CF = 0
`sbb eax, eax`	eax = -1	eax = 0
`inc eax`	eax = 0	eax = 1

in C:

```
if (ebx >= ecx)
        eax = 1;
else
        eax = 0;
```

SHR Comparison Optimization

in assembly:

```
shr ecx, 10h
test cx, cx
jnz foo
```

in C:

```
if (ecx > 65535)
        goto foo;
```

I've seen this in multiple files, but it is not clear if this is a compiler optimization or if the programmer just used a division operator:

```
if (ecx / 65535 == 0)
        goto foo;
```

Switch Optimization

Non-optimized code :

```
switch (arg_0)
{
        case 1:         ...
        case 2:         ...
        case 3:         ...
        case 8001:      ...
        case 8002:      ...
}
```

```
00401030                cmp [ebp+arg_0], 1
00401034                jz short case_1
00401036                cmp [ebp+arg_0], 2
0040103A                jz short case_2
0040103C                cmp [ebp+arg_0], 3
00401040                jz short case_3
```

Switch Optimization

Optimized code:

```
767AFDA1 _GetResDesSize@4 proc near
767AFDA1
767AFDA1       arg_0  = dword ptr 4
767AFDA1
767AFDA1              mov eax, [esp+arg_0]
767AFDA5              mov ecx, 8001h
767AFDAA              cmp eax, ecx
767AFDAC              ja short greater_than_8001
767AFDAE              jz short case_8001
767AFDB0              dec eax
767AFDB1              jz short case_1        ; after 1 dec
767AFDB3              dec eax
767AFDB4              jz short case_2        ; after 2 decs
767AFDB6              dec eax
767AFDB7              jz short case_3        ; after 3 decs
```

Data Directories

The PE header contains a list of IMAGE_DATA_DIRECTORY entries, each specifying a starting address and the size of the data. The data directories contain the DLL imports and exports, debugging information, delayed loading information and more.

Some of the data directories are located in their own PE sections, but most of the time the data directories are in the .text or .data sections. IDA will often try to disassemble the contents of a data directory as code or data. This might lead to a confusing disassembly.

Exception Handling

This is better than anything I could have said about it:

Reversing Microsoft Visual C++ Part I: Exception Handling
by Igor Skochinsky:

http://www.openrce.org/articles/full_view/21

Microsoft Hotpatching

The Microsoft hotpatching implementation is described in US patent application 20040107416. It is currently supported only on Windows 2003 SP1, but we'll probably see more of it in Vista.

The hotpatches are generated by an automated tool that compares the original and patched binaries. The functions that have changed are included in a file with a .hp.dll extension. When the hotpatch DLL is loaded in a running process, the first instruction of the vulnerable function is replaced with a jump to the hotpatch.

The /hotpatch compiler option ensures that the first instruction of every function is a `mov edi, edi` instruction that can be safely overwritten by the hotpatch. Older versions of Windows are not compiled with this option and cannot be hotpatched.

Part III

Improving IDA

Improving IDA

- IDA autoanalysis
  - Overview of the autoanalysis algorithm
  - Problems with the disassembly
- Loading debugging symbols
  - IDA PDB plugin
  - Determina PDB plugin

Autoanalysis Algorithm

The autoanalysis algorithm is not documented very well, but it can be roughly described as follows:

1. Load the file in the database and create segments
2. Add the entry point and all exports to the analysis queue
3. Find all typical code sequences and mark them as code. Add their addresses to the analysis queue
4. Get an address from the queue and disassemble the code at that address, adding all code references to the queue
5. If the queue is not empty, go to 4
6. Make a final analysis pass, converting all unexplored bytes in the text section to code or data

For more details, see this post by Ilfak Guilfanov:
http://www.hexblog.com/2006/04/improving_ida_analysis.html

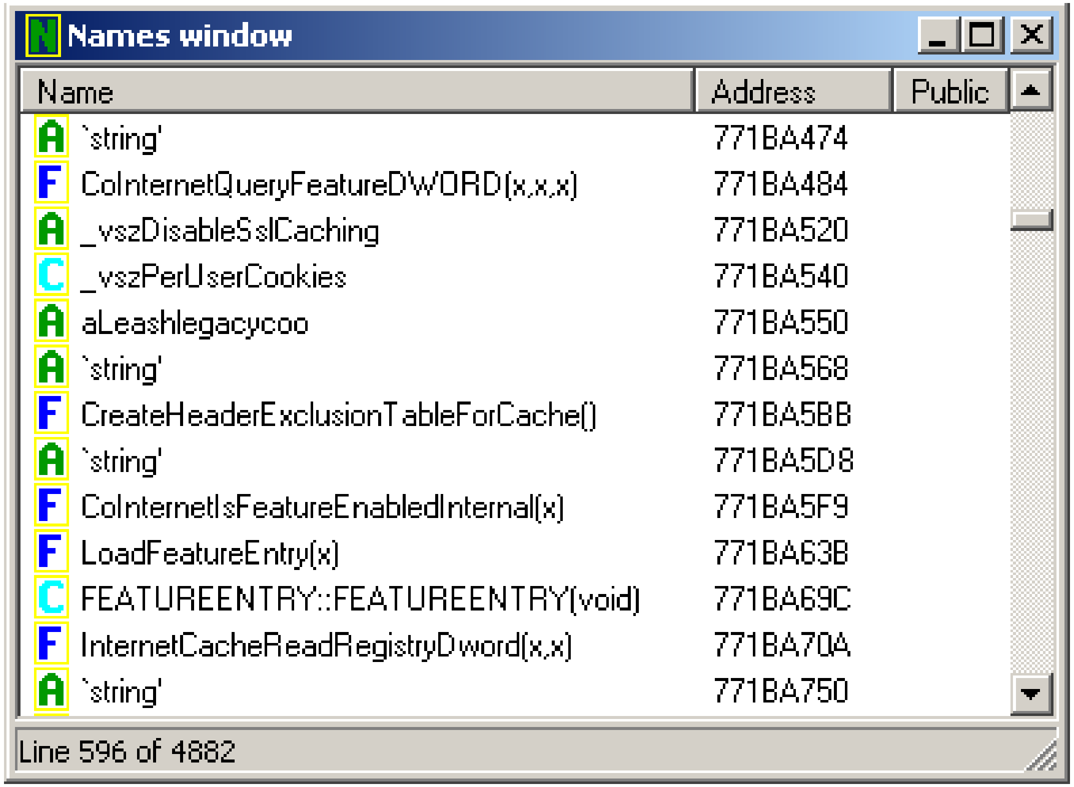
Autoanalysis Problems

There are a number of situations where the autoanalysis heuristics lead to incorrect disassembly. Some of these problems create serious difficulties for automatic analysis tools like BinDiff. The two main problem areas are:

- Data disassembled as code
- Function detection and function chunking problems

Autoanalysis Problems

Code outside a function is an indication of incorrectly disassembled data or a function detection problem:



	Name	Address	Public
A	`string'	771BA474	
F	CoInternetQueryFeatureDWORD(x,x,x)	771BA484	
A	_vszDisableSslCaching	771BA520	
C	_vszPerUserCookies	771BA540	
A	aLeashlegacycoo	771BA550	
A	`string'	771BA568	
F	CreateHeaderExclusionTableForCache()	771BA5BB	
A	`string'	771BA5D8	
F	CoInternetIsFeatureEnabledInternal(x)	771BA5F9	
F	LoadFeatureEntry(x)	771BA63B	
C	FEATUREENTRY::FEATUREENTRY(void)	771BA69C	
F	InternetCacheReadRegistryDword(x,x)	771BA70A	
A	`string'	771BA750	

Line 596 of 4882

should be a string →

should be a function →

Data Disassembled as Code

```
771B7650 ; const CHAR _vszSyncMode
771B7650 _vszSyncMode:
771B7650              push ebx
771B7651              jns short near ptr loc_771B76BF+2
771B7653              arpl [ebp+6Fh], cx
771B7656
771B7656 loc_771B7656:
771B7656              db 64h, 65h
771B7656              xor eax, 48000000h
771B765D              imul esi, [ebx+74h], 2E79726Fh
771B7664              dec ecx
771B7665              inc ebp
```

Instead of:

```
771B7650 ; const CHAR _vszSyncMode
771B7650 _vszSyncMode db 'SyncMode5',0
```

Data Disassembled as Code

Solution:

- disable "Make final analysis pass"

The final analysis pass runs after the initial autoanalysis is complete and converts all unexplored bytes in the text segment to data or code. It is often too aggressive and disassembles data as code. Disabling the option ensures that only real code is disassembled, but might leave some functions unexplored. If it is disabled, only the first element in a vtable is analyzed, leaving the rest of the member functions unexplored.

Data Disassembled as Code

Solution:

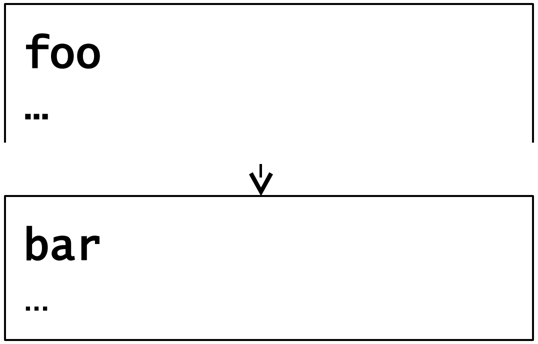
- use symbol names to distinguish code from data
- create data items before functions

Public debugging symbols don't include type information, but it is often possible to determine if a symbol is a function or data from its name. For example, GUID variables that are used to refer to COM objects often start with the same prefix. This allows us to define them as data in the IDA database.

Creating data items before functions establishes the boundaries of the functions. IDA will not undefine a data item if it falls within the body of a function, even if there are erroneous code references to it.

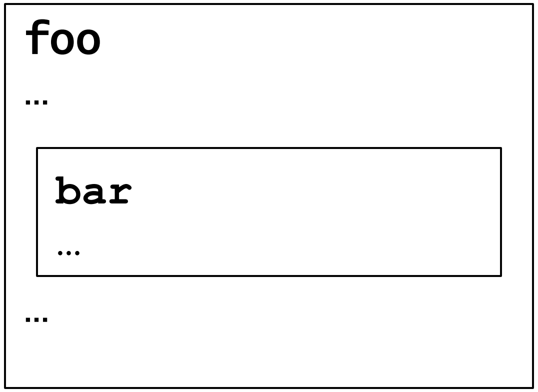
Function Chunking Problems

- If foo is a wrapper around bar, the compiler can put the two functions next to each other and let foo fall through to bar.



If function foo is analyzed first, it will include the code of bar. Even if there are calls to bar later, IDA will not create a function there.

- Function chunks inside another function



If function foo is analyzed first, it will include the bar chunk. If bar is analyzed first, foo will be split in two chunks around the bar chunk.

Function Chunking Problems

Solution:

- create the functions in reverse order, starting at the bottom of the file and going up

This is a very simple solution with an amazing effect on the accuracy of function chunking. IDA can deal with functions that fall-through or include code from other functions, as long as the other functions are created first. Since code usually flows downwards, we just need to create the functions at the bottom of the file before those on top.

Improving the Analysis

The best way to improve the analysis is to give IDA more information. We have focused on improving the PDB plugin that is used to load public debugging symbols from Microsoft.

IDA PDB Plugin

- Source code included in the IDA SDK
- Uses the DbgHelp API
- Supports DBG and PDB symbols through dbghelp.dll
- Algorithm:
    - create a name in the database
    - if the symbol name is `` `string' ``, create a C or UNICODE string
    - if the demangled name is not of type MANGLED_DATA, create a function at that address

Determina PDB Plugin

- Uses FPO records to detect functions and symbol names to determine data types
- Does not create functions for demangled names of an unknown type
  - reduces the instances of data disassembled as code
- Special handling for imports, floats, doubles and GUIDs
- Better string type detection (ASCII vs. UNICODE strings)
- Creates vtables as arrays of function pointers
- Applies symbols in multiple passes, and creates functions starting at the bottom of the file and going up
  - significantly improves function chunking
- Much better GUI

Determina PDB Plugin

Available under a BSD license from:

`http://www.determina.com/security.research/`

Version 0.4 released today!

Symbol Types

Public debugging symbols are stripped and don't include type information. We have to rely on the symbol names and the availability of FPO records to determine their types. The following types are recognized:

• import	`__imp__FunctionName`
• float	`__real@3fc00000`
• double	`__real@0000000000000000`
• string	`` `string' ``
• guid	starts with a prefix like `_IID_`, `_SID_`, `__GUID_`
• vtable	contains `` `vtable' `` in the name
• function	has an FPO record, or the demangler returns `MANGLED_CODE`
• data	the demangler returns `MANGLED_DATA`
• unknown	everything else

Applying the Symbols

When the user clicks OK, the symbols are applied in 4 passes:

Pass 1: If a symbol location already has a name, change it to the symbol name. This pass makes sure that there are no duplicate names during the second pass.

Pass 2: Set the names of all symbols in the database. Having all names in the database before the next pass is necessary to avoid the creation of data items that don't fit in the space before the next symbol.

Pass 3: Iterate through all data symbols and create data items. The data is left undefined if there's not enough space for a data item of the right type (4 bytes for floats, 8 bytes for doubles, 16 bytes for GUIDs)

Pass 4: Iterate through all function symbols in reverse order and create functions.

Determina PDB Plugin Demo

Questions?

asotirov@determina.com