Metadata Internals

Wintellect
Raise Your Windows IQ™

About this talk

- This talk is about:
  - Metadata tables and tokens
  - System.Reflection and IMetaDataImport
  - Metadata on-disk format
  - Metadata scopes and reference resolution
- This talk is NOT about:
  - System.Reflection details
  - Dynamic binding
  - Reflection.Emit

Overview

- What is metadata?
- How do I use the reflection API?
- How do I use the unmanaged metadata API?
- What's different about these API's?
- How do I deal with metadata scopes?

What is metadata?

- Metadata: *n. data about data*
- Describes types
  - Member fields, methods, properties, events
- Describes signatures
  - Fields, properties, methods, delegates, local vars
- Describes types and *references*
- Describes miscellaneous entities
  - Files, modules, assemblies

Metadata in action

- Metadata by itself is useless
  - Like abstract base classes in C++ / Java / C#
  - Like IDL / TLB in COM
- CIL instructions reference metadata via **tokens**
  - First byte of token indicates type
  - Last three bytes of token are either row # (tables) or offsets (heaps)
  - Tokens are stored compressed in signatures (binary file viewers *not* very useful)

How to examine tokens

- Use ILDASM
  - Use /TOKENS option: displays token values
  - Use /BYTES option: displays CIL bytes
  - Use /OUT option: dump disassembly to disk file

```
ildasm /TOKENS /BYTES /OUT=HelloWorld.il HelloWorld.exe
```

How to examine metadata heaps

- Use MetaInfo
  - Fully documented sample metadata reader
  - In Tool Developers Guide\Samples\MetaInfo
- You must build it yourself!
  - See comments in metainfo.mak
  - Use short path names (GetShortPathName API)

```
metainfo /raw /heaps mscorlib.dll > dump.txt
```

Hello, World

```
class HelloWorld
{
    static void Main(string[] args)
    {
     Console.WriteLine( "Hello, World" );
    }
}
```

```
.method /*06000001*/ private hidebysig static
        void  Main(string[] args) cil managed
{
  .entrypoint
  .maxstack  8
  IL_0000:  /* 72   | (70)000001 */ ldstr   "Hello, World"      /* 70000001 */
  IL_0005:  /* 28   | (0A)00000E */ call    void [mscorlib      /* 23000001 */ ]
                                            System.Console      /* 0100000F */
                                            ::WriteLine(string) /* 0A00000E */
  IL_000a:  /* 2A   |                  */   ret
} // end of method HelloWorld::Main
```

About HelloWorld.exe

- Five tokens (and types) in listing

MethodDef:	(0x06000001)
User string:	(0x70000001)
AssemblyRef:	(0x23000001)
TypeRef:	(0x0100000F)
MemberRef:	(0x0A00000E)

- Only two are directly referenced by CIL in this example:
  - String (by **ldstr** instruction)
  - MemberRef (by **call** instruction)

What's a string token?

- String tokens refer to strings
  - Offset into user string (#US) heap
- Points to UTF-8 string
  - Length prefixed (PackedLen) (includes null)
  - Terminated by single-byte null
- Used only by **ldstr** instruction

Definition vs. reference tokens

- Two general categories of metadata tokens
  - Definition tokens
  - Reference tokens
- Definition tokens define the entity
  - e.g. TypeDef tokens describe definition of Types
- Reference tokens define references to the entity
  - e.g. TypeRef token describes how to find TypeDef

A MemberRef is logically …

- MemberRefs reference "type members"
  - Fields, Methods
  - Colloquially referred to as FieldRef, MethodRef
- What information does a MemberRef need?
  - Member's signature (think method overloading)
  - Type that member is found in
  - Assembly that type is found in

A MemberRef is physically …

- Dereferencing MemberRef token yields table row
- MemberRef row contains
  - Parent (typically a TypeRef token)*
  - Name (offset in #Strings heap)
  - Signature (offset in #Blob heap)
- * simplified for purposes of discussion
- Documented in ECMA Partition II spec

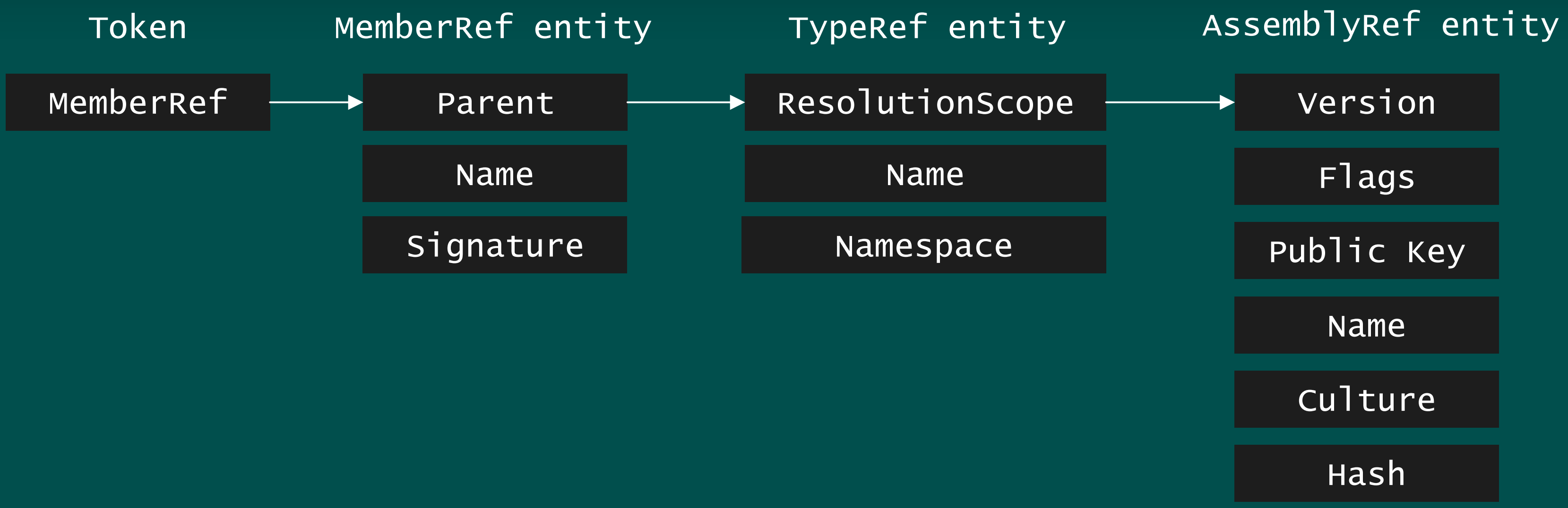
TypeRef tokens

- Dereferencing TypeRef token yields TypeRef row
- TypeRef row contains
  - ResolutionScope (typically an AssemblyRef token)*
  - Name (string)
  - Namespace (string)
- * simplified for purposes of discussion

AssemblyRef tokens

- AssemblyRef row contains
  - Major, Minor, Build, Revision #'s (ushort)
  - Flags (uint)
  - Public key (blob)
  - Name (string)
  - Culture (string)
  - Hash value (blob)

MemberRef


Token
MemberRef entity
TypeRef entity
AssemblyRef entity
MemberRef
Parent
Name
Signature
ResolutionScope
Name
Namespace
Version
Flags
Public Key
Name
Culture
Hash

Metadata scopes

- Metadata definition tokens scoped by module
  - TypeDef, MethodDef, AssemblyDef ...
- Metadata reference tokens scoped by app domain
  - TypeRef, MemberRef, AssemblyRef, ModuleRef
- CLR loads into app domain:
  - Statically referenced assemblies at start-up
  - Dynamically referenced assemblies as needed

Metadata API's

- Two major metadata API's available
  - Managed: System.Reflection
  - Unmanaged: IMetaDataImport and friends
- Both are limited in different ways
  - System.Reflection does not permit access to CIL
  - System.Reflection does not reveal token values
  - IMetaDataImport cannot resolve *Ref tokens

System.Reflection

- API for inspecting metadata
  - Provides a *logical* view of metadata
  - Physical details (e.g. token values, CIL) obscured
- API for emitting metadata (and CIL)
  - System.Reflection.Emit

Loaded assemblies

- AppDomain object returns array of assemblies

```
private static void EnumerateLoadedAssemblies()
{
  Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
  foreach( Assembly assembly in assemblies )
    Console.WriteLine( assembly.FullName );
}
```

Referenced assemblies

- Assembly object returns array of AssemblyNames
- AssemblyName == AssemblyRef
  - Version, Name, Culture, Public Key, Hash

```
private static void EnumerateReferencedAssemblies( Assembly assembly )
{
  AssemblyName[] referencedAssemblies = assembly.GetReferencedAssemblies();
  foreach( AssemblyName assemblyName in referencedAssemblies )
    Console.WriteLine( "-- {0}", assemblyName.FullName );
}
```

Types in an assembly

- Assembly object returns array of Type objects
    - Struct, Enum, Class, Interface

```
private static void EnumerateTypes( Assembly assembly )
{
  Type[] types = assembly.GetTypes();
  foreach( Type type in types )
    Console.WriteLine( "== {0}", type.FullName );
}
```

Methods in a type

- Type object exposes array of MethodInfo objects
- MethodInfo represents method
  - Signature, return value, visibility ...

```
private static void EnumerateMethods( Type type )
{
  MethodInfo[] methods = type.GetMethods();
  foreach( MethodInfo method in methods )
    Console.WriteLine( "**** {0}", method.Name );
}
```

Method's CIL

- System.Reflection does not permit access to CIL
  - API would need to understand / expose tokens
  - API would be significantly more complex
- Need to use a different API

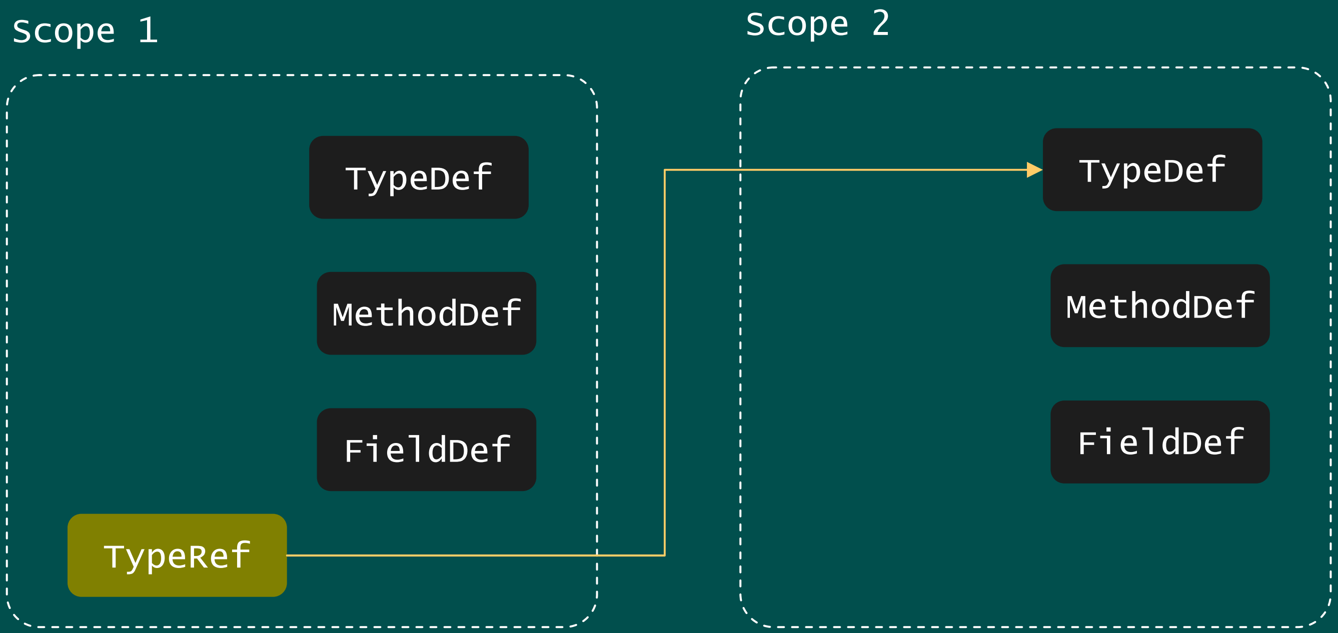
Unmanaged metadata API

- COM-based API
  - CLSID_CorMetaDataDispenser implementation
  - Initial interface is IMetaDataDispenserEx

```
hr = CoCreateInstance( CLSID_CorMetaDataDispenser,
                       0,
                       CLSCTX_INPROC_SERVER,
                       IID_IMetaDataDispenserEx,
                       reinterpret_cast< void** >( &_spMDD )
                       );
```

Metadata scopes

- Dispensers enable access to *scopes*
  - Scopes define boundary for metadata definitions
  - Scopes are usually found in physical files



Opening a scope

- Open scope using OpenScope() method
  - Returns IMetaDataImport interface
  - Each module has its own scope

```
SIMetaDataImport OpenScope( SIMetaDataDispenserEx dispenser, string path )
{
  USES_CONVERSION;
  SIMetaDataImport result;

  dispenser->OpenScope( A2W( path.c_str() ),
                        0,                                          // read
                        IID_IMetaDataImport,
                        reinterpret_cast< IUnknown** >( &result )
                        );
  return result;
}
```

Enumerators

- Enumerators iterate over items in a scope
  - Identified by HCORENUM handle
  - Types, Methods, Fields, Parameters ...
- Usage: acquire enumerator, use, close

```
void Enumerate( SIMetaDataImport metadata )
{
  HRESULT   hr;
  HCORENUM  hEnum = 0;
  hr = metadata->EnumXXX( &hEnum, // [in/out] enumerator handle
                          ...
                          );
  if( SUCCEEDED( hr ) )
  {
    // Use
    metadata->CloseEnum( hEnum );
  }
}s
```

Enumerating types

- Use EnumTypeDefs() method to retrieve tokens

```
void EnumTypes( SIMetaDataImport metadata )
{
  HRESULT   hr;
  DWORD     count;
  HCORENUM  hEnumTypeDef = 0;  // NOTE that this is an in/out param and MBZ!
  mdTypeDef typeDefs[ MAX_TYPEDEFS_PER_MODULE ];

  hr = metadata->EnumTypeDefs( &hEnumTypeDef,         // [in/out] enumerator handle
                               typeDefs,              // [out] array of mdTypeDefs
                               NumItems( typeDefs ), // [in] size of array
                               &count                 // [out] number of items
                               );
  if( SUCCEEDED( hr ) )
  {
    for( unsigned int i = 0; i < count; i++ )
      cout << GetTypeName( typeDefs[ i ], metadata ).c_str() << endl;
    metadata->CloseEnum( hEnumTypeDef );
  }
}
```

Enumerating types

- Use GetTypeDefProps() to retrieve token data

```
string GetTypeName( mdTypeDef typeDef, SIMetaDataImport import )
{
  USES_CONVERSION;
  HRESULT hr;
  ULONG   sizeName;
  wchar_t wszName[ MAX_SYMBOL_LENGTH ];

  hr = import->GetTypeDefProps( typeDef,               // [in] typeDef token
                                wszName,               // [out] buffer for name
                                NumItems( wszName ),   // [in] size of name buffer
                                &sizeName,             // [out] size of name
                                0,                     // [out] CorTypeAttrs
                                0                      // [out] type it extends
                                );
  _ASSERT( SUCCEEDED( hr ) );
  _ASSERT( sizeName < NumItems( wszName ) );

  return W2A( wszName );
}
```

Enumerating methods

- Use EnumMethods() to retrieve tokens

```
void EnumMethods( SIMetaDataImport metadata, mdTypeDef type )
{
  HRESULT     hr;
  DWORD       count;
  HCORENUM    hEnumMethods = 0;
  mdMethodDef methodDefs[ MAX_METHODDEFS_PER_TYPE ];

  hr = metadata->EnumMethods( &hEnumMethods,              // [in/out] enumerator handle
                              type,                       // [in] typeDef
                              methodDefs,                 // [out] mdMethodDefs array
                              NumItems( methodDefs ),     // [in] size of array
                              &count                      // [out] number of items
                              );
  if( SUCCEEDED( hr ) )
  {
    for( unsigned int i = 0; i < count; i++ )
      cout << GetMethodName( methodDefs[ i ], metadata ).c_str() << endl;
    metadata->CloseEnum( hEnumMethods );
  }
}
```

Enumerating methods

- Use GetMethodProps() to retrieve token data

```
string GetMethodName( mdMethodDef methodDef, SIMetaDataImport import )
{
  USES_CONVERSION;
  HRESULT hr;
  ULONG   sizeName;
  wchar_t wszName[ MAX_SYMBOL_LENGTH ];
  hr = import->GetMethodProps( methodDef,             // [in] methodDef token
                               0,                     // [out] typeDef token
                               wszName,               // [out] buffer for name
                               NumItems( wszName ),   // [in] size of name buffer
                               &sizeName,             // [out] size of name
                               0,                     // [out] CorMethodAttrs
                               0,                     // [out] ptr to sig blob
                               0,                     // [out] size of sig blob
                               0,                     // [out] RVA to code
                               0                      // [out] implementation flags
                               );
  _ASSERT( SUCCEEDED( hr ) );
  _ASSERT( sizeName < NumItems( wszName ) );

  return W2A( wszName );
}
```

RVA to offset conversion

- Start of method's CIL determined by RVA
  - Relative Virtual Address
- Convert to offset into PE file
  - ImageRvaToVa() in dbghelp.lib

```
pHeader = (unsigned __int8*) ImageRvaToVa( g_File->getNTHeaders(),
                                           g_File->getBase(),
                                           rva,
                                           0
                                           );
```

Examining CIL

- CIL stream contains two sections
  - Header
  - CIL instructions
- Header depends on size of CIL stream
  - First byte determines type of header
  - CorILMethod_TinyFormat
  - CorILMethod_FatFormat

Examining CIL

- Convert to byte pointer

```
unsigned __int8* GetMethodPointer( ULONG rva, ULONG* codeSize )
{
  unsigned __int8*        pHeader;
  IMAGE_COR_ILMETHOD_FAT* pFatMethodHeader;
  pHeader = (unsigned __int8*) ImageRvaToVa( g_File->getNTHeaders(),
                                             g_File->getBase(), rva, 0 );
  switch( *pHeader & CorILMethod_FormatMask )
  {
  case CorILMethod_TinyFormat:
    *codeSize = ((TinyHeader*)pHeader)->size;
    return ++pHeader;
  case CorILMethod_FatFormat:
    pFatMethodHeader = reinterpret_cast< IMAGE_COR_ILMETHOD_FAT* >( pHeader );
    *codeSize = pFatMethodHeader->CodeSize;
    return reinterpret_cast< unsigned __int8* >( pFatMethodHeader )
           + pFatMethodHeader->Size * sizeof( DWORD );
  }
}
```

Examining CIL

- Enumerate over byte pointer

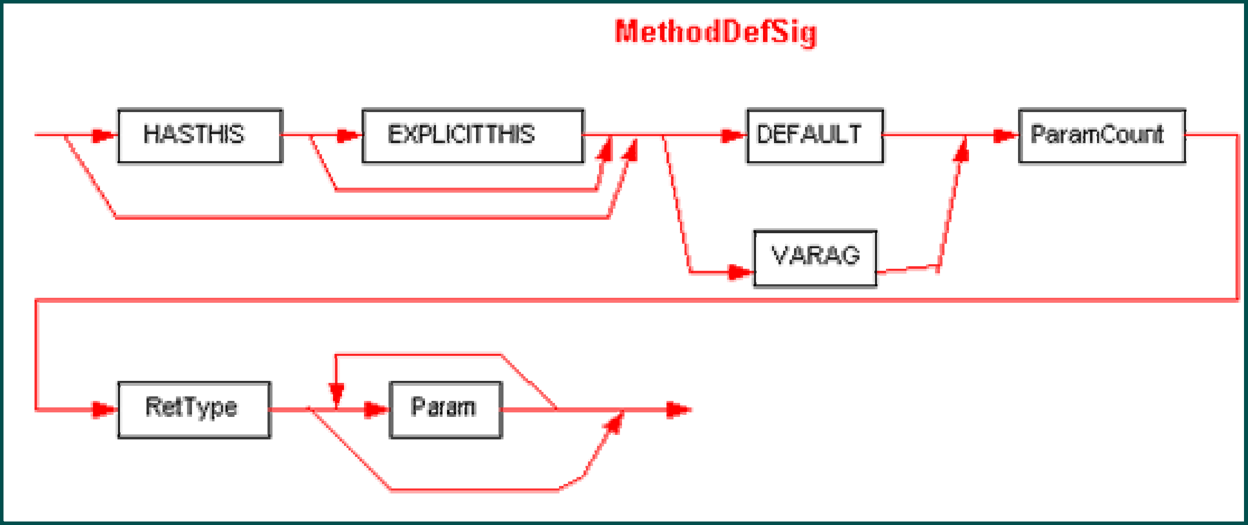
```
void DumpCIL( SIMetaDataImport metadata, mdMethodDef methodDef )
{
  HRESULT hr;
  ULONG   rva;
  hr = metadata->GetMethodProps( methodDef, 0, 0, 0, 0, 0, 0, 0,
                                 &rva, 0 );
  if( SUCCEEDED( hr ) )
  {
    ULONG codeSize;
    unsigned __int8* pMethodBytes = GetMethodPointer( rva, &codeSize );

    for( ULONG i = 0; i < codeSize; i++ )
    {
      char szOutput[ 254 ];
      wsprintf( szOutput, "%2.2x", *pMethodBytes++ );
      cout << szOutput << " ";
    }
    cout << endl;
  }
}
```

Signatures and blobs

- Signatures describe many elements
  - Methods, fields, properties, local variables …
- Signatures are variable-length data structures
  - Stored in metadata blob heap (#Blob)
- Method signatures describe
  - Parameters, calling convention, return values …

Cracking method signatures

- Method signatures expose
  - Calling convention
  - Parameter list
  - Return values



Cracking method signatures

```
MethodDefSig::MethodDefSig( PCCOR_SIGNATURE sig, ULONG sigSize )
{
  PCCOR_SIGNATURE sigPos = sig;
  sigPos += CorSigUncompressData( sigPos, &_methodSig );   // read flags
  sigPos += CorSigUncompressData( sigPos, &_paramCount );  // read count

  _retvalSig = new RetvalSig( sigPos );  // crack return value sig
  sigPos += _retvalSig->TypeSize();      // advance pointer

  for( int i = 0; i < _paramCount; i++ )
  {
    ParamSig* paramSig = new ParamSig( sigPos ); // crack parameter sig
    sigPos += paramSig->TypeSize();              // advance pointer
    _paramSigs.push_back( paramSig );            // add to param sig array
  }

  _ASSERT( sigPos == sig + sigSize );   // sanity check
}
```

Uncompressing tokens

- Tokens are stored using compressed format
- Bits 0, 1 indicate token type
  - TypeDef (00), TypeRef (01), TypeSpec (10)
- Bits 2-31 stored as compressed integer
- Type info for all other token types discarded
  - Must discern token type from context

```
private static uint CorSigUncompressToken( BinaryReader reader )
{
  uint token = CorSigUncompressData( reader );
  return (token >> 2) | (uint)CorEncodeToken[ token & 0x03 ];
}
```

Uncompressing tokens

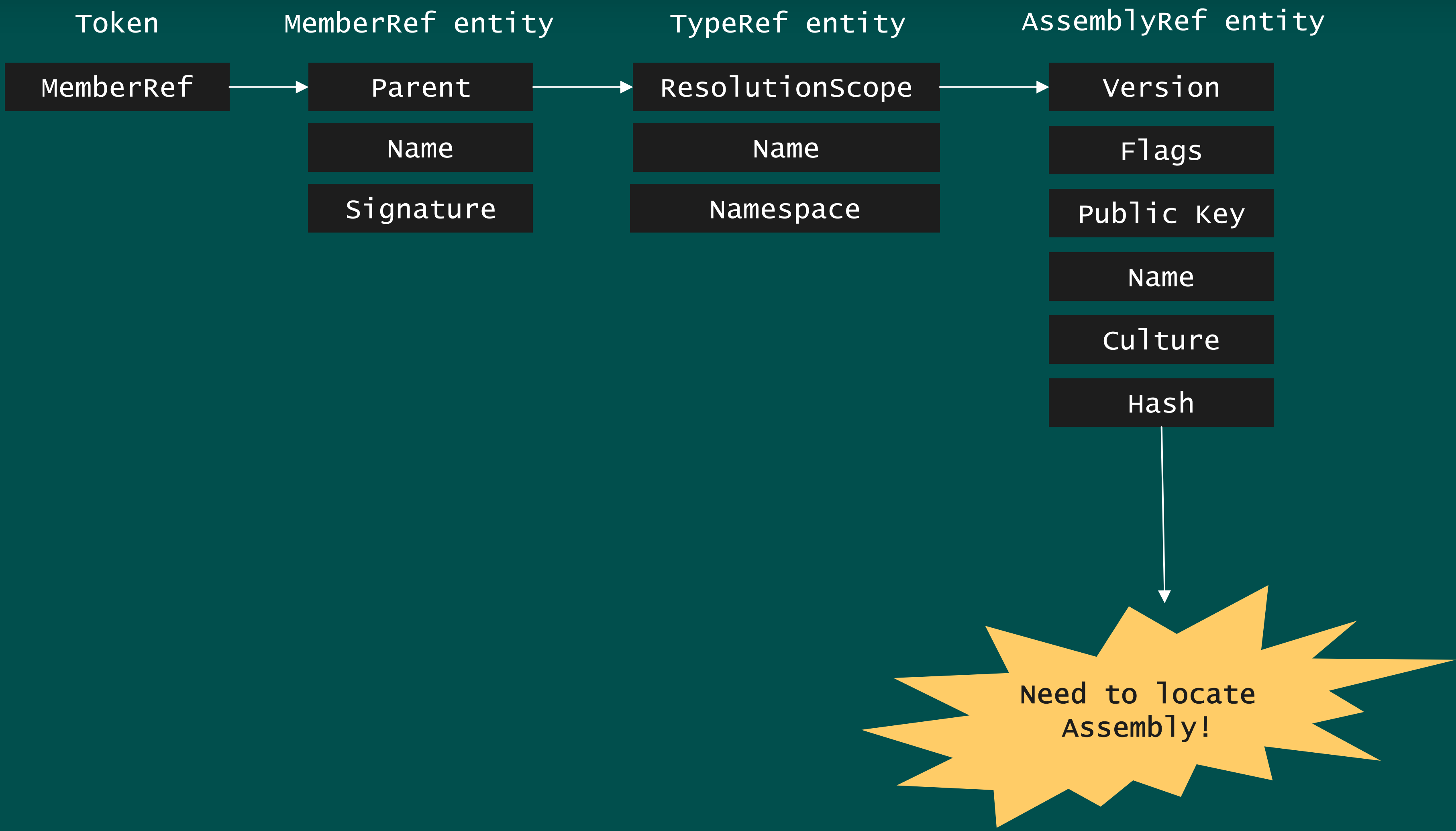
- Tokens are stored in a compressed format

```
private static uint CorSigUncompressData( BinaryReader reader ) {
  uint result;
  byte first = reader.ReadByte();
  if( (first & 0x80) == 0x00 ) {         // 0xxx xxxx == 1 byte case
    return (uint)first;
  }
  else if( (first & 0xC0) == 0x80 ) {    // 10xx xxxx == 2 byte case
    result = (uint)((first & 0x3F) << 8);
    result += reader.ReadByte();
  }
  else if( (first & 0xE0) == 0xC0 ) {    // 110x xxxx == 4 byte case
    result = (uint)((first & 0x1F) << 24);
    result += (uint)(reader.ReadByte() << 16);
    result += (uint)(reader.ReadByte() << 8);
    result += (uint)reader.ReadByte();
  }
  else
    throw new Exception( "Illegal compressed token: first byte = {0}", first );
  return result;
}
```

Resolving Ref tokens

- Ref tokens resolve to Def tokens in foreign scope
- Foreign scope is identified in metadata
    - AssemblyRef provides information
- Need to *resolve* foreign scope
    - Find assembly file on disk
    - Download on-demand assemblies
- Need to use Fusion API
    - Not doc'd in current form

Resolving MemberRef



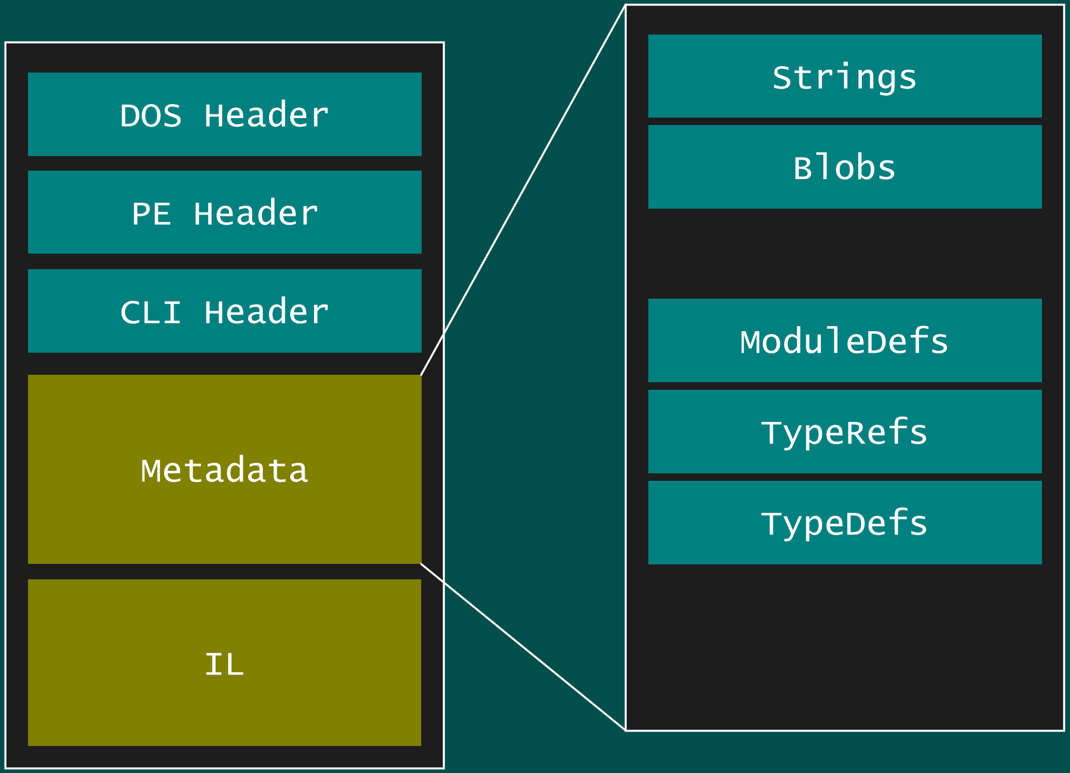
A better metadata API

- Combines benefits of managed, unmanaged API
  - Access to CIL
  - Access to tokens
  - System.Reflection ease of use
- Need to roll your own
  - Must understand metadata on-disk format

How is metadata stored?

- Metadata always stored in an assembly
  - Assemblies are usually single files
  - Single file is portable executable (PE) file
  - Custom CLI header contains metadata offset/size
- Metadata stored in tables / heaps
  - Tables indexed by row number
  - Heaps indexed by offset

How is metadata stored



Metadata on-disk format

- Documented in ECMA Partition II spec
  - CLI header format
  - Metadata directory format
  - Metadata table structures
- Metadata on-disk format is highly compressed
  - Optimized for size over speed

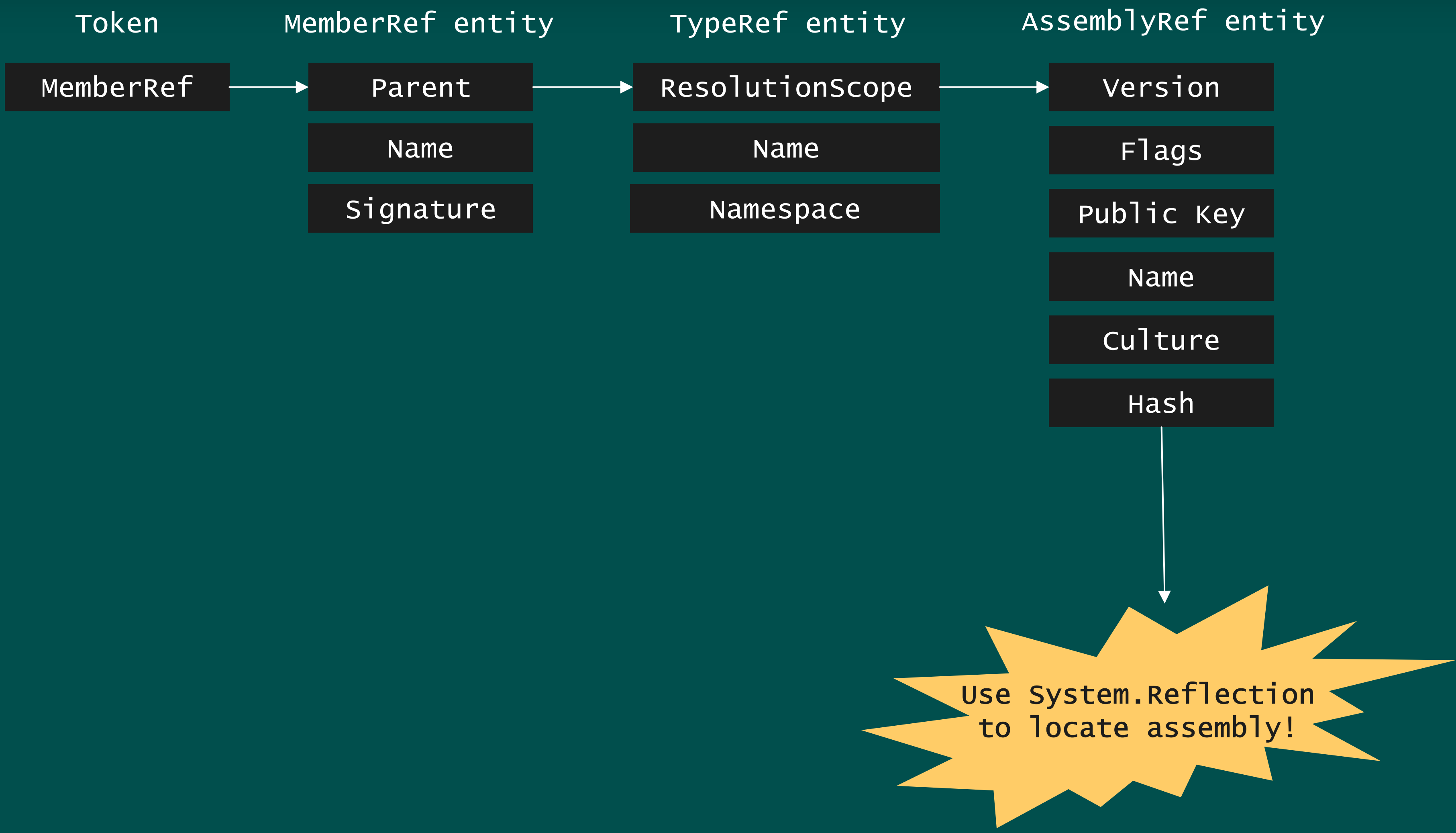
Resolving by name

- Resolving reference tokens done *by name*
  - Token elements contain a String token
- Resolving each can lead to different types
  - MemberRef -> MethodDef, PropertyDef, FieldDef, EventDef
  - TypeRef -> TypeDef
  - AssemblyRef -> AssemblyDef
  - ModuleRef -> ModuleDef

Resolving *Ref tokens

- Resolving *Ref token should yield corresponding *Def token
- Cannot do from System.Reflection
  - No access to CIL anyways ...
- Cannot do from IMetaDataImport
  - No access to Fusion API
- Solution requires using BOTH API's

Resolving MemberRefs



Resolving AssemblyRefs

- Problem:
  - Given an AssemblyRef
  - Return me the path to the assembly
- Solution:
  - Assembly.Load() solves this in managed API
  - Unmanaged API has no equivalent

Loading referenced assemblies

- Assembly.Load() used to locate assembly
  - AssemblyName is parameter
  - Uses internalcall nLoad method to do dirty work
  - Probably calls underlying Fusion API for resolution

```
public Assembly Load( AssemblyName assemblyRef );
```

References

- ECMA Partition II (Metadata) Specification
  - http://msdn.microsoft.com/net/ecma
  - Get PDF version!
- Unmanaged Metadata API
  - Tool Developer’s Guide\docs directory