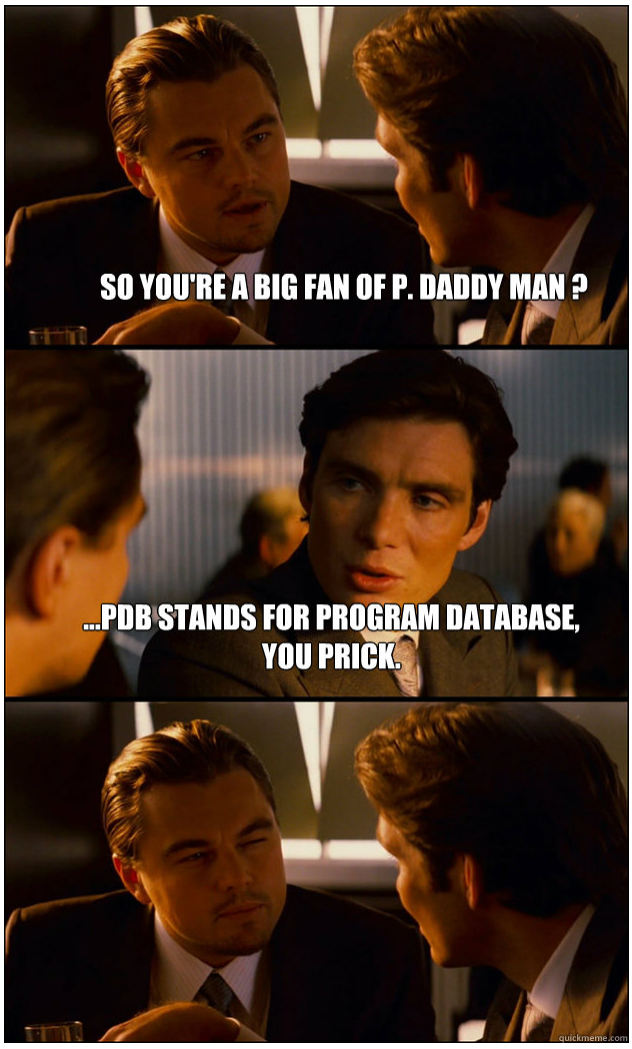
PDB ain't PDD: Let's introduce program database files

Written by:
Axel "0vercl0k" Souchet.

Twitter:
@0vercl0k



March 18, 2013

Contents

Part I
Introduction

After spending some cool time for the New Year's eve, I was back at home ready to start a new year full of system programming, exploit writing, bug-hunting, etc. So the 01/01, glad to be alive, I was finally back at my desk and I wanted to write a toy program using libogg[1] in order to play with it (who said fuzzing?). After reading the examples provided by the libogg team, I compiled a sort of libogg hello-world, fired up IDA to analyze it and at this very moment the impossible happened:

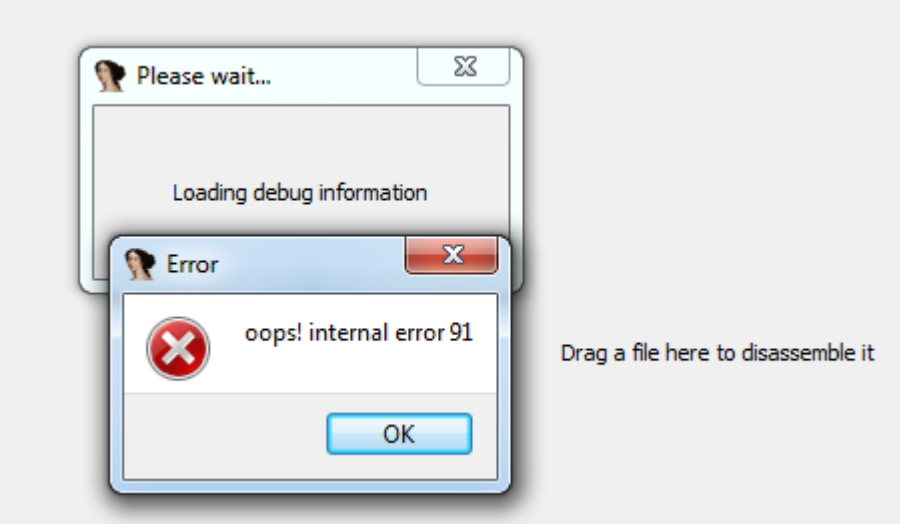


Figure 1: The impossible happened.

First, I thought this was just a random crash of IDA, maybe because of the IDB or something like that. Thus, I tried to reload what I will call `the magic binary` several times and it kept crashing IDA: every time a message box pops up and as I click on "OK", IDA closes. It was quite fun, because a few days ago I read nitr0us' paper on the bugs in IDA: he also got that type of message box error. However, as you will see in the following, my bug is really different from his, but I do not want to spoil the paper.

Next step was to check whether the crash occurred only on my machine, or if it was really something related to IDA. I powered on my reversing-dedicated VM, downloaded the last IDA demo version (version 6.3) and tried to load `the magical binary`. Unfortunately, I didn't see the message box :-(.

RIGHT, after doing some trollfaces I decided to investigate a little bit more the issue. So I started my debugger:

1. The message box is triggered by IDA!build_func_type2()

[1] http://www.xiph.org/ogg/

2. The faulting function is called from the IDA PDB management module (*pdb.plw* in the *modules* directory)

Let's give it another shot, I moved `the magical binary` and its PDB file on my VM and launched IDA on the binary. I finally got my message box :-).

So in this paper I am going to describe my journey into the PDB file format, and of course I will explain the reason of the IDA crash.

I hope you will enjoy the read, despite my English is quite bad... So you are cordially invited to take a seat and grab a cup of your favorite coffee:

GLADIATORS, LET'S GET STARTED!

Part II
Back to basics

1 What we already know

The usual way to analyze such a nasty bug is to dig in depth. In our case, the first step is to understand the PDB file format. First of all, I think you all know that these files store debug information, which are particularly handy for reverse-engineering:

- Structure definitions ;
- Function prototypes, argument types, argument names ;
- Class declarations ;
- Enumeration definitions ;
- etc.

You surely also know that debuggers can load these files and use them to annotate and enhance their listings. A common example is the Windows kernel whose Microsoft provides some PDB files via the symbol store. But I always thought these files were very simple, and actually I was quite wrong. The PDB format is a proprietary format developed by Microsoft, it exists several versions of the format (usually when a new version is out, it is shipped with the Visual C++ compiler) but we will focus on the version 7.0: this is the version currently generated by VC++.

In the next parts, we will dig a bit into those databases to get a brief overview of which information is stored, how they are stored and above all how we can extract them easily (or not).

2 The MS Debug Interface Access

The first thing I found when googling around was the DIA (Debug Interface Access) SDK provided by Microsoft. Microsoft had to create an API for the users that want to extract useful information from these files. The DIA binaries are shipped with the Redistribuable Microsoft Visual C++ 2008 (x86) package, and after the installation you can find them in the following directory (on my Win7 x64 machine):

```
C:\Program Files (x86)\Common Files\microsoft shared\VC>dir
19/02/2011  23:03           799 568 msdia100.dll
19/04/2011  09:47           670 032 msdia90.dll
```

With these files, IDA will be able to load debug information via the DIA library instead of using the dbghelp.dll way. In IDA the DIA interface is considered as the new-way of loading debugging information and the dbghelp.dll as the old. If it can't find the DIA binaries, IDA will smoothly fall back on the old-way.

For those who have already installed VS2010 on your computer, you should find the development files in the following directory and the official documentation here:

```
C:\Program Files (x86)\Microsoft Visual Studio 10.0\DIA SDK>dir
03/05/2012  17:39    <DIR>          bin
03/05/2012  17:39    <DIR>          idl
03/05/2012  17:39    <DIR>          include
03/05/2012  17:39    <DIR>          lib
03/05/2012  16:21    <DIR>          Samples

C:\Program Files (x86)\Microsoft Visual Studio 10.0\DIA SDK\include>dir
31/08/2009  01:34            94 165 cvconst.h
13/01/2010  00:40           281 626 dia2.h
31/08/2009  01:35             1 241 diacreate.h
```

2.1 First steps with DIA

Before going on the technical stuff, I just wanted to say a few words about the library: msdia90.dll. Its export-address table is nearly empty, only six functions are exported:

```
  Ordinal    RVA                 Symbol Name
  -------  ---------- ----------------------
   0x0001  0x00011C1D "DllCanUnloadNow"
   0x0002  0x0001293C "DllGetClassObject"
   0x0003  0x0001278B "DllRegisterServer"
   0x0004  0x000128A7 "DllUnregisterServer"
   0x0005  0x00012751 "VSDllRegisterServer"
   0x0006  0x0001276E "VSDllUnregisterServer"
```

This is because the API is COM based, long story short: COM is something that allows inter-object communication between COM clients and COM providers. The object can be in the same address-space (a .dll file for example) or not, but from the programmer's point of view it doesn't change anything: the code is the same (even if you know that behind the hood, there is an RPC mechanism like). Also, the interesting point is that you can query a COM object very easily from whatever language you like: you don't have to deal with the interfacing, COM is like a software bridge between your program and the provider.

2.1.1 Initializing the COM client

If you ever tried to interact with a COM provider, you should know that the first step is to call CoInitialize() to initialize the COM magic. Once you did that, you need to instantiate a COM object by calling CoCreateInstance() with the GUID identifying this object: in our case it will be *CLSID_DiaSource* defined in the DIA development files.

```
IDiaDataSource* instanciate_source(void)
{
    HRESULT hr;
    IDiaDataSource* src = NULL;

    hr = CoInitialize(NULL);
    if(FAILED(hr))
        return NULL;

    hr = CoCreateInstance(
        CLSID_DiaSource, // be sure to link with
           diaguids.lib
        NULL,
        CLSCTX_INPROC_SERVER, // the provider is an in
           -process one
        __uuidof(IDiaDataSource),
        (void **)&src
    );

    return src;
}
```

Listing 1: Initialize the COM stuff

2.1.2 Loading a PDB file

When the initialization of the COM client is done, we can start to play with the DIA API. We now have a pointer on a IDiaDataSource instance, and we can call the method loadDataFromPdb() with the path of our PDB file. If the call is a success, we have to open a session to gain access to the debugging information, to do so you call openSession(). If the session is opened, you can now query debugging information through this session.

```
IDiaSession* load_pdb_file(IDiaDataSource* src,
   pdb_path)
{
    HRESULT hr;
    IDiaSession* sess;

   hr = src->loadDataFromPdb(pdb_path);
    if(FAILED(hr))
        return NULL;

    hr = src->openSession(&sess);
```

```
        if(FAILED(hr))
            return NULL;

        return sess;
    }
```

Listing 2: Load a PDB file

2.1.3 Querying the database

Before going deeper, let's talk about database's organization. It really works like a tree, there is a root (given by the get_globalScope() method) and this root has children: each element of the tree is an instance of the IDiaSymbol interface. This interface has a really high number of methods, but only a small set is available to an IDiaSymbol instance. All the instances (during my tests sessions at least) define several methods to determine which type of information they hold, and to know how you are supposed to handle them, here are the two functions we will massively use:

- get_symTag() gives a value taken from the available tags listed in SymTagEnum. With this tag you know exactly what you are supposed to do with the symbol: for example if it is a SymTagFunctionType you may want to find the number of arguments the function has, the type of these arguments, the type of the return value, etc.
- get_symIndexId() is used to have a unique integer that identifies the symbol. You will see that when an IDiaSymbol instance is unnamed, this identifier is pretty useful.

In order to query the database, first get the root of the tree via get_globalScope() as I said earlier, then call findChildren() to handle each child.

```
IDiaSymbol* get_symbols_root(wchar_t* pdb_path)
{
    IDiaDataSource* src = instanciate_source();
    if(src == NULL)
        Fatal("instanciate_source\n");


    IDiaSession* sess = load_pdb_file(src, pdb_path);
    if(sess == NULL)
        Fatal("load_pdb_file\n");

    IDiaSymbol* root = NULL;
    HRESULT hr = sess->get_globalScope(&root);
    if(FAILED(hr))
        Fatal("get_globalScope\n");
```

```
    return root;
}
```

Listing 3: Get the root of the symbols tree

When you call the findChildren() method, the third argument can be used to find only one type of debugging information: it is a kind of a filter. If you don't want to filter the children the symbol has, you can just pass SymTagNull to have them all. Well, as you may notice, the API is a bit strange, and not really convenient for querying some specific stuff because you always have to go down, and up in the tree. But anyway, if you currently don't really get the API philosophy, next parts will help you a lot by showing some examples and illustrations.

```
void play_with_dia()
{
    IDiaEnumSymbols *enu = NULL;
    IDiaSymbol *sym = NULL, *global = NULL;
    HRESULT hr;
    ULONG celt = 0;

    global = get_symbols_root(PATH_PDB_FILE);

    hr = global->findChildren(
        SymTagNull, // We want all the children
        NULL,
        nsNone,
        &enu
    );

    if(FAILED(hr))
        Fatal("find_children\n");

    hr = enu->Next(1, &sym, &celt));
    while(SUCCEEDED(hr)  && celt == 1)
    {
        // Do something with sym
    }
}
```

Listing 4: Iterate through the children of the global scope

2.2 Playing with structures and functions

2.2.1 Extract the field types of a structure from a PDB

The first thing to do is to create a new project in which we define a structure, then compile it in order to have a PDB file to analyze. Let's take this definition:

```
typedef struct
{
    int* a;
    char b;
    int array_1[15];
    int array_2[2][15];
    int array_3[137][2][15];
} test_array_struct_t;
```

Listing 5: This is the structure we want to observe

As previously, we will reuse our open_pdb function to have a pointer on the global scope and will try to find the symbol named "test_array_struct_t" with a call to get_name(). Because we don't know (before testing) the tag of the symbol, we will print the tag of the symbol to see what it is.

```
int main()
{
    IDiaEnumSymbols *enu;
    IDiaSymbol *sym, *global;

    global = get_symbols_root(PATH_TEST);
    if(global == NULL)
        Fatal("get_symbols_root\n");

    HRESULT hr = global->findChildren(
        SymTagNull, // No filter, remember
        NULL,
        nsNone,
        &enu
    );

    if(FAILED(hr))
        Fatal("findChildren\n");

    ULONG celt = 0;
    while(SUCCEEDED(hr = enu->Next(1, &sym, &celt)) &&
        celt == 1)
    {
        BSTR name;
```

```
            DWORD tag;
            if(SUCCEEDED(sym->get_name(&name)))
            {
                if(StrCmpW(name, L"test_array_struct_t")
                   == 0)
                {
                    sym->get_symTag(&tag);
                    printf("Found test_array_struct_t, tag
                       : %d\n", tag);
                    printf("We're done.\n", tag);
                    return 0;
                }
            }
        }

        return 0;
    }
```

Listing 6: Finding the test_array_struct_t symbol

And after execution we get:

```
D:\TODO\tests_dia\Debug>parse_array_types.exe
Found test_array_struct_t, tag: 11
We're done.
```

As we can see in the SymTagEnum enumeration, it is a SymTagUDT. All the function names, structures names, enumerations names, etc. are SymTagUDT. But this symbol doesn't really describe the structure itself, because it is only a name. If we want to get more information about that structure, like the number of fields, or their types, we need to deal with the symbol tree. The following presents many examples to illustrate how the tree works when you are dealing with structures (actually the same philosophy will be applied for function arguments for example):

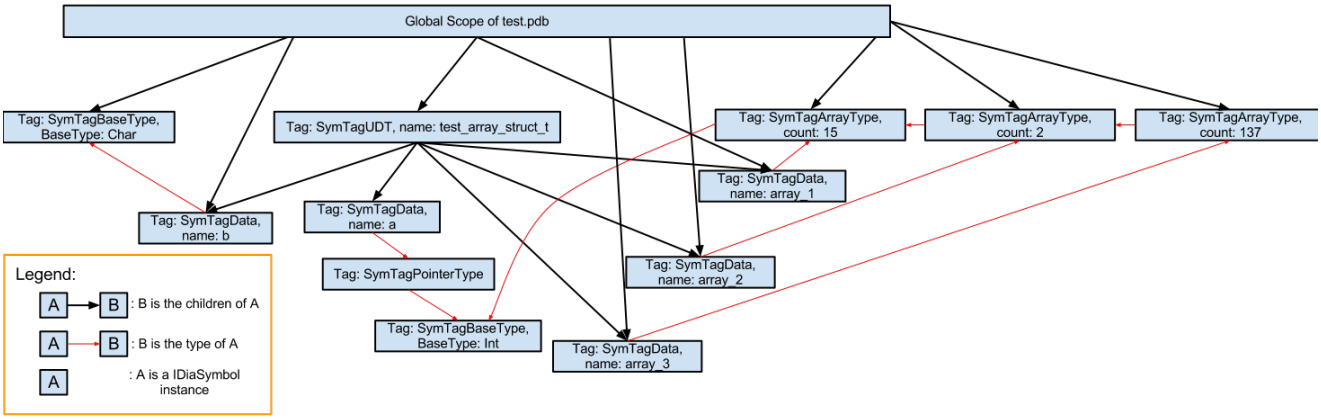


Figure 2: Global overview of the symbol tree.

Read carefully because what follows is a little bit tricky. Let's take a look at this tree, and more precisely at how you are supposed to read it:

1. As I said earlier, the global scope is the root of the tree ;
2. Then, we are looking for an instance named "test_array_struct_t" ;
3. Once you found your instance, we need to enumerate its children ;
4. That's where we will get the name of each field, each child is a field ;
5. Now if you want to get the type of each field, you have to call the get_type() method.

Another interesting detail you can see on Figure 2 is, that complex types are divided into several basic ones and those basic types make what we will call a `type-chain`.

Let's take some examples, first the field *a* from our structure (cf Figure 3). The type of *a* (in the tree) is defined as a SymTagPointer, that's the first element of the *type-chain*. Now, we must take the type of that pointer to know on what type of data it is pointing, and we see the last element (because we got a SymTagBaseType, the most basic type you can find) is an integer ; thus *a* is a *int**, it matches :-).

Figure 3: The *type-chain* of *a*.

Same thing for the field *array_1* (cf Figure 4): the first element of the chain is a SymTagArrayType that means, obviously, *array_1* is an array. To know the type of

data this array stores, we always have to call get_type(). Here our last element is a SymTagBaseType: *array_1* is an *int[15]*.

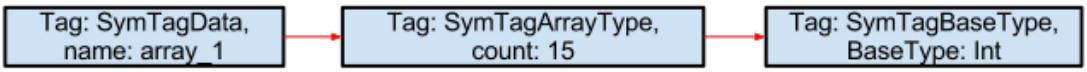


Figure 4: The *type-chain* of *array_1*.

For the last example (cf Figure 5), we are going to analyze the *type-chain* of *array_3* (if you get this one, it means you have understood!). As the others, the process is the same: we call get_type() until we find a SymTagBaseType symbol, and of course we keep building simultaneously our *type-chain*. The type of *array_3* is a SymTagArrayType, it is the first element of our chain, but it is not a SymTagBaseType so we keep going. Then it is another SymTagArrayType (that means it is an array with 2 dimensions), then again SymTagArrayType (it is now a three-dimensional array now), and finally we find the last element of the chain, the type of data stored by our three-dimensional array. *array_3* is a *int[137][2]15]*.

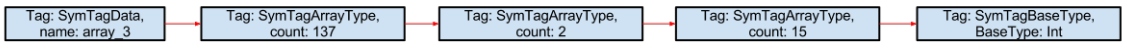


Figure 5: The type chain of *array_3*.

All these things are really important to understand before we talk about the evocated IDA message box. To implement what we did manually just before, you can define a recursive function that resolves the type of each field ; the recursion is finished when you get an error calling get_type().

```
void display_type(IDiaSymbol *sym, unsigned int lvl)
{
    IDiaSymbol *type;
    if(sym->get_type(&type) == S_OK)
    {
        for(unsigned int i = 0; i < lvl; ++i)
            printf(" ");

        DWORD id, tag;
        type->get_symIndexId(&id);
        type->get_symTag(&tag);

        printf("ID: %d, type: %s", id, tag_to_str(tag)
          );
```

```
            if(tag == SymTagArrayType)
            {
                DWORD count;
                type->get_count(&count);
                printf(", count: [%d]", count);
            }

            if(tag == SymTagBaseType)
            {
                DWORD basetype;
                type->get_baseType(&basetype);
                printf(", BaseType: %s", basetype_to_str(
                   basetype));
            }

            printf("\n");
            display_type(type, lvl + 4);
        }
    }
```

Listing 7: Resolve recursively an IDiaSymbol type

Let's be sure we are right:

```
D:\TODO\tests_dia\Debug>parse_array_types.exe
Found test_array_struct_t at id: 1250, type: SymTagUDT
- a - ID: 1251, type: SymTagData
    ID: 1252, type: SymTagPointerType
        ID: 1253, type: SymTagBaseType, BaseType: Int
[...]
- array_3 - ID: 1258, type: SymTagData
    ID: 1249, type: SymTagArrayType, count: [137]
        ID: 1248, type: SymTagArrayType, count: [2]
            ID: 1247, type: SymTagArrayType, count: [15]
                ID: 1253, type: SymTagBaseType, BaseType: Int

We're done.
```

2.2.2 Extract the arguments of a function

In this section, we are going to focus on function symbols. So we start a new project with this function definition:

```
    typedef struct
    {
        unsigned char field_1[10];
        bool field_2;
```

```
} structure_t;

bool testing_function(int arg1, int* arg2, char arg3,
   structure_t* arg4)
{
    printf("testing_function.\n");
    return true;
}
```

Listing 8: Function definition

The first step is to find the "testing_function" identifier in the symbol tree. As for the structure name, the function name is a SymTagUDT. Then, the arguments of the function are its children. If you want to resolve their types, you can even use the work we did before with the recursive type resolving. The only difference is that when you want to obtain information on the return type of the function, instead of calling findChildren() (that returns the arguments) you resolve directly the type of the symbol. Here is the global overview of our symbol tree that summaries what I just said (I chose not to draw the global scope neither the children of the "structure_t" symbol just to simplify the tree) :

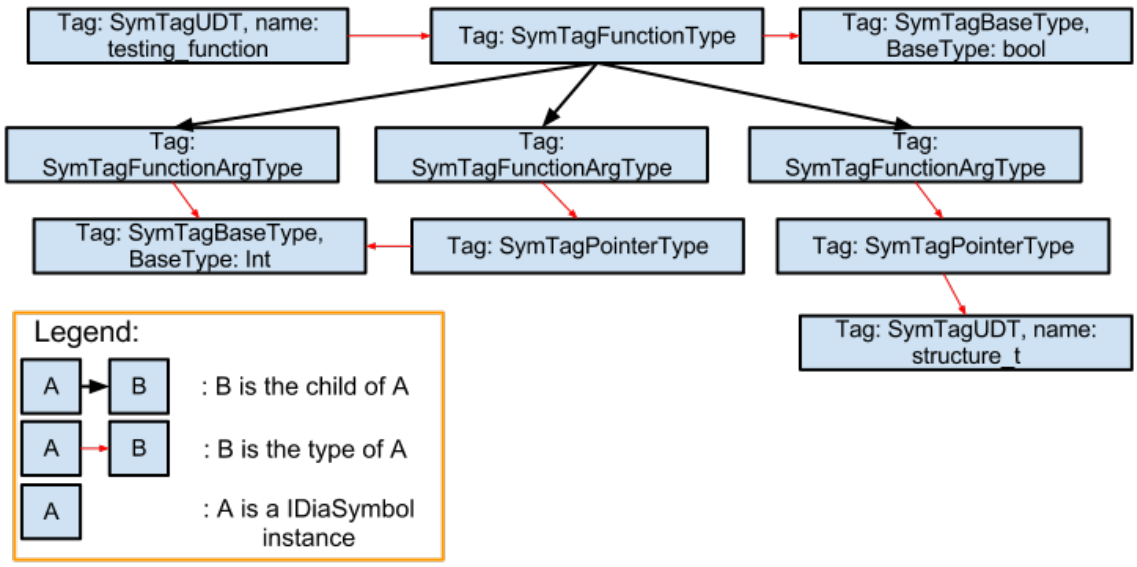


Figure 6: Overview of the symbol tree.

And here is the output of our PDB parser:

```
D:\TODO\tests_dia\Debug>parse_function_types.exe
tests_dia - parse_function_types
PDB loaded, enumerating types..
Found testing_function id: 1.
Getting the ret type of the function:
```

```
    ID: 2, type: SymTagFunctionType
        ID: 3, type: SymTagBaseType, BaseType: Bool

Type of testing_function: tag: SymTagFunctionType, id: 2
Getting the arguments of the functions:
    - SymTagFunctionArgType
        ID: 4, type: SymTagBaseType, BaseType: Int
    - SymTagFunctionArgType
        ID: 5, type: SymTagPointerType
            ID: 4, type: SymTagBaseType, BaseType: Int
    - SymTagFunctionArgType
        ID: 6, type: SymTagBaseType, BaseType: Char
    - SymTagFunctionArgType
        ID: 7, type: SymTagPointerType
            ID: 8, type: SymTagUDT, name: structure_t
```

3 PDB format, the hard way

The purpose of this section is to have a brief overview of the organization of a PDB database. You will see that knowing just a bit of how the data are organized in this type of file will be required to implement a clean bug trigger.

Fortunately the file format is documented by a chapter in "Undocumented Windows 2000 Secrets: A programmers cookbook" written by Sven Boris Schreiber. The author also gives the sources of a tool "win_pdbx" he developed to parse the database[2]. But of course this is not the only available tool, you can also check pdbparse written in Python and developed by mooyix. Note that I will speak only about the version 7.0 of the format.

3.1 It looks like a file-system

The first element of a PDB file is a header that describes several important things like the page size, the version of the format used or the total number of pages in the database:

- The file is subdivided into pages of the same size ;
- Pages don't need to be contiguous ;
- There are things stored thanks to this minimal file-system, those things are called "streams".

Once you have read this header you can find the position of another crucial structure called the "stream directory pointers", the header tells you in which page you can

[2]he also made an update of the tool to work with the version 7.0 of the format, the version we focus on

find this directory. This stream directory pointers is a table where you find which pages is used to store the different stream directories. A stream directory is another structure that will reference several streams. Here is an illustration to explain the basics:

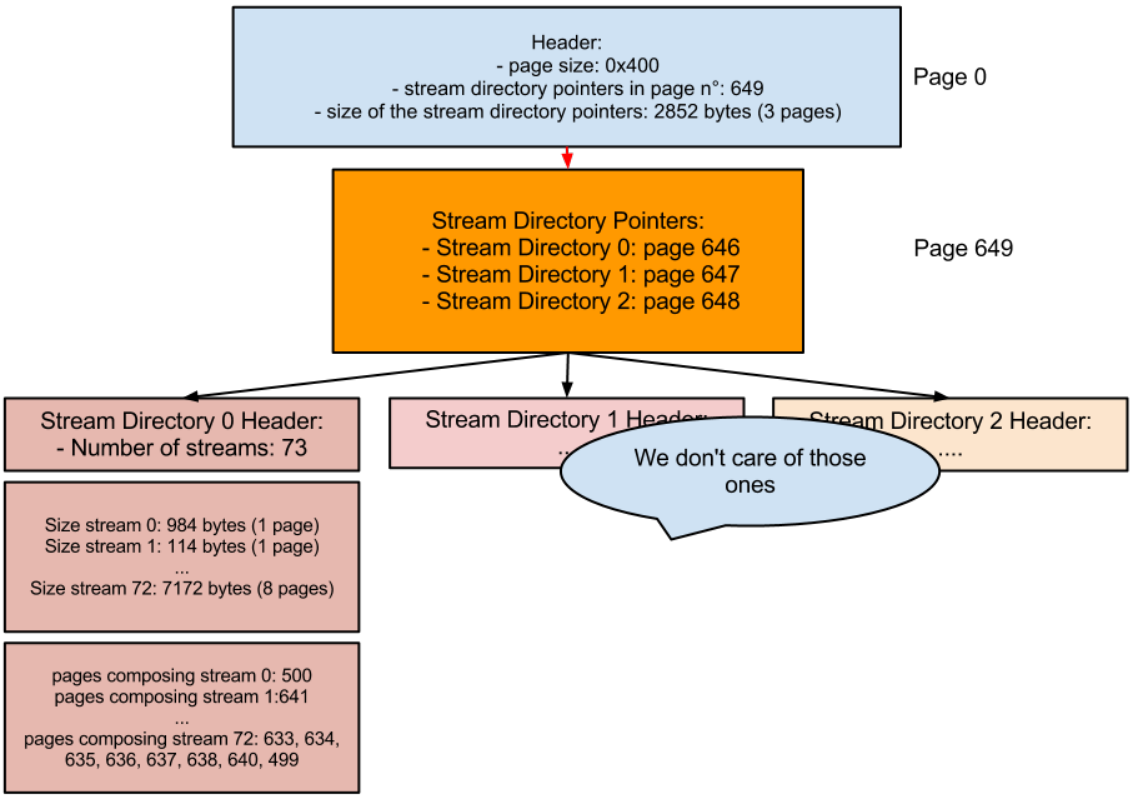


Figure 7: Brief overview of a PDB file.

Note that it is not really required to understand the whole file structure to tamper successfully our PDB file.

3.2 The Type Info stream

Among a lot of streams, there is a specific one that is really interesting for us. It is called the TPI stream (this is the third one for PDB v7), you can find a list of all defined types[3]. As we saw earlier there are different types, here is a list of the most important ones:

```
#define LF_ARRAY      0x00001503
#define LF_BITFIELD   0x00001205
#define LF_CLASS      0x00001504
#define LF_STRUCTURE  0x00001505
#define LF_UNION      0x00001506
```

[3]Check "tpi.py" from the pdbparse project for more details

```
#define LF_ENUM      0x00001507
#define LF_POINTER   0x00001002
#define LF_PROCEDURE 0x00001008
#define LF_MFUNCTION 0x00001009
#define LF_ARGLIST   0x00001201
#define LF_VTSHAPE   0x0000000A
#define LF_FIELDLIST 0x00001203
```

Listing 9: Different types of symbols

In a few words: if we want to tamper the PDB file, we will have to deal with the TPI stream. Don't hesitate to check the *parse_types_from_tpi_stream* project, and the stream itself in your hexeditor to understand briefly the structure.

Part III
Explaining the message box problem

4 Recon

4.1 Copy that

We saw in the introduction the problem seemed to come from the PDB module when it called the function IDA!build_func_type2(). The PDB module is a simple *DLL* that either uses the DbgHelp or the DIA API to obtain symbol information. As we saw earlier, the DbgHelp way is the *old* one and the DIA API is the new, check the log window of IDA to know which one is used.

```
[...]
PDB: using DIA dll "C:\Program Files (x86)\Common Files\Microsoft Shared\VC\msdia90.dll"
PDB: DIA interface version 9.0
Assuming __cdecl calling convention by default
PDB: loaded 0 types
PDB: total 2529 symbols loaded for C:\Users\0vercl0k\Desktop\user32.dll
[..]
```

After spending some time reverse-engineering the PDB module, I finally found the sources of the module in the leaked IDA SDK. As the module is coded in C++, there are a lot of structures / classes used: in one word it would have been a real pain in the ass to reverse-engineer it without this discovery :P. Then I compiled a debug build of the PDB module to ease the debugging.

4.2 Read the source luke

I'm going to explain the main actions realized by this module:

1. The module checks if it can use the DIA API, does some initialization job ;
2. Then the PDB file is loaded in the module in order to extract relevant information: types, function definitions, number of arguments, etc. ;
3. The problem appears when the module wants to handle the symbols, with the function PDB!handle_symbols(). The code isn't really trivial to read, because it uses recursive functions and a visitor/visited pattern[4] ;
4. Each symbol is handled by PDB!handle_symbol(). In this function the module tries to resolve recursively the type of the symbol like we did in the first part ;

[4] See the definition of PDB!for_all_children(), and the classes that inherit PDB!children_visitor_t (and more precisely the method "visit")

5. When it is a SymTagFunction (that's why we focused this type before), the function wants to know, among other things, the type of the return value. To accomplish that it calls the function PDB!really_convert_type() with the SymTagFunctionType (remember you call get_type()) symbol in argument ;

6. This function will after call PDB!get_symbol_type(), PDB!retrieve_type(), PDB!convert_type(), ..., and eventually it will call IDA!build_func_type() ;

7. The IDA!build_func_type() creates a structure describing a function type: the type of the return-value, the number of arguments and the type of each of them.

   Then this type is added to a global list where all the details of all types are stored (PDB!typemap). That way, if a function has the same type, the resolver will see that its type has already been added to the PDB!typemap list and stop the type resolving ;

8. When IDA!build_func_type() finds something weird with a resolved type, it shows the message box we saw earlier. The pseudo-code of IDA!build_func_type() is:

```
int build_func_type(qtype *ftype, qtype *ffields,
   func_type_info_t *fi)
{
    return build_func_type2(idati, ftype, ffields,
        fi);
}

//Note: func_type_info_t is a type used to store
// *all* the available information, like the
   number
// of arguments, the type of the arguments, the
// type of the return value, etc.
int build_func_type2(int a1,
   _qstring_unsigned_char *ftype,
   _qstring_unsigned_char *ffields,
   func_type_info_t *fi)
{
  char *v25;

  if ( fi->rettype.body.n )
  {
    v25 = fi->rettype.body.array;
    if ( !v25 )
      goto LABEL_38;
  }
```

```
//[...]

  if ( (*v25 & TYPE_BASE_MASK) == BT_ARRAY )
    goto LABEL_40;

//[...]

LABEL_40:
    if ( under_debugger )
      __debugbreak();

    interr(91); // That calls MessageBox and
       displays the error 91
  }
```

Listing 10: *IDA!build_func_type* pseudo-code

To sum-up, IDA shows me the message box error because my `magical PDB` file is a bit weird: the type of the return value of some functions were a two-dimensional array, but of course it isn't possible in C / C++, that's the reason why IDA shows an error[5].

I must admit I was a bit disappointed because of two things:

- No memory corruption bug is involved ;
- I couldn't reproduce the generation of the `magical PDB` file in my compiler (maybe that's a good thing, because I doubt that debugging VC++ 2010 is really fun :P).

My wild guess is: this is a bug of a VC++'s component.

5 Building a clean trigger

Let's create a clean, minimal, `magical PDB` trigger). Reminder: all the types are stored in a specific stream called "the TPI stream". We will have to modify some bytes in this stream.

The first thing I did was to use the win_pdbx tool to extract all the streams, then I only kept the stream number 2 (the third): the TPI stream. Then I read several sources like the win_pdbx and the pdbparse ones to see how I could enumerate each type in order to understand better how I can tamper the database. As the types aren't name identified, we can use their arguments number to identify them[6]. Thus, we define this function in our trigger project:

[5] I don't understand why it doesn't simply quit the PDB loading and still runs IDA without symbol information instead of killing the IDA instance.

[6] Feel free to check the "parse_types_from_tpi_stream" project (link at the end) to understand how to parse this stream.

```
int BOOM[1337][1338] = {0};
void crashing_function(int a, int b, int c, int d, int
    e, int f, int g, int h, int i, int j, int k, int l
   )
{
    printf("crashing_function.\n");
}
```

Listing 11: Minimal clean trigger!

Keep in mind our purpose: we want to tamper the PDB file in order to have the type of the return value of trigger!crashing_function() pointing on the type of BOOM: a 2-dimensions array just to trigger the message box. Launching my tool to enumerate the different types in the TPI stream returns me something like that:

```
D:\TODO\trigger\Debug\experimentation\original>parse_types_from_tpi_stream.exe
ID: 1527, Type: 00001008, Size: 000e, Offset: 0001f870 -- LF_PROCEDURE
RetType:00000003 ; calltype: 00 ; parmcount: 000c ; arglist: 00001526
ID: 1528, Type: 00001503, Size: 000e, Offset: 0001f880 -- LF_ARRAY
ID: 1529, Type: 00001503, Size: 0012, Offset: 0001f890 -- LF_ARRAY
EOF.
```

We can see that the type 0x1527 is the trigger!crashing_function() one: it is the only symbol with 12 arguments (it is also the only *LF_ARRAY* one).

Let's doing some voodoo magic on the PDB file. Open your favorite hex-editor and start looking for the first occurrence of the string "crashing_function". On my side I got this:

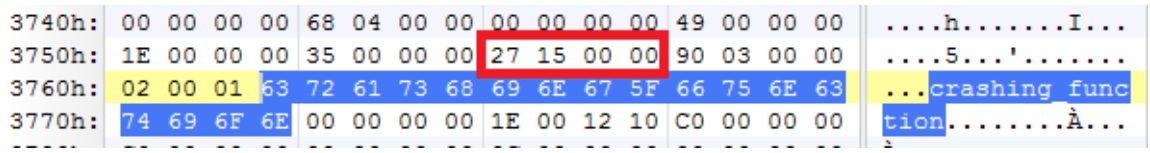


Figure 8: First occurrence of "crashing_function" in the trigger.pdb.

As you may have noticed, the highlighted DWORD (cf Figure 8) is the ID of the crashing_function's type we saw in the previous dump. That means that if we modify this DWORD and we put the ID of 2-dimensions array type we will get our clean and minimal trigger. In the previous dump we saw that the type 0x1528 is an array type, so let's try to update the DWORD with this ID:

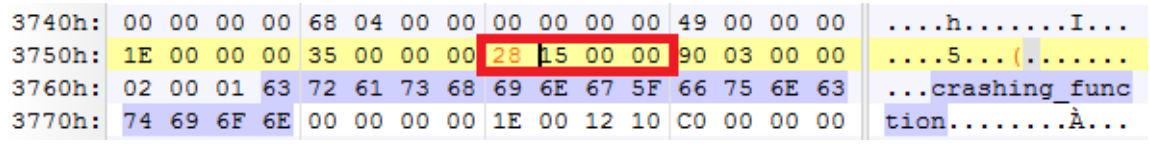


Figure 9: The type of *trigger!crashing_function()* (a SymTagFunction) is now an array.

And to be sure, we can even use the DIA API to parse the type of the trigger!crashing_function() before the modification:

```
D:\TODO\trigger\Debug\experimentation\original>parse_function_types.exe
tests_dia - parse_function_types
Initializing DIA..
PDB loaded, enumerating types..
Found testing_function id: 1.
Getting the ret type of the function:
    ID: 2, type: SymTagFunctionType
        ID: 3, type: SymTagBaseType, BaseType: Void

[...]
```

And after the modification:

```
D:\TODO\trigger\Debug\experimentation\boom>parse_function_types.exe
tests_dia - parse_function_types
Initializing DIA..
PDB loaded, enumerating types..
Found testing_function id: 1.
Getting the ret type of the function:
    ID: 2, type: SymTagArrayType, count: [1337]
        ID: 3, type: SymTagArrayType, count: [1338]
            ID: 4, type: SymTagBaseType, BaseType: Int

[...]
```

We finally got our clean and minimal trigger! With this trigger, I can crash IDA 6.1, IDA 6.3 but not IDA 6.4 (I don't have a license, so I couldn't debug the problem ; if you have some precisions I will be happy to merge them into this section ;-)).

5.1 We want moar crashes!

After finishing the previous trigger, I wasn't really satisfied because the last version of IDA didn't crashed. So I came up with a fun idea: What if we have a symbol S and its type is S?

Clearly in the way we implemented our type resolving in the first part, our algorithm will run infinitely (until the stack explodes by the recursive calls) ; but I

was curious to know if I could crash IDA with that tips. By the way, the exercise of making the modification in the PDB file is left to the reader (though if you diff the original trigger.exe and the recursive one, you will quickly spot how I did it).

```
002a373c 64b88584 msdia100!GetData::getArrayData+0x2d
002a3748 64b92740 msdia100!GetTypeData::disp_LF_ARRAY+0x14
002a3750 64b8745c msdia100!TypeDispatcher::TypeDispatch+0x240
002a37b0 64b8787d msdia100!GetData::getTypeData+0x17c
002a3a54 64b88584 msdia100!GetData::getArrayData+0x4d
002a3a60 64b92740 msdia100!GetTypeData::disp_LF_ARRAY+0x14
002a3a68 64b8745c msdia100!TypeDispatcher::TypeDispatch+0x240
002a3ac8 64b8787d msdia100!GetData::getTypeData+0x17c
002a3d6c 64b88584 msdia100!GetData::getArrayData+0x4d
002a3d78 64b92740 msdia100!GetTypeData::disp_LF_ARRAY+0x14
002a3d80 64b8745c msdia100!TypeDispatcher::TypeDispatch+0x240
002a3de0 64b8787d msdia100!GetData::getTypeData+0x17c
002a4084 64b88584 msdia100!GetData::getArrayData+0x4d
002a4090 64b92740 msdia100!GetTypeData::disp_LF_ARRAY+0x14
002a4098 64b8745c msdia100!TypeDispatcher::TypeDispatch+0x240
002a40f8 64b8787d msdia100!GetData::getTypeData+0x17c
002a439c 64b88584 msdia100!GetData::getArrayData+0x4d
002a43a8 64b92740 msdia100!GetTypeData::disp_LF_ARRAY+0x14
002a43b0 64b8745c msdia100!TypeDispatcher::TypeDispatch+0x240
```

Figure 10: Let's crash our program.

With that neat trick I can crash IDA 6.1, 6.3 and finally IDA 6.4!
Then I started OllyDBG 2 to add a custom symbol directory like that:

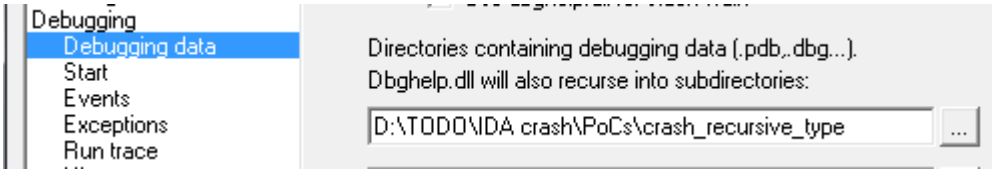


Figure 11: Configure a symbol directory in OllyDBG 2.

and ..BOOM, again :-) ![7].

[7]Don't forget it needs user interaction to configure the symbol directory ; though I wouldn't be surprise it exists a trick to avoid this user interaction: if you succeed, feel free to share the details, I will add them!)

```
(efc.fa0): Stack overflow - code c00000fd (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Debugger\OllyDbg\odbg200\DBGHELP.DLL -
eax=c04a8b03 ebx=00093668 ecx=00093790 edx=00000003 esi=00093668 edi=0009333c
eip=64e34858 esp=00092f10 ebp=00093320 iopl=0         nv up ei ng nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00210286
DBGHELP!MiniDumpReadDumpStream+0x58d28:
64e34858 83a5f8fbffff00  and     dword ptr [ebp-408h],0 ss:002b:00092f18=00000000
0:000> k
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
00093320 64e36fbf DBGHELP!MiniDumpReadDumpStream+0x58d28
00093330 64e37e03 DBGHELP!MiniDumpReadDumpStream+0x5b48f
0009335c 64e38478 DBGHELP!MiniDumpReadDumpStream+0x5c2d3
000935d8 64e38818 DBGHELP!MiniDumpReadDumpStream+0x5c948
000935e8 64e3a7dc DBGHELP!MiniDumpReadDumpStream+0x5cce8
000935f4 64e369ba DBGHELP!MiniDumpReadDumpStream+0x5ecac
0009363c 64e3849d DBGHELP!MiniDumpReadDumpStream+0x5ae8a
000938bc 64e38818 DBGHELP!MiniDumpReadDumpStream+0x5c96d
000938cc 64e3a7dc DBGHELP!MiniDumpReadDumpStream+0x5cce8
000938d8 64e369ba DBGHELP!MiniDumpReadDumpStream+0x5ecac
00093920 64e3849d DBGHELP!MiniDumpReadDumpStream+0x5ae8a
00093ba0 64e38818 DBGHELP!MiniDumpReadDumpStream+0x5c96d
00093bb0 64e3a7dc DBGHELP!MiniDumpReadDumpStream+0x5cce8
00093bbc 64e369ba DBGHELP!MiniDumpReadDumpStream+0x5ecac
00093c04 64e3849d DBGHELP!MiniDumpReadDumpStream+0x5ae8a
00093e84 64e38818 DBGHELP!MiniDumpReadDumpStream+0x5c96d
```

Figure 12: B.O.O.O.O.M.

6 Debug information inside the binary

Keep in mind I wanted to have an anti-IDA, so I tried to embed the debug information inside the PE itself. After googling a lot around the subject I saw that a lot of years ago, it was possible to embed directly the debug information into the PE, more precisely in the DEBUG_DIRECTORY. @gentilkiwi kindly uploaded me a really old version of MS VC++ to try this feature (the new version of VC++ doesn't support it anymore). I thought it was a cool idea to have a "real" maybe usable anti-IDA, but I didn't really succeed at this part. If you load the file into IDA, the codeview information embedded seem to be handled by the DBG module instead of calling directly the PDB module (like earlier). So to trigger the message box, after loading the file, you have to click on *File* and then *Load PDB file* and BOOM. That's not that great :-([8].

[8]Note that I didn't really made a lot of research to bypass this user interaction ; I was a bit tired to work on these nasty PDB files, so if you get cool results feel free to shoot me!

Part IV
This is the end

That's it guys, I hope you learned a lot of things (because I really did!) even if the purpose of the research is pretty useless. As usual, you will find several dirty codes (you have been warned :-P) I've produced to play with many different subjects.

If you are still interested in the subject, here is a nice list of links:

- http://undocumented.rawol.com/
- http://code.google.com/p/pdbparse/
- http://waleedassar.blogspot.fr/2012/06/ida-pro-and-codeview-debug-info-bug.html
- http://pdbdump.sourceforge.net/pdbleak.html
- https://www.mandiant.com/blog/exploring-symbol-type-information-pdbxtract/
- http://www.informit.com/articles/article.aspx?p=22685
- http://support.microsoft.com/default.aspx?scid=kb;en-us;Q121366
- http://www.godevtool.com/Other/pdb.htm
- http://www.debuginfo.com/articles/debuginfomatch.html
- http://moyix.blogspot.fr/2007/10/types-stream.html

High five for the reviewers and the guys who helped me: Jiss, @joancalvet, @Ivanlef0u and @gentilkiwi.