Hi GDB, this is Python.

0vercl0k aka Souchet Axel.
Email: 0vercl0k@tuxfamily.org
Twitter: @0vercl0k

Contents

Part I
Introduction

Since the version 7 of the *Gnu Debugger*, I'm pretty sure you already know that, but the python interpreter is accessible from GDB. The person behind this work is Tom Tromey: that guy made python available inside GDB, thanks! If you are interested by the implementation of the API, you should read GDB's sources, and take a look at the *gdb/python* directory (also consultable online here). There are a lot of nice functions you can use to extend your debugger, they are all documented here: GDB Python-API. Indeed, with this API you can do things like:

- Define new (prefixed) commands
- Create pretty-printing modules
- Manipulate breakpoints
- Access the stack frames
- Read/Write/Search directly in process memory
- A lot more!

As I said earlier, you can execute python code easily from the gdb shell:

```
# for oneliner
gdb$ python print 42
42

# for larger code
gdb$ python
> a = 'SGVsbG8sIFdvcmxkIQ=='.decode('base64')
> print a
CTRL+D
Hello, World!
```

Actually, it is very convenient during exploitation or real debugging sessions to create GDB scripts that can assist you: for example if you need to search for a specific thing in the stack, or in the binary. Here is my story: two weeks ago, I was doing a level of the sm0k's wargame and to complete my exploit I was supposed to find a specific pointer in the stack. This pointer was very important, because it allowed me to bypass the ASLR making my exploit completely reliable. When you are researching things like that in GDB it's hard: you dump the stack and you pray your eyes will recognize a libc pointer in this huge amount of data.

With this article and this little problem, I will try to give you a global view of what the GDB python API offers us, what you can build with it. By the way, the stuff I will expose in next parts have been only tested with the GDB version 7.4.1. Our futur GDB command will work like that:

1. Retrieve the two arguments given to the command: the first one is a CPU register or an address and the second one is the number of pointers to display
2. If the first argument is a CPU register we need to get its content
3. Read the memory pointed by the address
4. Check if the pointer is interesting or not
5. Display the whole information

All those things will be discussed in next parts of this article.

Part II
”Dump Pointers with Symbols” like

1 The WinDbg’s dps command

There are a lot of useful commands in WinDbg, and one of them is *dps*. This function displays only one DWORD per line, and if the pointer is a known symbol, it displays the symbol name, almost exactly what I wanted. Here is what you see if you try to dump the Windows SSDT structure:

```
lkd> dps nt!KeServiceDescriptorTable l4
8296b9c0  828726f0 nt!KiServiceTable
8296b9c4  00000000
8296b9c8  00000191
8296b9cc  82872d38 nt!KiArgumentTable
```

The previous dump is organized in three columns, the first one is the address of the pointer, the second column is the value contained at the address and the last column (the most interesting) is the symbol. According to this specific dump, we can know that the function *nt!KiServiceTable* starts at address 0x828726f0 ; but you cannot see that easily with a classic dump like this one:

```
lkd> dd nt!KeServiceDescriptorTable l4
8296b9c0  828726f0 00000000 00000191 82872d38
```

It’s the same thing in GDB, you have a huge dump with addresses so if you’re looking for something, it is really hard to spot something that meets your requirements.

Fortunately, you can create a very similar function with the python API.

2 Defining a new command

If you want to define a new command accessible from the GDB shell, you must create a python class that inherits from *gdb.command*. Next, you must call the constructor of *gdb.command* with the name of the command and other arguments detailed precisely here: gdb.commad.__init__. Then, you define a method that will be called when you type the name of your command in the GDB shell ; its name must be *invoke*.

We can now write a little HelloWorld to be sure we have well understood the basics:

```
class HelloWorld (gdb.Command):
"""Greet the whole world."""
  def __init__ (self):
    super (HelloWorld, self).__init__ ("hello-world", gdb.
        COMMAND_OBSCURE)
  def invoke (self, args, from_tty):
    print "Hello, World!"
HelloWorld()
```

Listing 1: Hello World command from the official documentation

3 Dump the stack

3.1 Get the content of a CPU register

We need to know how to get the content of a CPU register ; the only way to do that is to use the *gdb.parse_and_eval()* method. It takes a string in parameter and tries to evaluate it in order to return a *gdb.Value* object. Actually, you can even make computations, make dereferences, etc. I decided to use it only to get the content of a register, just like that:

```
gdb$ python print gdb.parse_and_eval('$eax')
0x6e1b68
gdb$ python print gdb.parse_and_eval('$ax')
0x1b68
gdb$ python print gdb.parse_and_eval('$al')
0x68
```

3.2 Dereference pointers

As you have seen in the previous part, the first objective of our future command is to be able to display data contained in a specific memory area: pointed by a CPU register or directly by its address.To do that, we can use the method *gdb.Value.dereference()*, it returns a new *gdb.Value* object containing the data. The thing is you cannot call the *gdb.Value.dereference()* method from any *gdb.Value* object, the object must be considered as a pointer internally by the API. We can see that directly in the C sources of the API:

```
/* Given a value of a pointer type, apply the C unary * operator to
   it.  */
struct value *
value_ind (struct value *arg1)
{
  struct type *base_type;
  struct value *arg2;
//[...]

  base_type = check_typedef (value_type (arg1));

  if (VALUE_LVAL (arg1) == lval_computed)
    {
//[...]
    }

  if (TYPE_CODE (base_type) == TYPE_CODE_PTR)
    {
//[...]
    }

  error (_("Attempt to take contents of a non-pointer value."));
  return 0;     /* For lint -- never reached.  */
}

/* Given a value of a pointer type, apply the C unary * operator to it.
     */
static PyObject *
valpy_dereference (PyObject *self, PyObject *args)
{
  struct value *res_val = NULL;    /* Initialize to appease gcc warning.
       */
  volatile struct gdb_exception except;
  TRY_CATCH (except, RETURN_MASK_ALL)
    {

      res_val = value_ind (((value_object *) self)->value);
    }
  GDB_PY_HANDLE_EXCEPTION (except);

  return value_to_value_object (res_val); // returns a new Value object
      with the content dereferenced
}
```

To have a *TYPE_CODE_PTR* object, we have to use the *gdb.Value.cast()* method, but we must have a *gdb.Value* instance that represents a pointer somewhere in order to cast correctly our instance.

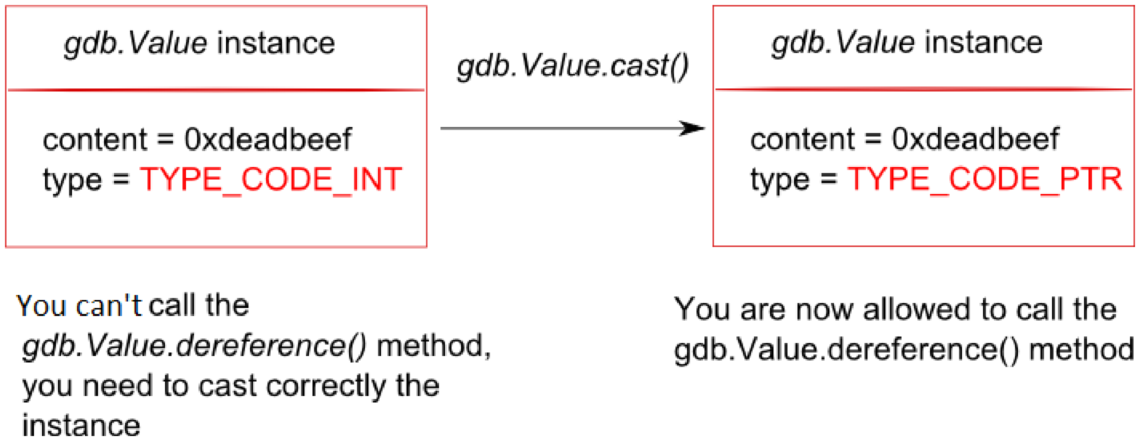

gdb.Value instance
content = 0xdeadbeef
type = TYPE_CODE_INT
gdb.Value.cast()
gdb.Value instance
content = 0xdeadbeef
type = TYPE_CODE_PTR
You can't call the gdb.Value.dereference() method, you need to cast correctly the instance
You are now allowed to call the gdb.Value.dereference() method

Figure 1: *gdb.Value.cast()* illustration

Actually, we call the magic method *gdb.lookup_type()* to obtain a *gdb.Value* instance that holds an integer. Next we call the *gdb.Value.pointer()* method to obtain an integer pointer. Then, we give this instance in argument to *gdb.cast()* to cast correctly our object. After all those operations, we can use the *dereference()* function. Check the code below to see those different steps executed:

```
gdb$ x/dwx $esp
0xbffff7ac:     0xb7e4ee46
gdb$ python
> int_pointer_type = gdb.lookup_type('int').pointer()
> stack_address = gdb.Value(0xbffff7ac)
> stack_address_pointer = stack_address.cast(int_pointer_type)
> content = long(stack_address_pointer.dereference())
> print hex(content & 0xffffffff)
0xb7e4ee46L
```

Perfect, even if this way seems a little weird to use, it's in fact a very convenient one: you can ensure the x64 portability easily thanks to the *long* type: when you are debugging an x86 process inside GDB the size of the *long* type is 4 bytes, and when you debug an x64 process the size is 8 bytes. In addition, it raises a *gdb.MemoryError* exception if there isn't any memory where you're trying to look. You will see in the tips section this way wasn't exactly the only one, but I just wanted to play with *gdb.Value*. Now we can declare a small function able to dereference a pointer via its address:

```python
def deref_long_from_addr(addr):
    '''
    Get the value pointed by addr
    '''
    p_long = gdb.lookup_type('long').pointer()
    val = gdb.Value(addr).cast(p_long).dereference()
    return long(val) & 0xffffffff
```

Listing 2: deref_long_from_addr() function

4 Interesting pointers or not ?

Now we know how to dump data pointed directly by their addresses or by a CPU register, we have to deal with those pointers in order to see if they are interesting or not. They are interesting if they point in a shared-library like the libc (a pointer ideal to leak for defeating the ASLR) or if they are ASCII string pointers. Yeah, I know, it's not really symbols like in the *dps* command, but anyway it's relevant to see those information when you dump data. I've created the python command in a very simple way:

- The pointers references a location where the memory is accessible:
  - It's a pointer on an ASCII string of 3 characters minimum, so we display the string
  - It's not a string, so we try to disassemble one instruction at this address and we display it

4.1 String pointers ?

There is a function in *gdb.Value* class doing exactly what we want: it's the *gdb.Value.string()* method. This method returns the string pointed to by the *gdb.Value* instance or generates a *gdb.MemoryError* exception in the memory isn't accessible. As you have seen in previous parts, to do that we have to cast correctly our *gdb.Value* instance otherwise we can't call the *gdb.Value.string()* method.

```
gdb$ python print gdb.Value(0xdeadbeef).string()
Traceback (most recent call last):
  File "<string>", line 1, in <module>
gdb.error: Trying to read string with inappropriate type ‘long’.
```

We just have to look for the *char* type, and to retrieve a pointer as we did for the *long* type:

```
gdb$ python
>char_pointer_type = gdb.lookup_type('char').pointer()
>address_str = gdb.Value(0xdeadbeef)
>string_pointer = address_str.cast(char_pointer_type)
>print string_pointer.string()
Traceback (most recent call last):
  File "<string>", line 1, in <module>
gdb.MemoryError: Address 0xdeadbeef out of bounds
```

This function is very useful for us, it just does exactly what we wanted: check if a specific area of memory contains an ASCII NULL terminated string (yeah, it only works with ASCII string, I guess you will have to define your own function if you want to recognize UNICODE string).

```
gdb$ x/s 0x7ffff775861b
0x7ffff775861b:  "/bin/sh"
gdb$ python
>char_pointer_type = gdb.lookup_type('char').pointer()
>address_str = gdb.Value(0x7ffff775861b)
>address_str_ptr = address_str.cast(char_pointer_type)
>print address_str_ptr.string()
/bin/sh
```

4.2 Disassemble ?

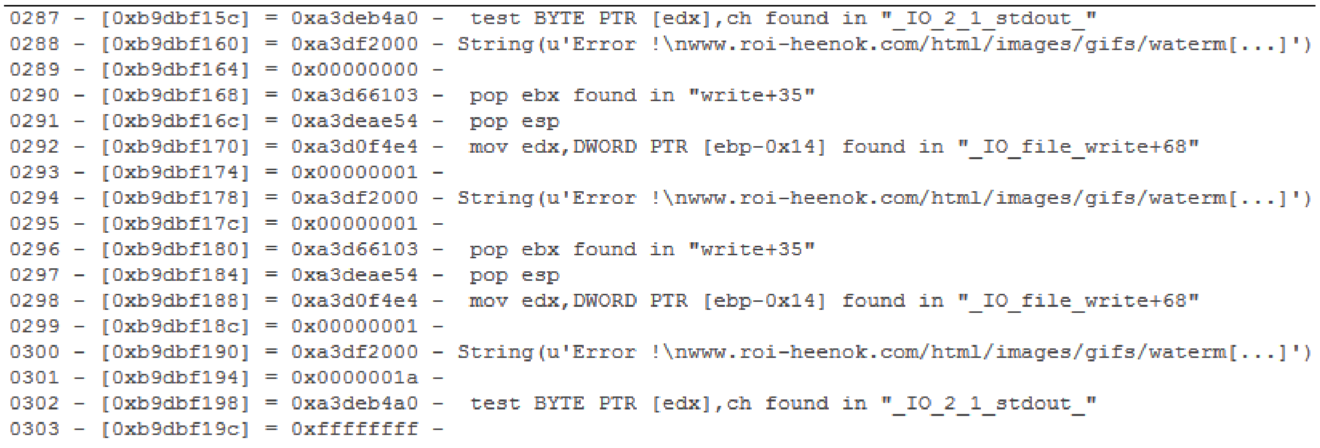
In this part, we need to be a bit more crafty because the python API doesn't give us the possibility to call the internal disassembler of GDB. We only know we can disassemble some code via the *x/i* GDB command. Fortunately, there is a useful function called *gdb.execute()* and this function is able to execute a GDB command. In addition, this command allows us to retrieve the result of the command in a string: we just need to parse the output and to extract exactly the data we want, like the instruction disassembled and the name of the function where the instruction is. Here is a little example:

```
gdb$ python print gdb.execute('x/i %#.8x' % 0x420010, to_string=True).split(':')
['=> 0x420010 <main>', '\tpush   r15\n']
```

Remember this function can be really powerful, and it's sometimes the only way to accomplish the thing you want. I've also used this function to obtain information concerning the memory mapping of the process. I use the output of the *info files* GDB command in order to get the addresses of the shared-libraries sections, and the binary sections.

5 Put it all together!

Here we are, I believe you have now all the materials to build this little command. You will find mine on my github account, and here is an example on the sm0k's wargame:



```
0287 - [0xb9dbf15c] = 0xa3deb4a0 -  test BYTE PTR [edx],ch found in "_IO_2_1_stdout_"
0288 - [0xb9dbf160] = 0xa3df2000 - String(u'Error !\nwww.roi-heenok.com/html/images/gifs/waterm[...]')
0289 - [0xb9dbf164] = 0x00000000 -
0290 - [0xb9dbf168] = 0xa3d66103 -  pop ebx found in "write+35"
0291 - [0xb9dbf16c] = 0xa3deae54 -  pop esp
0292 - [0xb9dbf170] = 0xa3d0f4e4 -  mov edx,DWORD PTR [ebp-0x14] found in "_IO_file_write+68"
0293 - [0xb9dbf174] = 0x00000001 -
0294 - [0xb9dbf178] = 0xa3df2000 - String(u'Error !\nwww.roi-heenok.com/html/images/gifs/waterm[...]')
0295 - [0xb9dbf17c] = 0x00000001 -
0296 - [0xb9dbf180] = 0xa3d66103 -  pop ebx found in "write+35"
0297 - [0xb9dbf184] = 0xa3deae54 -  pop esp
0298 - [0xb9dbf188] = 0xa3d0f4e4 -  mov edx,DWORD PTR [ebp-0x14] found in "_IO_file_write+68"
0299 - [0xb9dbf18c] = 0x00000001 -
0300 - [0xb9dbf190] = 0xa3df2000 - String(u'Error !\nwww.roi-heenok.com/html/images/gifs/waterm[...]')
0301 - [0xb9dbf194] = 0x0000001a -
0302 - [0xb9dbf198] = 0xa3deb4a0 -  test BYTE PTR [edx],ch found in "_IO_2_1_stdout_"
0303 - [0xb9dbf19c] = 0xffffffff -
```

Figure 2: *dps* in action

We can clearly see that this view is really adapted when you have to find an ideal pointer to leak, it can be also cool to see which pointers are pointing on which ASM instructions.

Part III
Conclusion

I hope you have enjoyed this little paper, I've really tried to give you an overview of the python API in GDB, but if you want to read other documentations and learn more about the subject, here are some interesting links I found:

- Official documentation
- PythonGdbTutorial, a bunch of examples written by Tom Tromey
- PythonGDB tutorial for reverse engineering - part 1, written by @delroth
- Using Python to debug C and C++ code (using gdb) by David Malcolm ; check out the talk video.

Also, I wanted to give you some other useful tips:

- Get the pid of the process debugged:

```
gdb$ python print gdb.selected_inferior().pid
1321
```

- Read the process memory (this is the other way to dump data without using *gdb.Value* objects ; note that you can also write memory thanks to *gdb.Inferior.write_memory()* and search with *gdb.Inferior.search_memory()*):

```
gdb$ python
>b = gdb.selected_inferior().read_memory(0x7fffffffe6b8, 10)
>print ', '.join('%#.2x' % ord(b[i]) for i in range(len(b)))
>0x8d, 0x1c, 0x65, 0xf7, 0xff, 0x7f, 0x00, 0x00, 0x00, 0x00
gdb$ x/10bx $rsp
0x7fffffffe6b8: 0x8d    0x1c    0x65    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffe6c0: 0x00    0x00
```

- Give an address, and retrieve in which shared library it's pointing to (NB: it only works for addresses pointing in shared libraries):

```
gdb$ p /x &system
$6 = 0x7ffff76715b0
gdb$ python print gdb.solib_name(0x7ffff76715b0)
/lib/libc.so.6
```

You will be able to find my different python examples at the end of the post on my personal blog and by the way, if you have already coded cool stuff with python gdb feel free to leave comments. Thanks to @Ivanlef0u, sha, @__x86 for the corrections and @sm0k_ for the cool discussions we had on this topic. One last thing, if you enjoy wargames, give a shot at Vanilla.