A Basic Guide to Discovering Attack Surface with Ghidra and GDB

In this article I will introduce, and provide an example of, another technique from my upcoming training, *A Basic Guide to Bug Hunting with Ghidra*. The topic this time is on attack surface enumeration and specifically how to use Ghidra to produce a secondary script for tracing targeted function calls in GDB.

The content associated with this post is available on GitHub and has been tested with Ghidra 11.0.3: https://github.com/cy1337/ghidra_gdb_tracer

- GenPyGDB.py
  - Ghidra Python script which generates a GDB Python script
- CommandServer
  - An example program to analyze
- CommandClient.py
  - A client for interacting with *CommandServer*
- Dockerfile
  - Source for the DockerHub image
- gdb_script.py
  - Example output generated by running *GenPyGDB.py* on CommandServer

The container suitable for running the target program is available on DockerHub.

You can test the example output script like so:

```
docker run -it cy1337demos/ghidra_gdb_tracer gdb -x gdb_script.py CommandServer
```

And then from a separate terminal you can interact with the command server:

```
docker exec -it \
$(docker ps --filter ancestor=ghidra_gdb --format "{{.Names}}") \
./CommandClient.py 1 /etc/issue 127.0.0.1:8080
```

The goal of this project is to have a flexible process for identifying which functions are called when processing a particular testcase. We will do this by adding logging breakpoints at the start of every function. This is an ideal way to identify targets for fuzzing without a substantial up front reversing effort. (More on this in my last post on using AFL QEMU.)

GDB Commands

Let's start by looking at how the debugger script will function and then circle back to discuss what will be needed in the Ghidra script.

The script will need to do a few things:

- Determine the base address of the debug target (e.g. because of ASLR)
- Add breakpoints at each function
- Log the function name on breakpoint trigger
- Optionally disable the triggered breakpoint
- Resume execution

Without getting into system specific modifications, the easiest way to overcome ASLR is to determine the base address of the process at runtime. One option for doing this is to get calculate an offset between the static symbol in Ghidra's listing and the dynamic address at runtime.

For example, given the following THUNK FUNCTION for *read*:

```
             *******************************
             *        THUNK FUNCTION        *
             *******************************
             thunk ssize_t read(int __fd, …
              Thunked-Function: <EX…
    ssize_t   RAX:8   <RETURN>
    int       EDI:4   __fd
    void *    RSI:8   __buf
    size_t    RDX:8   __nbytes
             <EXTERNAL>::read          XREF[… FUN_001369c8:001 …
00134610 f3       ENDBR64
         0f
         1e fa
```

We can calculate use this address, 0x134610 to calculate the runtime base address relative to the addresses in Ghidra:

```
starti
set $base_addr = read - 0x134610
```

Adding this $base_addr to an address in Ghidra will give the address in the target process where we want to set a breakpoint. For example, setting a breakpoint at *FUN_00134869* looks like:

```
break *($base_addr + 0x134869)
```

Next we can add a command to run at that breakpoint to print the function name and continue:

```
command
  print "FUN_00134869()"
  continue
end
```

GDB Python

We could use Ghidra scripting to produce a GDB command script like this but there are some more advanced situations where this becomes unfavorable. Using GDB's Python API can leave us in a better position for later adding more advanced logic to analyze data structures at runtime. Using GDB's Python API, the logic can be summarized with:

```
import gdb

class FunctionBreakpoint(gdb.Breakpoint):
    def __init__(self, location, func_name):
        location += base_addr
        bp_loc = '*{}'.format(hex(location))
        super(FunctionBreakpoint, self).__init__(bp_loc, gdb.BP_BREAKPOINT,
internal=False)
        self.func_name = func_name

    def stop(self):
        gdb.write('Function: {}\n'.format(self.func_name))
        return False

def set_base_address():
    gdb.execute('starti')
    gdb.execute('set $base_addr = read - 0x134610')
    base_addr = int(gdb.parse_and_eval('$base_addr'))
    gdb.write('Base address set to: {:#x}\n'.format(base_addr))
    return base_addr

base_addr = set_base_address()
```

```
FunctionBreakpoint(0x134869, 'FUN_00134869')
```

Ghidra Python

Now that we have a skeleton for the target script, it's time to start authoring a Ghidra script to generate it. The Ghidra script will need to write out this template while filling in the address for *read* and then writing a *FunctionBreakpoint* call for each function.

The address of *read* can be obtained by walking the symbol tree:

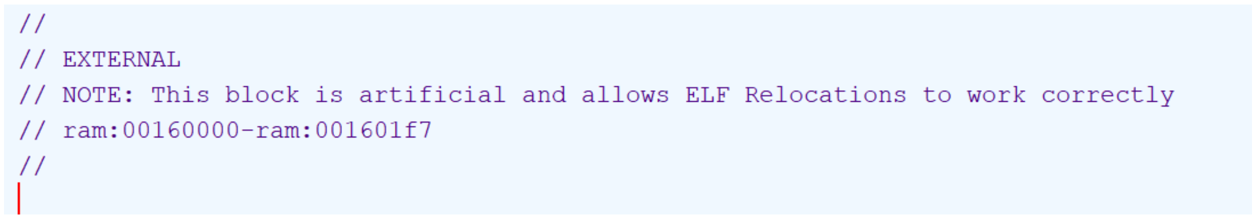
```
sym_table = currentProgram.getSymbolTable()
for symbol in sym_table.getPrimarySymbolIterator(True):
    if symbol.getName() == "read":
        read_addr = symbol.getAddress()
        break
```

Function addresses can then be listed through the function manager:

```
fm = currentProgram.getFunctionManager()
functions = fm.getFunctions(True)
for func in functions:
    func_name = func.getName()
    func_entry = func.getEntryPoint().getOffset()
```

Artificial Blocks

The above code snippet will iterate over all function symbols Ghidra is aware of. Unfortunately this includes some sections which are artificially created by Ghidra to support relocation:



```
//
// EXTERNAL
// NOTE: This block is artificial and allows ELF Relocations to work correctly
// ram:00160000-ram:001601f7
//
```

And so to avoid adding breakpoints within these artificial blocks, we need to walk the memory to identify these blocks so we can avoid adding breakpoints.

```
memory = currentProgram.getMemory()
blocks = memory.getBlocks()
artificial_blocks = []
```

```
for block in blocks:
    # Check if the block has a specific name or comment
    if block.getComment() and "artificial" in block.getComment().lower():
        artificial_blocks.append(block)
    elif block.getName() and "artificial" in block.getName().lower():
        artificial_blocks.append(block)
```

Test Run

Putting all these pieces together produces the script you'll find on GitHub. Now let's take it for a test drive with the provided *CommandServer*. First load the program into Ghidra and, using Ghidra's script manager, run the provided script from GitHub to produce a file, *gdb_script.py* or similar which can be run through *gdb* as follows:

```
$ docker run -it cy1337demos/ghidra_gdb_tracer gdb -q -x gdb_script.py
CommandServer
Reading symbols from CommandServer...
(No debugging symbols found in CommandServer)

Program stopped.
0x00007ffff7fd0100 in _start () from /lib64/ld-linux-x86-64.so.2
Base address set to: 0x555555454000
Breakpoint 1 at 0x555555588000
Breakpoint 2 at 0x555555588020
Breakpoint 3 at 0x5555555883d0
Breakpoint 4 at 0x5555555883e0
Breakpoint 5 at 0x5555555883f0
```
…

```
Breakpoint 90 at 0x55555558ad90
Breakpoint 91 at 0x55555558ad98
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Function: entry
Function: FUN_00136d20
Function: _DT_INIT
Function: _INIT_0
Function: FUN_001347e0
Function: FUN_00136a55
Function: socket
Function: setsockopt
Function: htons
Function: bind
```

```
Function: listen
Function: printf
Server is listening on port 8080
Function: puts
Waiting for a new connection...
Function: accept
```

You can now interact with *CommandServer* by sending data to it using *CommandClient.py* as follows:

```
$ docker exec -it $(docker ps --filter ancestor=ghidra_gdb --format "{{.Names}}")
./CommandClient.py 1 /etc/issue 127.0.0.1:8080

Received response: UBUNTU 20.04 LTS \N \L
```

While handling this request, GDB is also logging what additional (new) functions have been called:

```
Connection established with a client.
Function: FUN_001369c8
Read 0 bytes...
Function: read
Function: ntohs
Received header for 24 bytes
Function: malloc
Read 0 bytes...
read: 24
Processing command: 1
Function: FUN_00134869
Function: FUN_001349b0
Function: toupper
Function: send
Function: free
Processed opcode 1 and sending 24 bytesRead 0 bytes...
Failed to read complete header or client disconnected.
Function: shutdown
Function: close
Connection with client closed. Ready for a new connection.
Waiting for a new connection...
```

Wrapping Up

That’s as far as I’ll be taking this post but I hope you’ve enjoyed reading and will find this helpful in your own projects. If you want to continue the learning, be sure to follow cy1337 on Medium and consider joining me at Black Hat USA for *A Basic Guide to Bug Hunting with Ghidra* where we will continue down this rabbit hole by looking at ways to drill-down into specific points of interest by further inspecting decompiled outputs and call stacks. This course will be offered August 3-4 (Sat/Sun) and again August 5-6 (Mon/Tue). Hope to see you there!