Stack trace explained

Summary and quick start

Once in a while Manalink shows a message to inform us about a function call that didn‘t go as expected, for example get_card_instance(1, -1). The dump.dmp file provides us the information which function was the culprit. It consists of an error message („bad parameters“), what went wrong („get_card_instance(1, -1)“) and the way the program took in the form of a stack trace. If you don’t want details this section is all you need. If you want to look under the hood, read the whole thing.

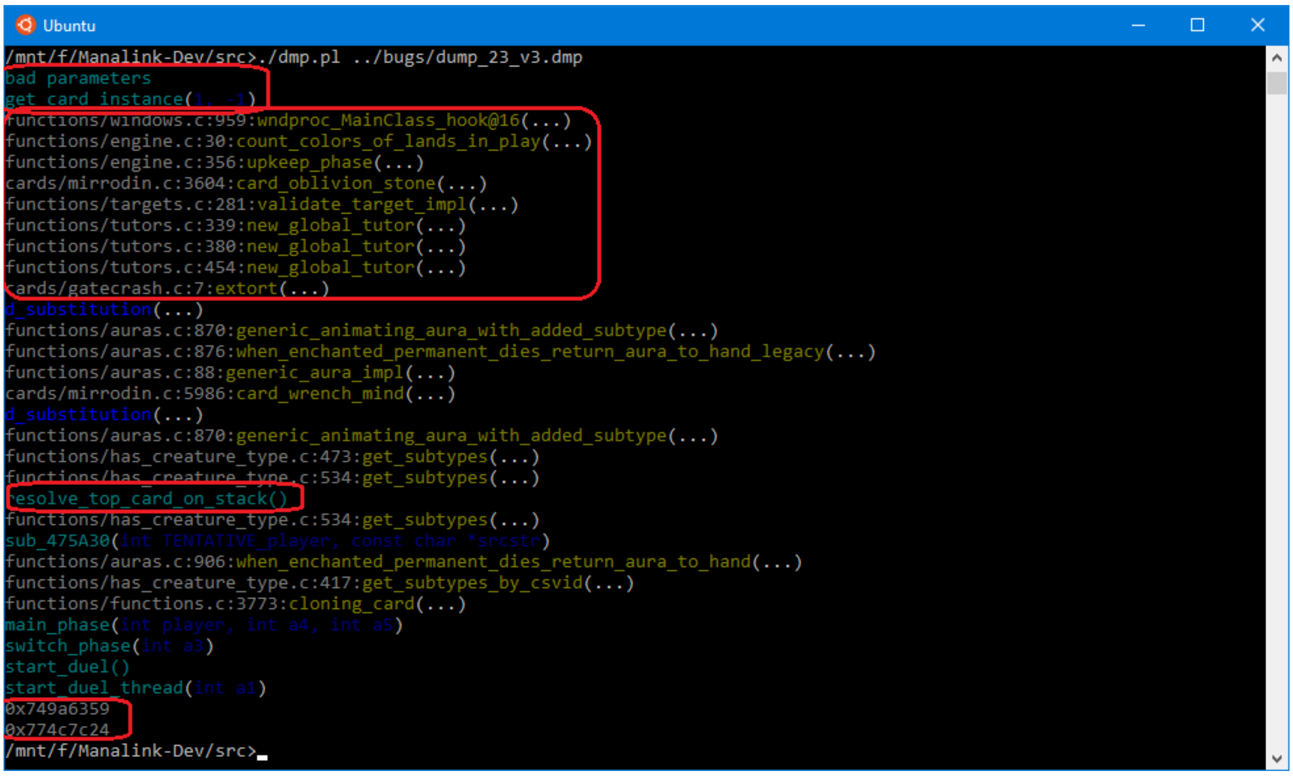
Without going into too much detail, a stack trace is the call order of functions with the last function at the top, when things went haywire. So if you know who called whom you can work backwards and see where the actual bug happens. Fortunately, some friendly programmer provided us with a perl script called dmp.pl in the main directory where also ManalinkEh.dll resides. It worked for me under the Linux Subsystem for Windows. Might as well under MinGW64 but I didn’t test this. To get it going do as follows:

Install perl and the package libcapture-tiny-perl. Under the Linux Subsystem enter

```
sudo apt install perl
sudo apt install libcapture-tiny-perl
```

in the command line and the packages will install. Fix the path to Magic-trace in line 33, Easiest thing to do is to set it to ‘./Magic-trace.c’ as both are in the same folder.
Say you dump file is in the same folder and called dump_23_v3.dmp. Then run it with

```
./dmp.pl ../bugs/dump_23_v3.dmp
```



```
/mnt/f/Manalink-Dev/src>./dmp.pl ../bugs/dump_23_v3.dmp
bad parameters
get_card_instance(1, -1)
functions/windows.c:959:wndproc_MainClass_hook@16(...)
functions/engine.c:30:count_colors_of_lands_in_play(...)
functions/engine.c:356:upkeep_phase(...)
cards/mirrodin.c:3604:card_oblivion_stone(...)
functions/targets.c:281:validate_target_impl(...)
functions/tutors.c:339:new_global_tutor(...)
functions/tutors.c:380:new_global_tutor(...)
functions/tutors.c:454:new_global_tutor(...)
cards/gatecrash.c:7:extort(...)
d_substitution(...)
functions/auras.c:870:generic_animating_aura_with_added_subtype(...)
functions/auras.c:876:when_enchanted_permanent_dies_return_aura_to_hand_legacy(...)
functions/auras.c:88:generic_aura_impl(...)
cards/mirrodin.c:5986:card_wrench_mind(...)
d_substitution(...)
functions/auras.c:870:generic_animating_aura_with_added_subtype(...)
functions/has_creature_type.c:473:get_subtypes(...)
functions/has_creature_type.c:534:get_subtypes(...)
resolve_top_card_on_stack()
functions/has_creature_type.c:534:get_subtypes(...)
sub_475A30(int TENTATIVE_player, const char *srcstr)
functions/auras.c:906:when_enchanted_permanent_dies_return_aura_to_hand(...)
functions/has_creature_type.c:417:get_subtypes_by_csvid(...)
functions/functions.c:3773:cloning_card(...)
main_phase(int player, int a4, int a5)
switch_phase(int a3)
start_duel()
start_duel_thread(int a1)
0x749a6359
0x774c7c24
/mnt/f/Manalink-Dev/src>
```

At the top is the actual error message in turquoise. The yellow functions belong to the dll and are shown with file name and line number. Turquoise functions lower on the screen are from the exe. There we don’t have line numbers, unfortunately. The grey functions (actually the addresses in hex code) are system calls.

What is a stack trace, anyway?

So you want more. Imagine a bunch of functions. Start with main() which in turn calls f_A() which in turn calls f_B():

```
main() → f_A() → f_B()
```

f_B() misbehaves and the system (or some friendly subroutine as in functions/show_backtrace.c) generates a trace of the order of calls **in reverse order**, starting with the misbehaving function:

```
address of f_B() where the issue happened
address of f_A() where f_B() was called
address of main() where f_A() was called
some other addresses from the OS initiating the call of main()
```

The last few lines don‘ t interest us since the issue started after calling main().

Imagine a pile of plates: to get to the bottom plates you have to take away all plates above in reverse order. Same story here.

That is so abstract. Can you give me an example?

Sure. Extract the stackTrace.zip to some folder and open a Powershell or Cmd. I assume you have mingw32 installed and paths are set accordingly.

StackTrace.c shows an example. The tracing code is pretty much the same as in show_backtrace.c. Run the following commands to compile it:

```
dlltool.exe -k -d .\capturestackbacktrace.def -l libcapturestackbacktrace.a
```

This create a Unix library with the link into the Windows OS function CaptureStackBackTrace which does the actual work.

```
mingw32-gcc-4.8.1.exe -g .\stackTrace.c -o .\stackTrace.exe -L. -lcapturestackbacktrace
```

The -g option is needed to create debugging symbols, i.e. names for function instead of addresses. Not sure whether we actually need the -L. option, the -lcapturestackbacktrace links our newly created library to the main file. Running it with

```
.\stackTrace.exe
```

creates the following output:

```
Calling function_a...
Calling function_b...
Calling function_c...
Stack trace...
Returning from function_c...
Returning from function_b...
Returning from function_a…
```

and a file called dump.dmp to boot. Now switch to a Unix shell. I use the Linux subsystem under Windows with Ubuntu, every other installation should also do. Mingw32 didn‘t work for my, I guess it is because Windows uses a different format for symbols and you would have to convert those first. In case you haven‘t installed the binutils package this is a good opportunity to do so.

Now run

```
addr2line -e stackTrace.exe -f -p < dump.dmp
```

A couple of comments:

- addr2line converts addresses (what dump.dmp gives us) to line numbers in the source code
- The -e option tells it which executable file to use,
- -f gives function names,
- -p makes for a nicer human readable format ...
- … and „< dump.dmp“ uses dump.dmp as input so you don‘ t have to type every address by hand (did I tell you that I am somewhat lazy?).

The output should look like this:

```
function_c at F:\Foo/./stackTrace.c:15
function_b at F:\Foo/./stackTrace.c:31
function_a at F:\Foo/./stackTrace.c:38
main at F:\Foo/./stackTrace.c:45
__mingw_CRTStartup at e:\p\giaw\src\pkg\mingwrt-4.0.3-1-mingw32-src\bld/../mingwrt-
4.0.3-1-mingw32-src/src/libcrt/crt/crt1.c:254
?? at e:\p\giaw\src\pkg\mingwrt-4.0.3-1-mingw32-src\bld/../mingwrt-4.0.3-1-mingw32-src/
src/libcrt/crt/crt1.c:272
?? ??:0
?? ??:0
```

Line 15 is where the stack trace was initiated, the other lines are where the calling functions ended. Everything below main is system-related stuff we don‘ t need.

Yeah, fine, but how does this help us with Manalink?

Quite a lot.

Before we start take the dump.dmp file (I attached an example) and strip out the first two lines as well as the numbering. I use Notepad++ as it has a column mode that is exactly what we need here. It should now look like this (I saved it as mydump.dmp to preserve the original file):

```
0x0251E298
0x024D48CE
0x024D52A8
...
```

Let‘ s continue with our actual program. When you run mingw32-make in the root directory it builds the ManalinkEh.dll and some other useful stuff. Look at the last two lines:

```
objcopy --only-keep-debug ManalinkEh.dll ManalinkEh.dbg
objcopy --strip-debug ManalinkEh.dll
```

These create two files, namely the debug symbols in ManalinkEh.dbg and the DLL itself, ManalinkEh.dll **without symbols**. This is the usual convention under Windows and something we need to fix and pronto please, because if you fire up a Unix shell and run

```
addr2line -e ManalinkEh.dll -f -p < mydump.dmp
```

you get only gibberish. No symbols, no proper output. We fix this now:

```
objcopy --add-gnu-debuglink=ManalinkEh.dbg ManalinkEh.dll
```

Let's try again:

```
addr2line -e ManalinkEh.dll -f -p < mydump.dmp
```

The output should be something like this (path depends on your installation):

```
add_counter at F:\Manalink-Dev/./manalink.h:433
select_card_from_zone at F:\Manalink-Dev/functions/tutors.c:103
new_global_tutor at F:\Manalink-Dev/functions/tutors.c:372
champion at F:\Manalink-Dev/cards/lorwyn.c:198
choose_existing_counter_type at F:\Manalink-Dev/functions/counters.c:1991
initialize_ability_tooltip_names at F:\Manalink-Dev/functions/windows.c:574
wndproc_CardClass_hook@16 at F:\Manalink-Dev/functions/windows.c:925
bitset128_empty at F:\Manalink-Dev/functions/windows.c:168
affect_me at F:\Manalink-Dev/./manalink.h:366
?? at ??:?
charge_mana_multi at F:\Manalink-Dev/functions/functions.c:2044
…
```

The ?? means that no symbols are available, this might be a call to the exe. We don‘ t know (yet).

One caveat: versions must match, i.e. this dump is produced based on the June 2020 v2 Manalink version (I think…). If you changed the source code in the meanwhile line numbers probably don’t match any more (I didn’t try it out) so you have to go back to the older version or keep an extra copy.

Isn’t there an easier way?

Yes, there is and if you made it until here you are really interested in how things work under the hood. There is a Perl script called dmp.pl in the main directory where also ManalinkEh.dll resides. It worked for me under the Linux Subsystem for Windows. Might as well under MinGW64 but I didn’t test this. To get it going do as follows:

Install perl and the package libcapture-tiny-perl. Under the Linux Subsystem enter

```
sudo apt install perl
sudo apt install libcapture-tiny-perl
```

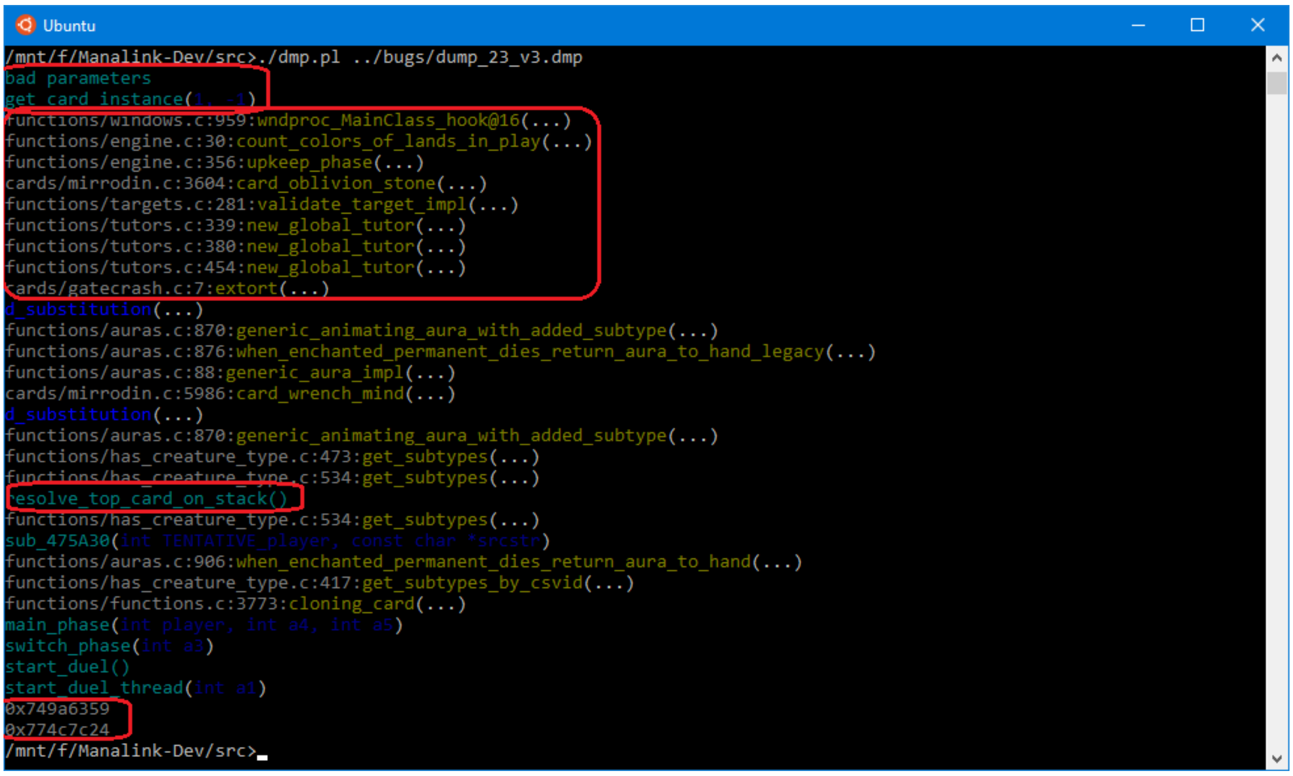
in the command line and the packages will install. If you use MinGW64 fire up Mingw-get and look for them.

Fix the path to Magic-trace in line 33, Easiest thing to do is to set it to ‘./Magic-trace.c’ as both are in the same folder.

Say you dump file is in the same folder and called dump_23_v3.dmp. Then run it with

```
./dmp.pl ../bugs/dump_23_v3.dmp
```

Your output should look like this:



```
/mnt/f/Manalink-Dev/src>./dmp.pl ../bugs/dump_23_v3.dmp
bad parameters
get_card_instance(1, -1)
functions/windows.c:959:wndproc_MainClass_hook@16(...)
functions/engine.c:30:count_colors_of_lands_in_play(...)
functions/engine.c:356:upkeep_phase(...)
cards/mirrodin.c:3604:card_oblivion_stone(...)
functions/targets.c:281:validate_target_impl(...)
functions/tutors.c:339:new_global_tutor(...)
functions/tutors.c:380:new_global_tutor(...)
functions/tutors.c:454:new_global_tutor(...)
cards/gatecrash.c:7:extort(...)
d_substitution(...)
functions/auras.c:870:generic_animating_aura_with_added_subtype(...)
functions/auras.c:876:when_enchanted_permanent_dies_return_aura_to_hand_legacy(...)
functions/auras.c:88:generic_aura_impl(...)
cards/mirrodin.c:5986:card_wrench_mind(...)
d_substitution(...)
functions/auras.c:870:generic_animating_aura_with_added_subtype(...)
functions/has_creature_type.c:473:get_subtypes(...)
functions/has_creature_type.c:534:get_subtypes(...)
resolve_top_card_on_stack()
functions/has_creature_type.c:534:get_subtypes(...)
sub_475A30(int TENTATIVE_player, const char *srcstr)
functions/auras.c:906:when_enchanted_permanent_dies_return_aura_to_hand(...)
functions/has_creature_type.c:417:get_subtypes_by_csvid(...)
functions/functions.c:3773:cloning_card(...)
main_phase(int player, int a4, int a5)
switch_phase(int a3)
start_duel()
start_duel_thread(int a1)
0x749a6359
0x774c7c24
/mnt/f/Manalink-Dev/src>_
```

At the top is the actual error message in turquoise. The yellow functions belong to the dll and are shown with file name and line number. Turquoise functions lower on the screen are from the exe. There we don’t have line numbers, unfortunately. The grey functions (actually the addresses in hex code) are system calls.

Now that you know which function misbehaved where you can take it from there.