



# Getting fun with Frida

## Turbo Talk – Ekoparty

Nahuel Riva

A solid orange square containing the word "FRIDA" in large white capital letters.

October 2016

# Getting fun with Frida



# Agenda

# Agenda

- Intro
  - What's DBI?
  - Why do we need DBI?
  - How do I perform DBI? (frameworks)
- Frida
  - What's Frida?
  - Why would I need Frida?
    - Differences with other frameworks
  - How do I use Frida
    - API
      - Interceptor
      - Stalker
    - Tools based on Frida
- Demos

# Intro

# Intro – What's DBI?

- Definition taken from:  
<http://uninformed.org/index.cgi?v=7&a=1&p=3>
- “Dynamic Binary Instrumentation (DBI) is a method of analyzing the behavior of a binary application at runtime through the injection of instrumentation code. [...] makes it possible to gain insight into the behavior and state of an application at various points in execution.“

# Intro – What's DBI?

- Instrumentation code executes as part of the normal instruction stream after being injected
- Instrumentation code will be entirely transparent to the application that it's been injected to
- Instrumentation code executes at runtime

# Intro – Why do we need DBI?

- As an alternative
  - Debuggers
  - API hooking engines
- Evolution
  - More complex tasks to achieve (profiling, taint analysis, detection of possible bugs)

# Intro – How do I perform DBI? (frameworks)

- Two main DBI frameworks:
  - PIN: proprietary framework written in C/C++. Works on Windows/Linux/OSX/Android and i386/AMD64
  - <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>
  - DynamoRIO: originally a proprietary framework then open sourced (BSD). Created by HP (Dynamo optimization system) and MIT (RIO research group). Works on Windows/Linux and i386/AMD64.
  - <https://en.wikipedia.org/wiki/DynamoRIO>

# Intro – How do I perform DBI? (frameworks)

- In both cases, you write a Pin/DynamoRIO tool using C/C++ language and inject C/C++ code
- Compile the Pin/Dynamo tool as a .dll/.so
- Inject the library into the target process using a command-line tool/GUI application

# Intro – How do I perform DBI? (frameworks)

Pintool example ([source/tools/ManualExamples/inscount0.cpp](#)):

```
int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

# Intro – How do I perform DBI? (frameworks)

```
ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every instruction is executed
VOID docount() { icount++; }

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction, no arguments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");
```

# Intro – How do I perform DBI? (frameworks)

```
// This function is called when the application exits  
VOID Fini(INT32 code, VOID *v)  
{  
    // Write to a file since cout and cerr maybe closed by the application  
    OutFile.setf(ios::showbase);  
    OutFile << "Count " << icount << endl;  
    OutFile.close();  
}
```

# Intro – How do I perform DBI? (frameworks)

- For example, Pin can be executed as follow:
  - pin.bat -t pintool.dll [pintoolargs] --program.exe [programargs]
  - pin.bat -pid<programpid> -t pintool.dll [pintoolargs]

# Intro – How these frameworks work?

- JIT compiler
  - Input: binary code
  - Output: equivalent code with introspection code
  - The code is generated only when it is needed
- The only code that is executed is the code generated by the JIT compiler
- The original code remains in memory just as a reference but it is **never** executed



# Frida

# Frida – What's Frida?

- Dynamic instrumentation toolkit
- Scriptable
  - Execute **Javascript** programs inside another process. It uses V8 and Duktape and JavaScriptCore (deprecated) engines.
- Multi-platform and multi-arch
  - Windows/Mac/Linux/Android/iOS/QNX – i386/AMD64/ARM/ARM64
- It has bindings for Python, .NET, C and Node.js
- Open-source (LGPL v2)

# Frida – Why would I need Frida?

- For reverse engineering in general
  - Dynamic binary instrumentation
  - Debugging
- To develop introspection tools very quickly to help you in the RE process

# Frida – Pros & Cons against other frameworks

## ■ Pros

- It has bindings for other languages like .NET, Python, C
- No need to compile the tool
- Rapid tool development
- Continuous development (new features and bug fixing)

## ■ Cons

- Less mature than other DBI frameworks (contains bugs)
- Lack of some functionality
- Less granularity than other frameworks

# How do I use Frida?

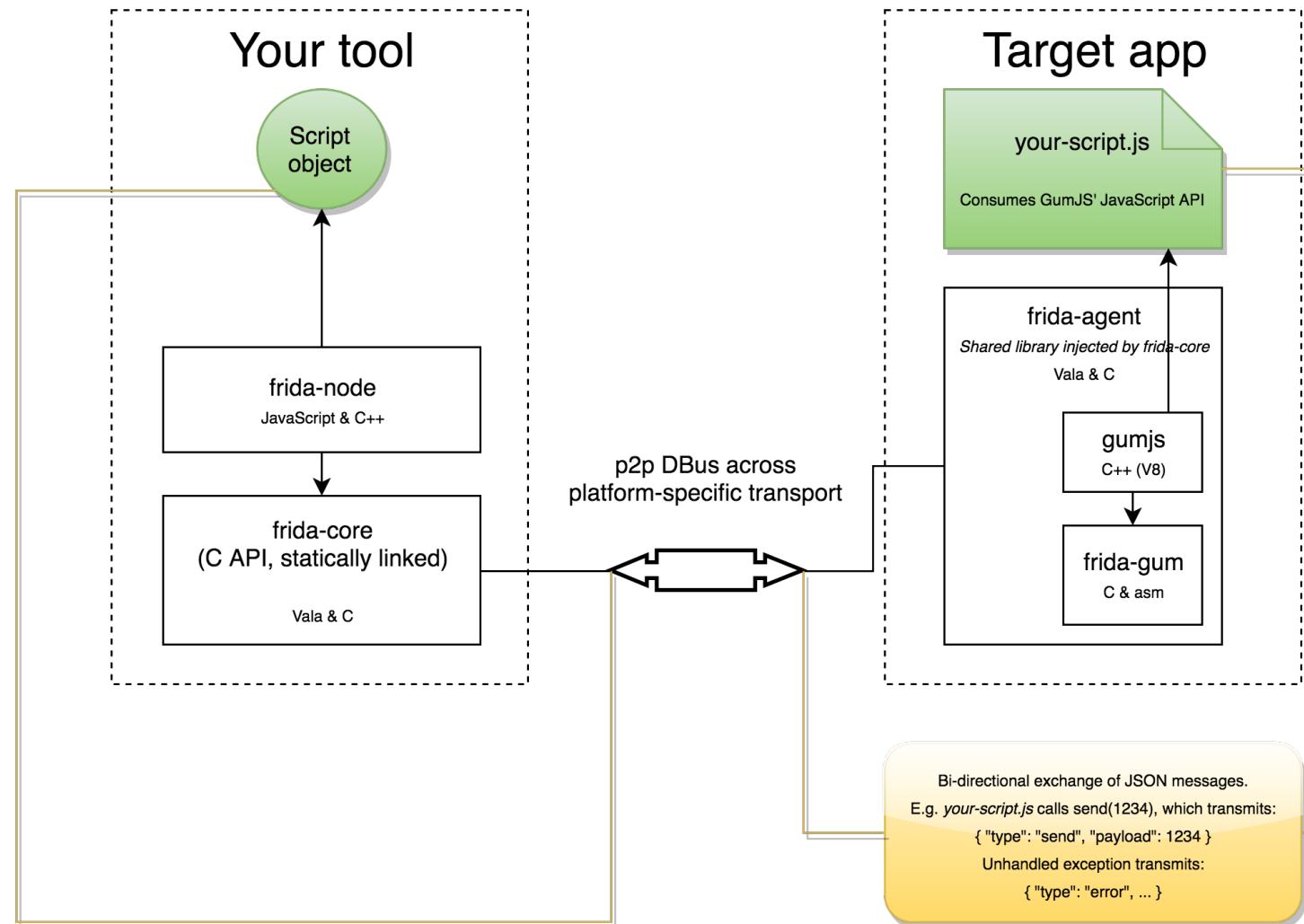
# Frida – How do I use Frida?

- First, you need to install it:
  - Windows:
    - C:\Users\travesti>pip install frida
  - Linux:
    - travesti@palermo:~\$ sudo pip install frida
  - Then ...

# Frida – How do I use Frida?

- As easy as this:
  - >> `import frida`
  - >> `session = frida.attach("notepad.exe")`
  - >> `print([x.name for x in session.enumerate_modules()])`
- [u'notepad.exe', u'ntdll.dll', u'kernel32.dll', u'KERNELBASE.dll', u'ADVAPI32.dl  
I', u'msvcrt.dll', u'sechohost.dll', u'RPCRT4.dll', u'GDI32.dll', u'USER32.dll', u  
'LPK.dll', u'USP10.dll', u'COMDLG32.dll', u'SHLWAPI.dll', u'COMCTL32.dll', u'SHE  
LL32.dll', u'WINSPOOL.DRV', u'ole32.dll', u'OLEAUT32.dll', u'VERSION.dll', u'IMM  
32.DLL', u'MSCTF.dll', u'actuser.dll', u'acdetoured.dll', u'msvcp60.dll', u'CRYP  
TBASE.dll', u'uxtheme.dll', u'dwmapi.dll', u'CLBCatQ.DLL', u'frida-agent-64.dll'  
, u'DNSAPI.dll', u'WS2\_32.dll', u'NSI.dll', u'WINMM.dll', u'PSAPI.DLL', u'ntmart  
a.dll', u'WLDAP32.dll']

# Frida – Architecture



# Frida – JavaScript API

- Its Javascript API has different components to interact with a process (<http://www.frida.re/docs/javascript-api>):
  - console
  - Process
  - Module
  - Memory
  - Thread
  - Socket
  - File
  - Instruction

# Frida – JavaScript API - Console

- **console**: used for output.

- `console.log(line)`
- `console.warn(line)`
- `console.error(line)`

# Frida – JavaScript API - Process

- **Process**: functions and properties used to interact with a process.
- `Process.arch`, `Process.platform`
- `Process.isDebuggerAttached`
- `Process.enumerateThreads(callbacks)`
- `Process.findModuleByAddress(address)`
- `Process.findModuleByName(name)`
- `Process.enumerateModules(callbacks)`
- [...]

# Frida – JavaScript API - Module

- **Module**: used to interact with modules residing in the process.
  - `Module.enumerateImports(name, callbacks)`
  - `Module.enumerateExports(name, callbacks)`
  - `Module.enumerateRanges(name, protection, callbacks)`
  - `Module.findBaseAddress(name)`
  - `Module.findExportByName(moduleInull, exp)`
  - [...]

# Frida – JavaScript API - Memory

- **Memory**: used to interact with memory pages residing in a given process.
- Memory.scan(address, size, pattern, callbacks)
- Memory.alloc(size)
- Memory.copy(dst, src, n)
- Memory.protect(address, size, protection)
- Memory.read\*/write\*
- MemoryAccessMonitor (monitor read/write/execute)
- [...]

# Frida – JavaScript API - Thread

- **Thread**: used to interact with threads from a process.
  - Thread.backtrace([context, backtracer])
  - Thread.sleep(delay)

# Frida – JavaScript API - Socket

- **Socket**: used to handle sockets.

- `Socket.type(handle)`
- `Socket.localAddress(handle)`
- `Socket.peerAddress(handle)`

# Frida – JavaScript API - File

- **File**: used to handle file I/O.

- `File(filePath, mode)`
- `write`
- `read`
- `flush`
- `close`

# Frida – JavaScript API - Instruction

- **Instruction**: used to get information about a given instruction from process's code.
- `Instruction.parse(target)`

# Frida – Interceptor/Stalker

- Frida has two main components exposed through its API:
- Interceptor
  - Normal operation mode (hooking)
  - No stealthiness
- Stalker
  - Instrumentation per-se
  - Stealth (kind of)
  - Lack of functionality (CALL/RET)
  - More details: <https://medium.com/@oleavr/anatomy-of-a-code-tracer-b081aadb0df8>

# Frida – How do I use Interceptor?

- Interceptor example:

```
def main(target_process):
    heapalloc_rva = getExportedFunctionRva('RtlAllocateHeap', 'ntdll.dll')

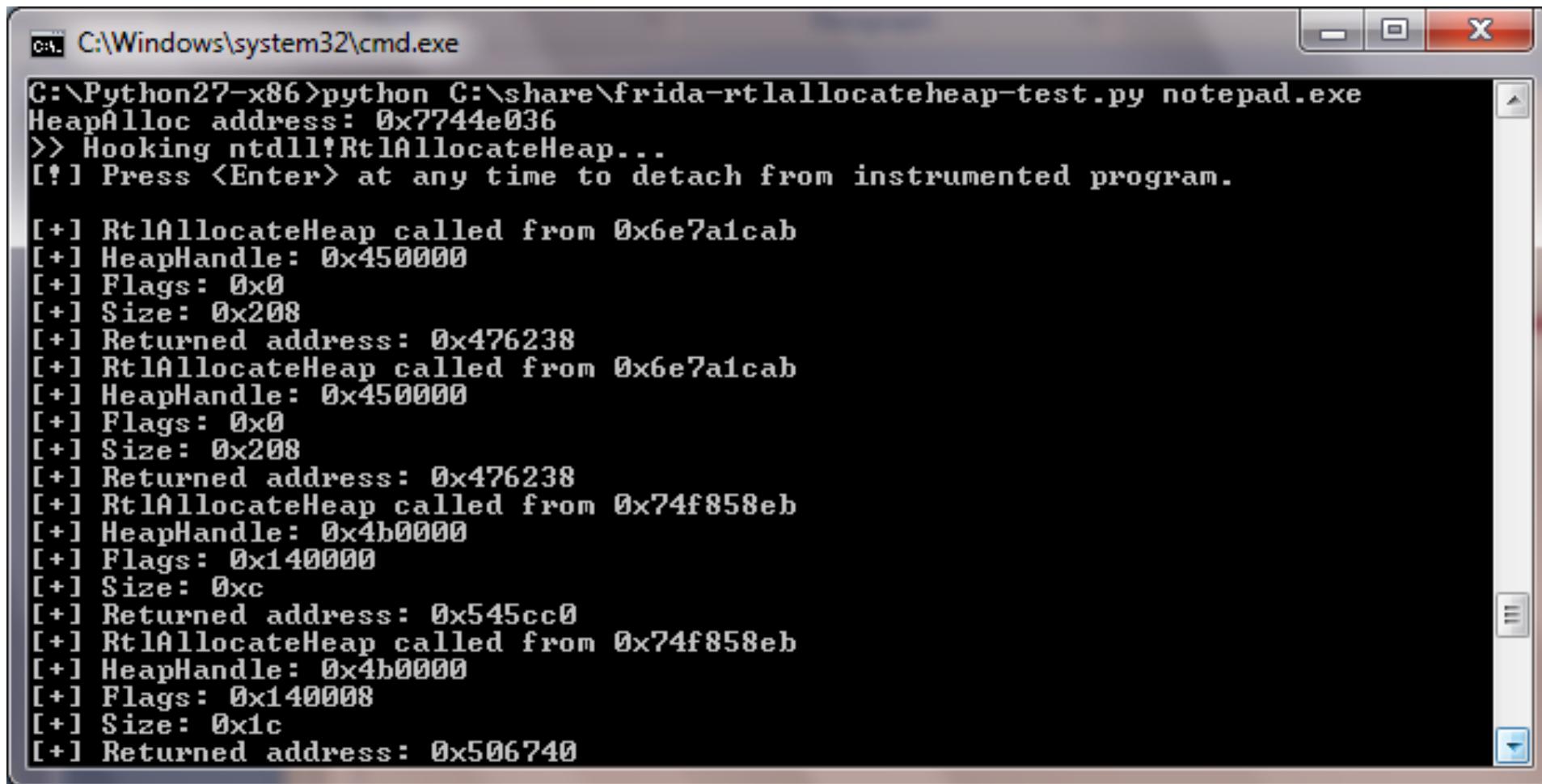
    session = frida.attach(target_process)
    script = session.create_script("""
        var RtlAllocateHeapAddr = Module.findBaseAddress('ntdll.dll').add(0x%x);
        console.log('HeapAlloc address: ' + RtlAllocateHeapAddr.toString());

        console.log('>> Hooking ntdll!RtlAllocateHeap...');

        Interceptor.attach(RtlAllocateHeapAddr, {
            onEnter: function (args) {
                console.log('[+] RtlAllocateHeap called from ' + this.returnAddress.sub(6).toString());
                console.log('[+] HeapHandle: ' + args[0].toString());
                console.log('[+] Flags: ' + args[1].toString());
                console.log('[+] Size: ' + args[2].toString());
            },
            onLeave: function (retval) {
                console.log('[+] Returned address: ' + retval.toString());
            }
        });
    """ % heapalloc_rva)

    script.on('message', on_message)
    script.load()
    raw_input('[!] Press <Enter> at any time to detach from instrumented program.\n\n')
    session.detach()
```

# Frida – Interceptor example output



The screenshot shows a Windows command prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The command entered is 'C:\Python27-x86>python C:\share\frida-rt\allocateheap-test.py notepad.exe'. The output displays multiple instances of the 'RtlAllocateHeap' function being called from the 'notepad.exe' process. Each call is detailed with its address, heap handle, flags, size, and returned address.

```
C:\Python27-x86>python C:\share\frida-rt\allocateheap-test.py notepad.exe
HeapAlloc address: 0x7744e036
>> Hooking ntdll!RtlAllocateHeap...
[!] Press <Enter> at any time to detach from instrumented program.

[+] RtlAllocateHeap called from 0x6e7a1cab
[+] HeapHandle: 0x450000
[+] Flags: 0x0
[+] Size: 0x208
[+] Returned address: 0x476238
[+] RtlAllocateHeap called from 0x6e7a1cab
[+] HeapHandle: 0x450000
[+] Flags: 0x0
[+] Size: 0x208
[+] Returned address: 0x476238
[+] RtlAllocateHeap called from 0x74f858eb
[+] HeapHandle: 0x4b0000
[+] Flags: 0x140000
[+] Size: 0xc
[+] Returned address: 0x545cc0
[+] RtlAllocateHeap called from 0x74f858eb
[+] HeapHandle: 0x4b0000
[+] Flags: 0x140008
[+] Size: 0x1c
[+] Returned address: 0x506740
```

# Frida – Interceptor at low level (API hook)

The screenshot shows the OllyDbg debugger interface with the assembly window active. The assembly code for the `RtlAllocateHeap` function is displayed. Several jumps (JA, JNZ, JE) are highlighted in yellow, indicating they are being monitored or hooked by Frida. The CPU register row above shows the current state of the registers.

Address	Instruction	Description
76FC209D	Rt!AllocateHeap	
76FC20A2	- E9 92E14F89	JMP 004C0234
76FC20A5	83EC 60	SUB ESP,60
76FC20A6	53	PUSH EBX
76FC20A7	56	PUSH ESI
76FC20A9	33F6	XOR ESI,ESI
76FC20A9	817D 10 FFFFFFF	CMP DWORD PTR SS:[EBP+10],7FFFFFFF
76FC20B0	57	PUSH EDI
76FC20B1	8975 F8	MOV DWORD PTR SS:[EBP-8],ESI
76FC20B4	✓ 0F87 3F900200	JA ntdll.76FEB0F9
76FC20BA	8B5D 08	MOV EBX,DWORD PTR SS:[EBP+8]
76FC20BD	8B43 44	MOV EAX,DWORD PTR DS:[EBX+44]
76FC20C0	8B4B 5C	MOV ECX,DWORD PTR DS:[EBX+5C]
76FC20C3	0945 0C	OR DWORD PTR SS:[EBP+C],EAX
76FC20C6	894D F4	MOV DWORD PTR SS:[EBP-C],ECX
76FC20C9	3BCE	CMP ECX,ESI
76FC20CB	✓ 0F85 DF8F0200	JNZ ntdll.76FEB0B0
76FC20D1	F745 0C 610F81	TEST DWORD PTR SS:[EBP+C],7D810F61
76FC20D8	✓ 0F85 D4130000	JNZ ntdll.76FC34B2
76FC20DE	3975 10	CMP DWORD PTR SS:[EBP+10],ESI
76FC20E1	✓ 0F84 C78A0000	JE ntdll.76FCABAE

# Frida – Interceptor stub

OllyDbg - notepad.exe - [CPU - thread 00000B4C]

C File View Debug Options Window Help

File Edit View Debug Options Window Help

01330234	9C	PUSHFD	
01330235	FC	CLD	
01330236	F0:FF05 240233	LOCK INC DWORD PTR DS:[1330224]	LOCK prefix
01330237	60	PUSHAD	
0133023E	50	PUSH EAX	
0133023F	80B424 20000000	LEA ESI,DWORD PTR SS:[ESP+2C]	
01330246	897424 10	MOV DWORD PTR SS:[ESP+10],ESI	
0133024A	89E6	MOV ESI,ESP	
0133024C	80BC24 28000000	LEA EDI,DWORD PTR SS:[ESP+28]	
01330253	83EC 08	SUB ESP,8	
01330256	57	PUSH EDI	
01330257	56	PUSH ESI	
01330258	68 A04E3201	PUSH 1324EA0	
0133025D	E8 43041164	CALL frida-ag.654406A5	
01330262	83C4 0C	ADD ESP,0C	
01330265	83C4 08	ADD ESP,8	
01330268	85C0	TEST EAX,EAX	
0133026A	2E:74 07	JE SHORT 01330274	Superfluous prefix
0133026D	F0:FF05 240233	LOCK INC DWORD PTR DS:[1330224]	LOCK prefix
01330274	58	POP EAX	
01330275	61	POPAD	
01330276	F0:FF0D 240233	LOCK DEC DWORD PTR DS:[1330224]	LOCK prefix
0133027D	90	POPFD	
0133027E	8BFF	MOV EDI,EDI	
01330280	55	PUSH EBP	
01330281	8BEC	MOV EBP,ESP	
01330283	- E9 1A1EC975	JMP ntdll.76FC20A2	
01330288	CC	INT3	
01330289	50	PUSH EAX	
0133028A	9C	PUSHFD	
0133028B	FC	CLD	
0133028C	60	PUSHAD	
0133028D	50	PUSH EAX	
0133028E	89E6	MOV ESI,ESP	
01330290	80BC24 28000000	LEA EDI,DWORD PTR SS:[ESP+28]	
01330297	89E5	MOV EBP,ESP	
01330299	83EC 0F	SUB ESP,0F	
0133029C	BA F0FFFFFF	MOV EDX,-10	
013302A1	21D4	AND ESP,EDX	
013302A3	83EC 04	SUB ESP,4	
013302A6	57	PUSH EDI	
013302A7	56	PUSH ESI	
013302A8	68 A04E3201	PUSH 1324EA0	
013302AD	E8 1F051164	CALL frida-ag.65440701	
013302B2	83C4 0C	ADD ESP,0C	
013302B5	89EC	MOU ESP,EBP	
013302B7	58	POP EAX	
013302B8	61	POPAD	
013302B9	F0:FF0D 240233	LOCK DEC DWORD PTR DS:[1330224]	LOCK prefix
013302C0	90	POPFD	
013302C1	C3	RETN	
013302C2	0000	ADD BYTE PTR DS:[EAX],AL	
013302C4	0000	ADD BYTE PTR DS:[EAX],AL	

# Frida – Stalker

- Stalker example:

```
10  function StalkerExample()
11  {
12      threadIds = [];
13
14      Process.enumerateThreads({
15          onMatch: function (thread)
16          {
17              threadIds.push(thread.id);
18              console.log("Thread ID: " + thread.id.toString());
19          },
20
21          onComplete: function ()
22          {
23              threadIds.forEach(function (threadId)
24              {
25                  Stalker.follow(threadId,
26                  {
27                      events: {call: true},
28
29                      onReceive: function (events)
30                      {
31                          console.log("onReceive called.");
32                      },
33                      onCallSummary: function (summary)
34                      {
35                          console.log("onCallSummary called.");
36                      }
37                  });
38              });
39          }
40      });
41  }
42
43  StalkerExample();
```

# Frida – How Stalker works?

- Stalker at low level:

```
00001000 push ebp  
00001001 mov ebp, esp  
00001003 call 00001234  
00001008 mov esp, ebp  
0000100A pop ebp  
0000100B ret
```



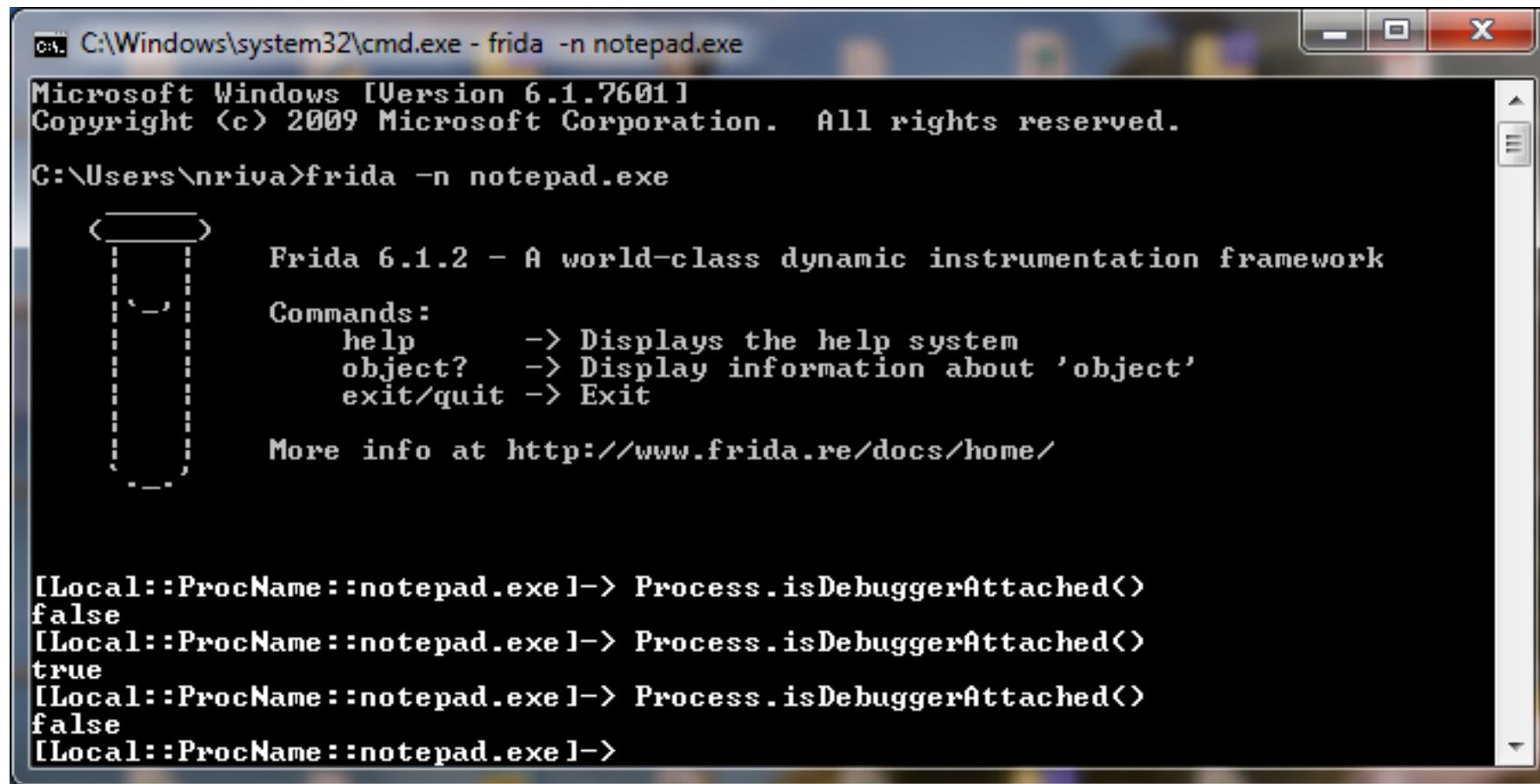
```
00004000 call log_handler  
00004005 push ebp  
00004006 call log_handler  
0000400B mov ebp, esp  
0000400D call log_handler  
00004012 push 00001008 ; CALL stack side-effect  
00004013 push 00001234 ; arg 2/2: branch target  
00004014 push exec_ctx ; arg 1/2: execution context  
00004019 call gum_exec_ctx_replace_current_block_with
```

- Hint: See `gum_exec_ctx_obtain_block_for` in `frida-gum/gum/backend-x86/gumstalker-x86.c`

# Tools based on Frida

# Tools based on Frida

- **frida-cli**: command line interpreter which emulates an IPython console for rapid prototyping and easy debugging.



The screenshot shows a Windows command prompt window titled 'cmd C:\Windows\system32\cmd.exe - frida -n notepad.exe'. The window displays the Frida 6.1.2 command-line interface, which includes a help system, object inspection, and exit/quit commands. At the bottom of the interface, it shows the result of running the 'Process.isDebuggerAttached()' method on the 'notepad.exe' process, which returns both 'false' and 'true' in separate runs, indicating a debugger attachment state change.

```
C:\Windows\system32\cmd.exe - frida -n notepad.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\nriva>frida -n notepad.exe

Frida 6.1.2 - A world-class dynamic instrumentation framework

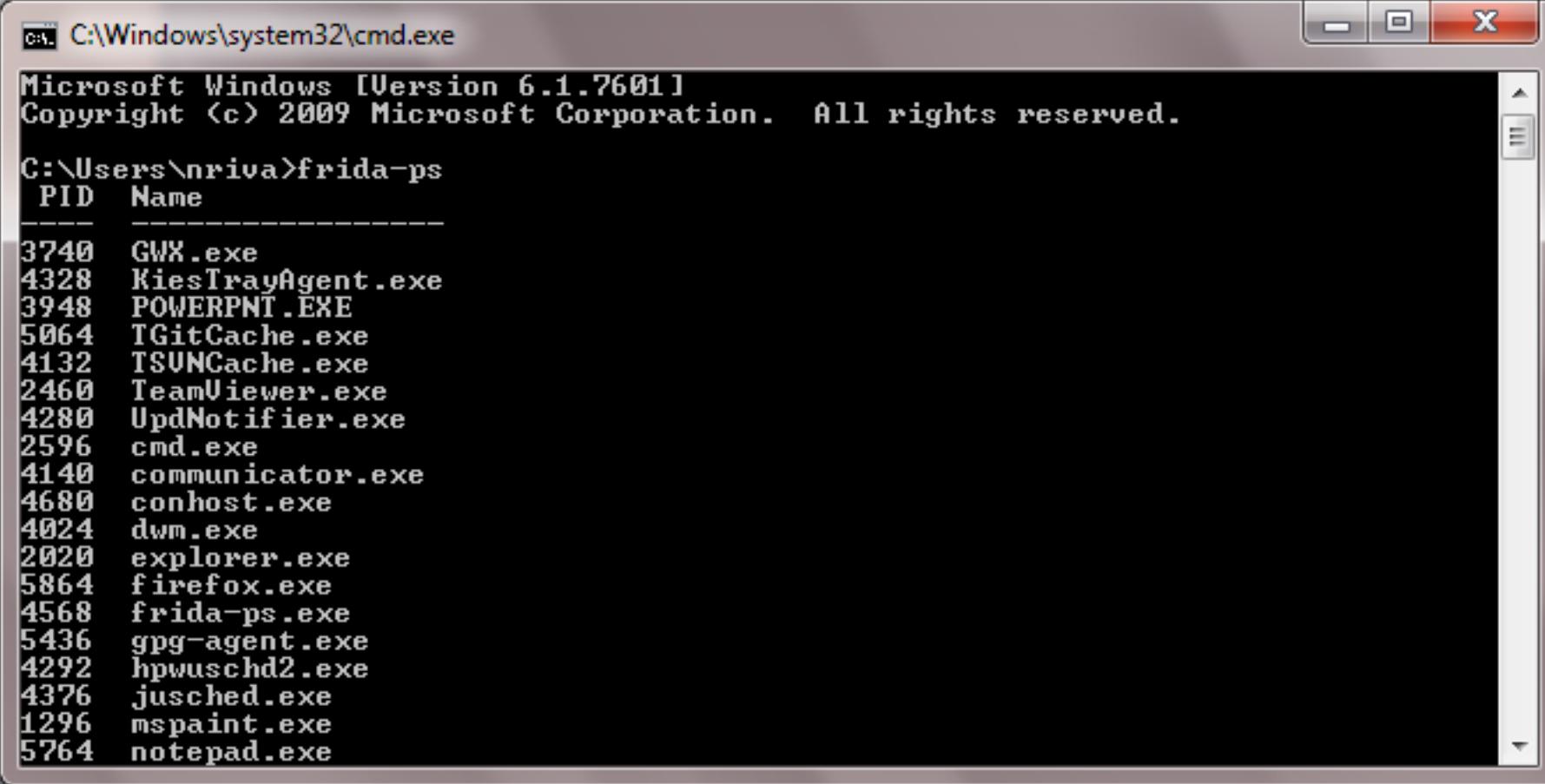
Commands:
  help      -> Displays the help system
  object?   -> Display information about 'object'
  exit/quit -> Exit

More info at http://www.frida.re/docs/home/

[Local::ProcName::notepad.exe]-> Process.isDebuggerAttached()
false
[Local::ProcName::notepad.exe]-> Process.isDebuggerAttached()
true
[Local::ProcName::notepad.exe]-> Process.isDebuggerAttached()
false
[Local::ProcName::notepad.exe]->
```

# Tools based on Frida

- **frida-ps**: command line tool for listing processes.

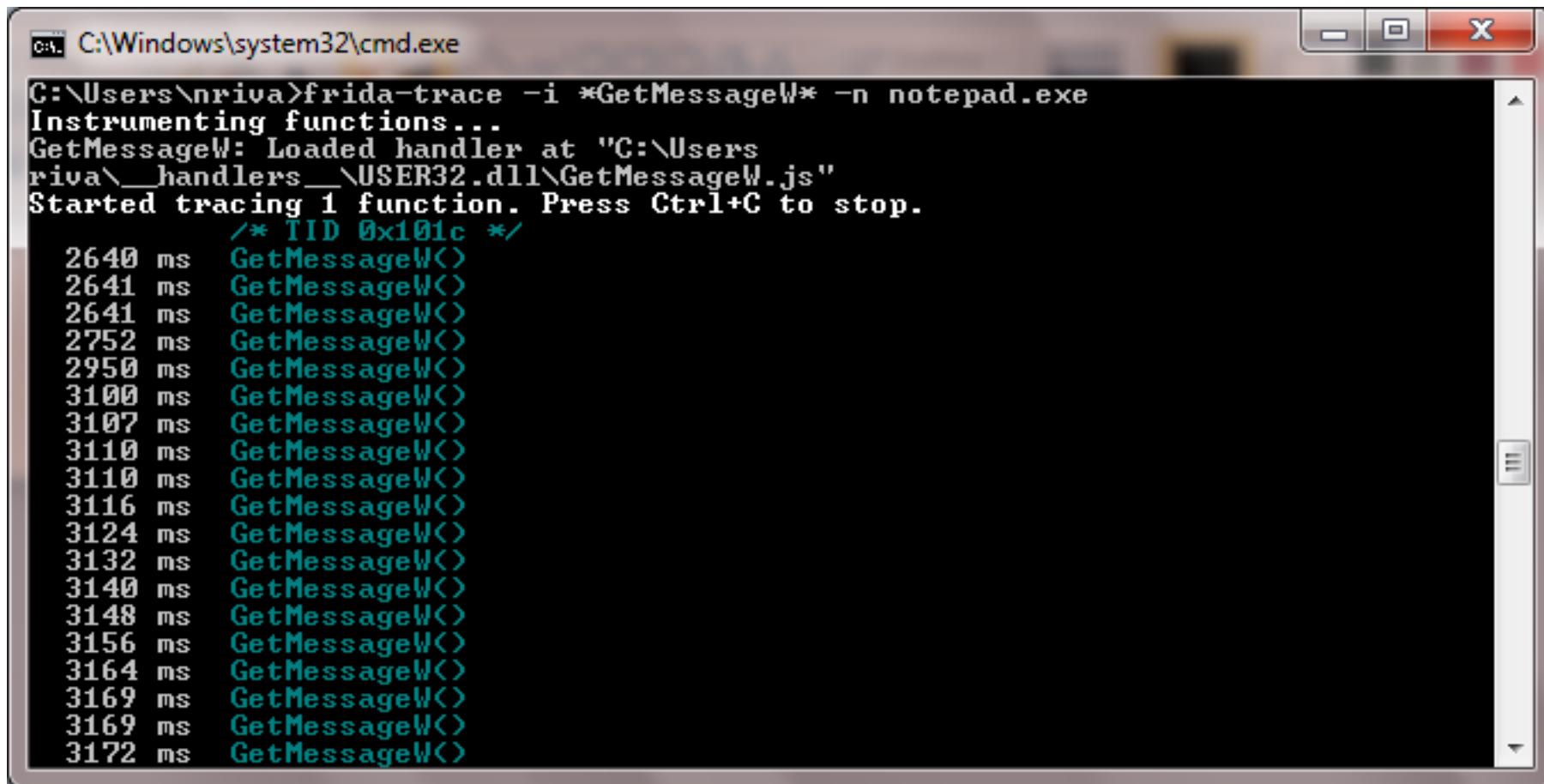


The screenshot shows a Windows command prompt window titled "cmd C:\Windows\system32\cmd.exe". The window displays the output of the "frida-ps" command, which lists processes by PID and name. The output is as follows:

PID	Name
3740	GWX.exe
4328	KiesTrayAgent.exe
3948	POWERPNT.EXE
5064	TGitCache.exe
4132	TSUNCache.exe
2460	TeamViewer.exe
4280	UpdNotifier.exe
2596	cmd.exe
4140	communicator.exe
4680	conhost.exe
4024	dwm.exe
2020	explorer.exe
5864	firefox.exe
4568	frida-ps.exe
5436	gpg-agent.exe
4292	hpwuschd2.exe
4376	jusched.exe
1296	mspaint.exe
5764	notepad.exe

# Tools based on Frida

- **frida-trace**: command like tool to dynamically trace function calls.



The screenshot shows a Windows Command Prompt window titled 'cmd' with the path 'C:\Windows\system32\cmd.exe'. The command entered is 'C:\Users\nriva>frida-trace -i \*GetMessageW\* -n notepad.exe'. The output indicates that the 'Instrumenting functions...' step has completed, and a handler for the 'GetMessageW' function has been loaded from 'C:\Users\nriva\\_handlers\_\USER32.dll\GetMessageW.js'. It then states 'Started tracing 1 function. Press Ctrl+C to stop.' Below this, a list of 26 function calls is shown, each with a timestamp (e.g., 2640 ms, 2641 ms, etc.) and the function name 'GetMessageW()'.

```
C:\Users\nriva>frida-trace -i *GetMessageW* -n notepad.exe
Instrumenting functions...
GetMessageW: Loaded handler at "C:\Users\nriva\_handlers_\USER32.dll\GetMessageW.js"
Started tracing 1 function. Press Ctrl+C to stop.

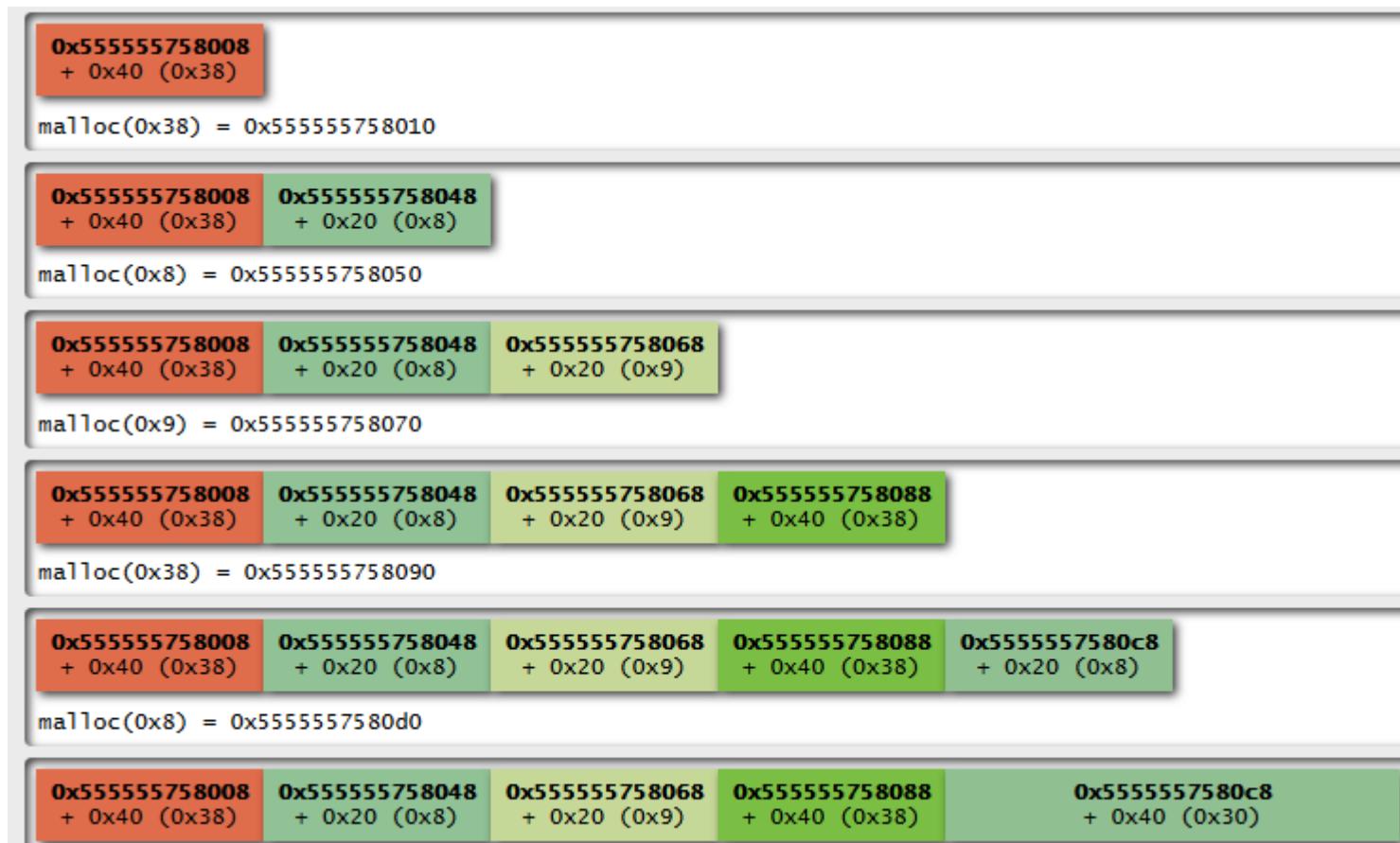
    /* TID 0x101c */
2640 ms  GetMessageW()
2641 ms  GetMessageW()
2641 ms  GetMessageW()
2752 ms  GetMessageW()
2950 ms  GetMessageW()
3100 ms  GetMessageW()
3107 ms  GetMessageW()
3110 ms  GetMessageW()
3110 ms  GetMessageW()
3116 ms  GetMessageW()
3124 ms  GetMessageW()
3132 ms  GetMessageW()
3140 ms  GetMessageW()
3148 ms  GetMessageW()
3156 ms  GetMessageW()
3164 ms  GetMessageW()
3169 ms  GetMessageW()
3169 ms  GetMessageW()
3172 ms  GetMessageW()
```

# Tools based on Frida

- **frida-heap-trace**: trace RtlAllocateHeap, RtlFreeHeap and RtlReAllocateHeap function calls and arguments and log them to a file.
- Combine it with Villoc to create a map for all the heap movements
- <https://github.com/poxyran/misc/blob/master/frida-heap-trace.py>

# Tools based on Frida

- <https://github.com/wapiflapi/villoc>

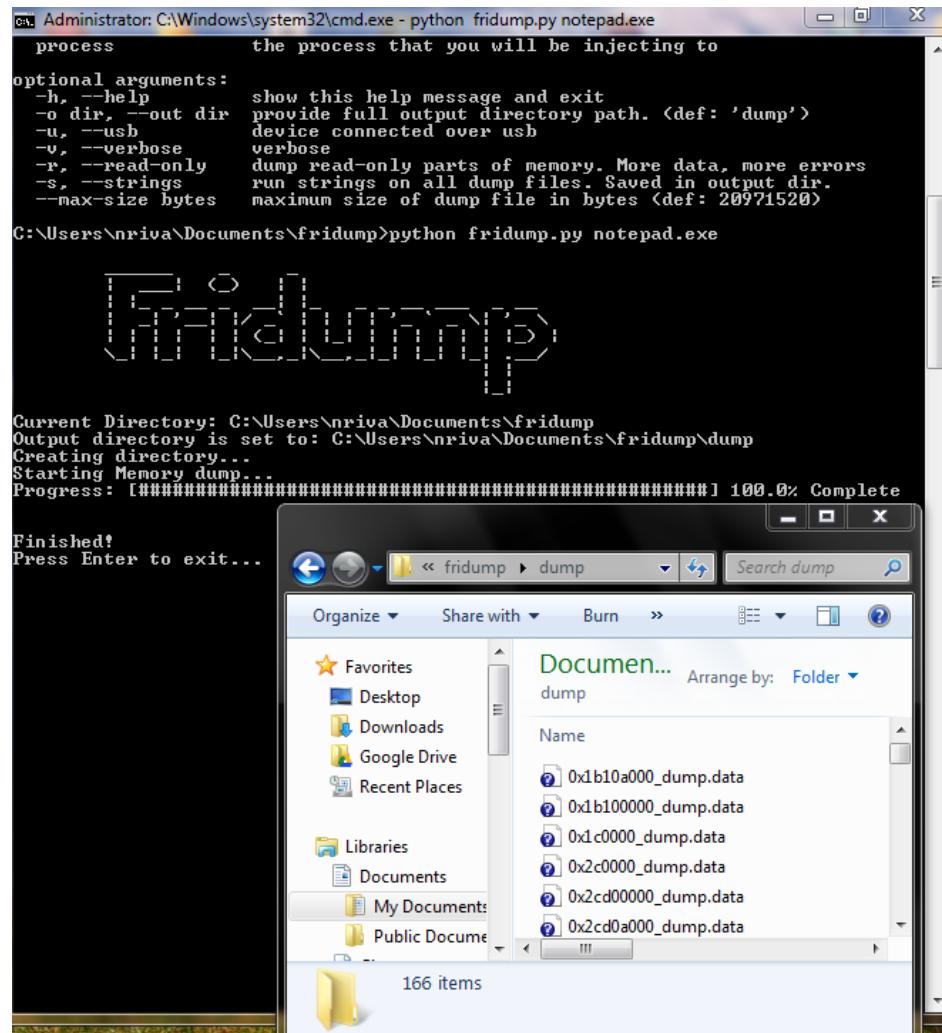


# Tools based on Frida

- **fridump**: Universal memory dumper tool. Aimed to dump accessible memory regions from any platform supported by Frida.

<https://github.com/Nightbringer21/fridump>

# Tools based on Frida



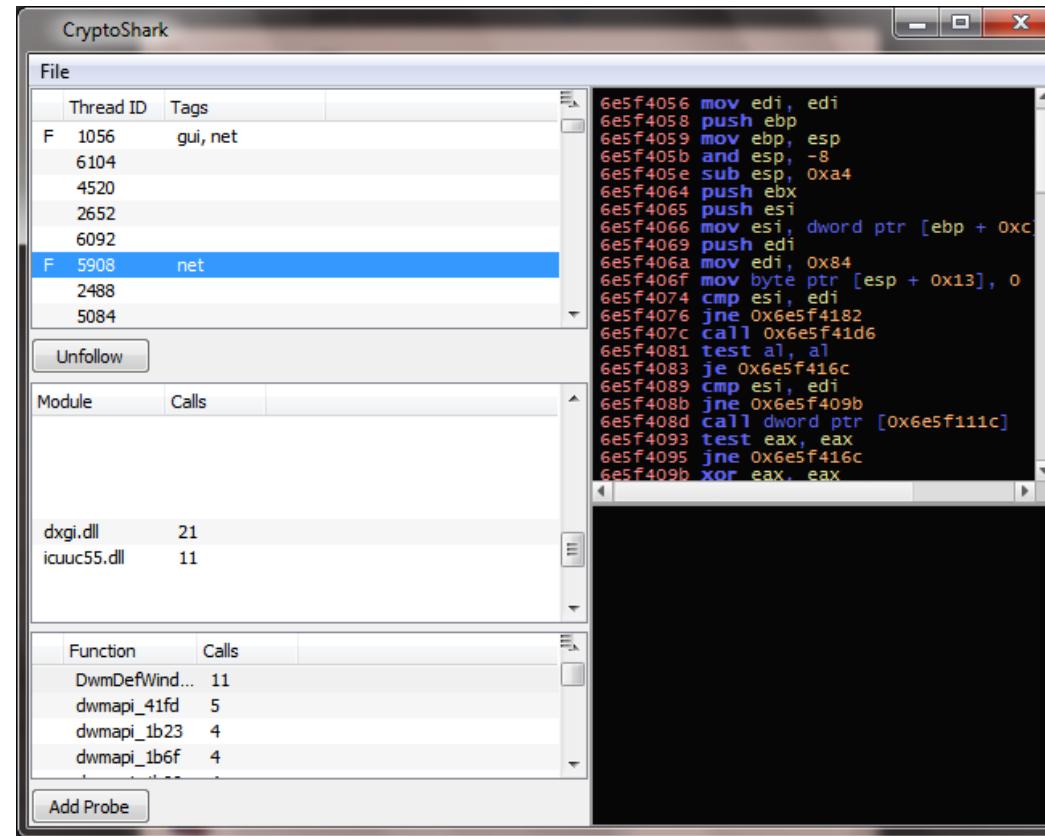
# Tools based on Frida

- **frida-extract:** FridaExtract is a [Frida.re](#) based [RunPE](#) extraction tool. Using FridaExtract you can automatically extract and reconstruct a PE file that has been injected using the RunPE method.

<https://github.com/OALabs/frida-extract>

# Tools based on Frida

- **frida-discover**: tool for discovering internal functions in a program. Eg: Cryptoshark: <https://github.com/frida/cryptoshark>



# Tools based on Frida

**Cryptoshark** and **frida-discover** are based on Frida's Stalker API.

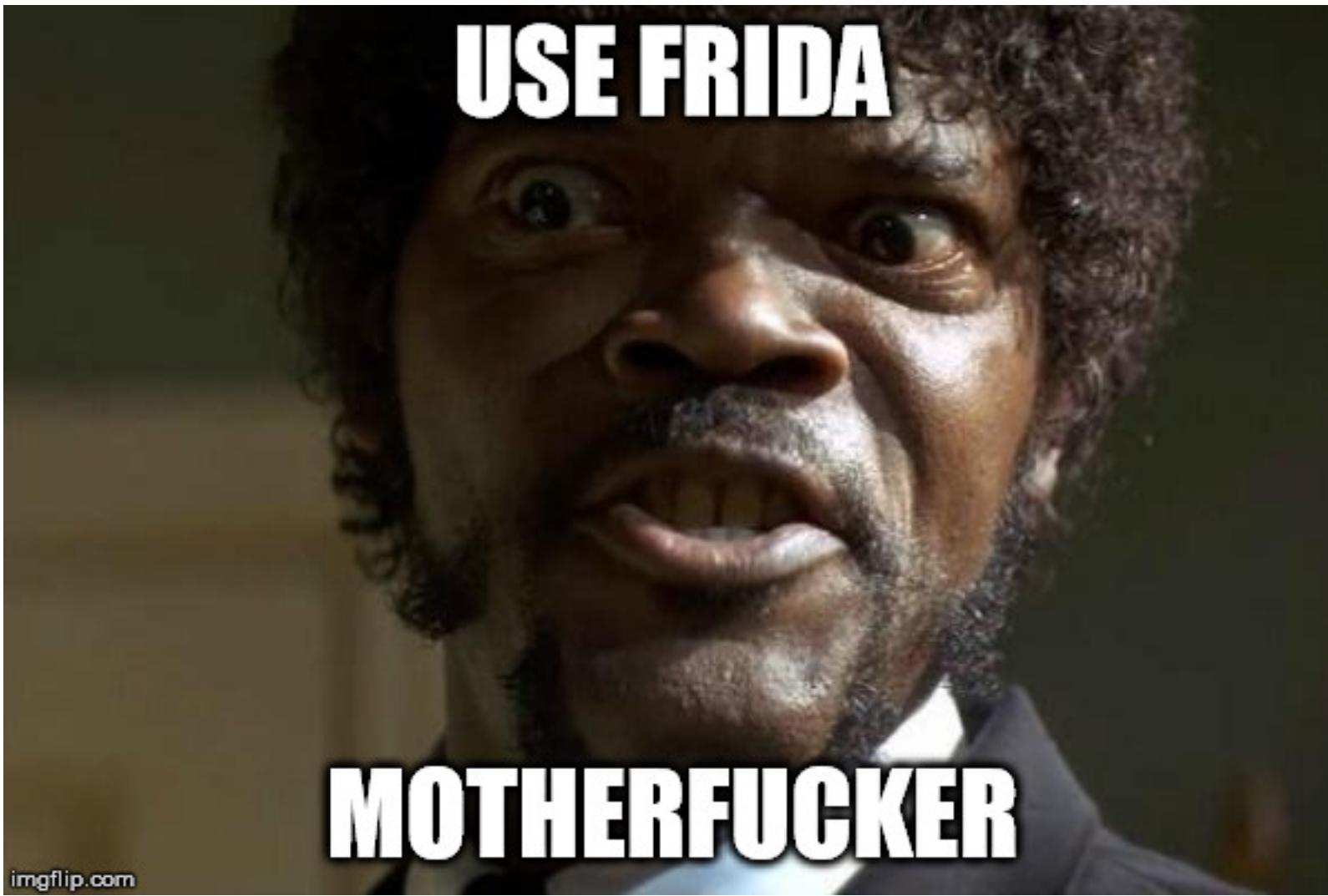
They dynamically instrument every thread in a given process and stalk every called function during process execution trying to discover internal functions like statically linked functions.

# Conclusions

# Conclusions

- When to use **Frida**? If you ...
  - Don't want to download a compiler and compile every time you make a change
  - Need to **quickly** write an introspection tool
  - Need low granularity (this may change in the near future)
  - Need multi-OS/Arch support
- Then ...

# Conclusions



imgflip.com

# Additional information

# Additional information

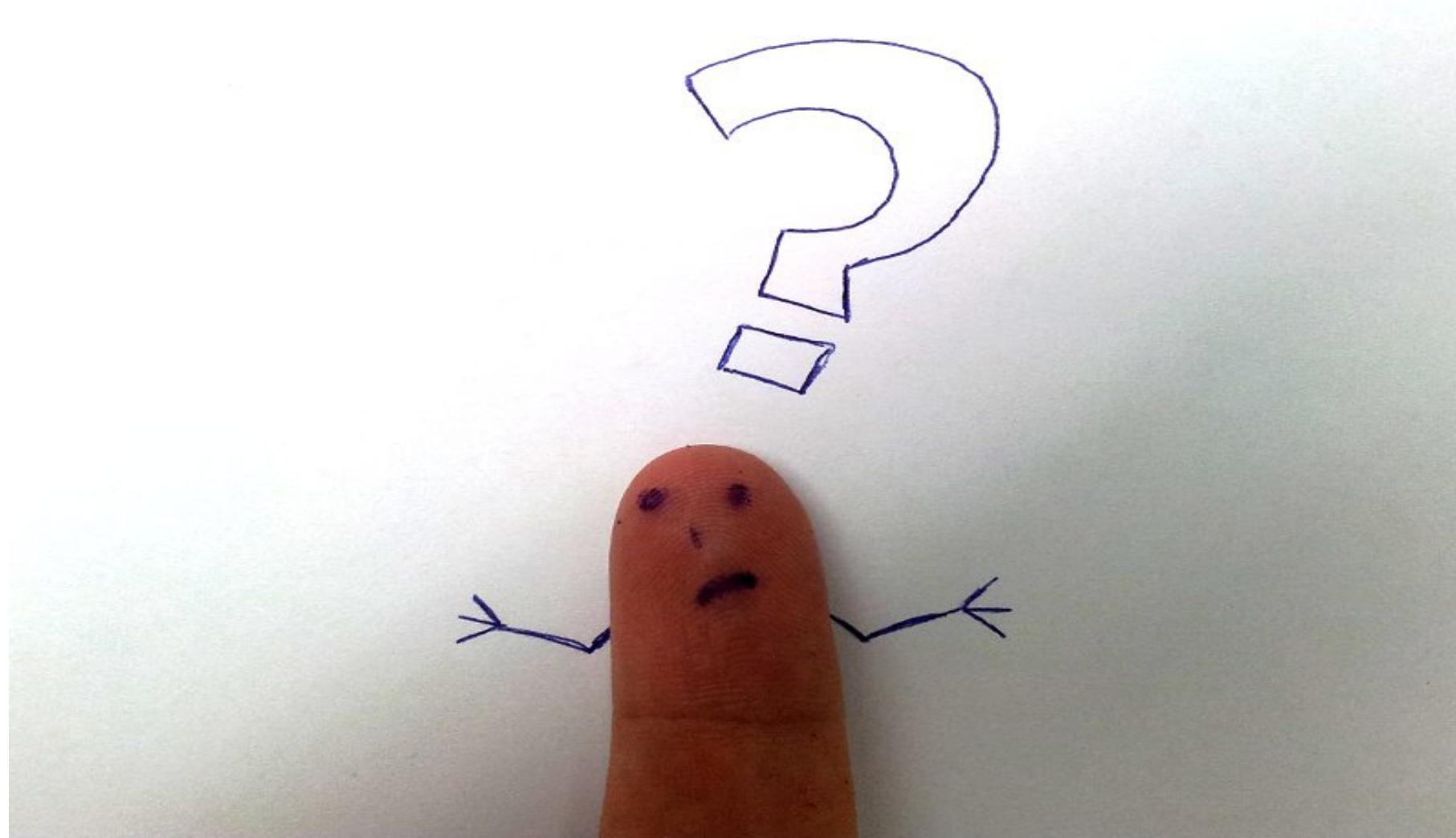
- **Questions to:**
  - <https://twitter.com/oleavr>
- **Frida news and docs:**
  - <http://www.frida.re/>
- **Frida source code:**
  - <https://github.com/frida>
- **Frida resources:**
  - <https://github.com/dweinstein/awesome-frida>

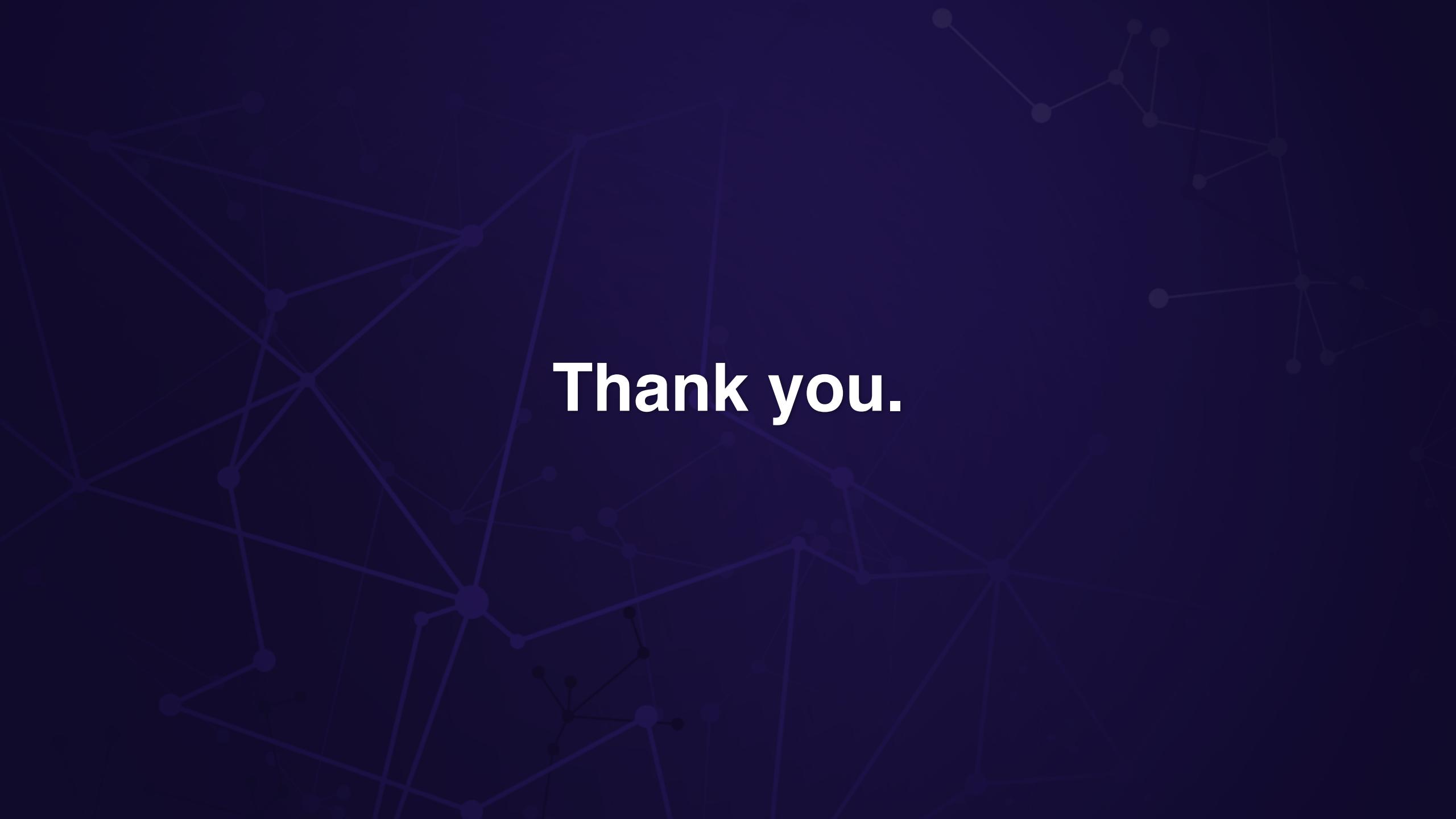
# Acknowledgments & Greetings

# Acknowledgments & Greetings

- Ole André V. Ravnås
  - For answering all my question about Frida
- Francisco Falcón
  - For the feedback about this presentation

# Questions?





Thank you.