IDA Pro - Appcall user guide

*Copyright 2023 Hex-Rays SA*

Contents

[Introduction 2](#_Toc139709089)

[Quick start 2](#_Toc139709090)

[Using AppCall with IDC 3](#_Toc139709091)

[Passing arguments by reference 5](#_Toc139709092)

[\_\_usercall calling convention 6](#_Toc139709093)

[Variadic functions 6](#_Toc139709094)

[Calling functions that might cause exceptions 7](#_Toc139709095)

[Functions that accept or return structures 7](#_Toc139709096)

[Calling an API that receives a structure and its size 9](#_Toc139709097)

[Working with opaque types 10](#_Toc139709098)

[FindFirst/FindNext APIs example 11](#_Toc139709099)

[Using LoadLibrary/GetProcAddress 12](#_Toc139709100)

[Retrieving application's command line 13](#_Toc139709101)

[Specifying Appcall options 13](#_Toc139709102)

[Manual Appcall 14](#_Toc139709103)

[Initiating multiple manual Appcalls 15](#_Toc139709104)

[Capturing exception debug events 16](#_Toc139709105)

[Appcall related functions 16](#_Toc139709106)

[parse\_decl/get\_tinfo/sizeof 16](#_Toc139709107)

[Accessing enum members as constants 17](#_Toc139709108)

[Storing/Retrieving typed elements 18](#_Toc139709109)

[Using Appcall with IDAPython 19](#_Toc139709110)

[Passing arguments by reference 21](#_Toc139709111)

[Functions that accept or return structures 22](#_Toc139709112)

[FindFirst/FindNext example 23](#_Toc139709113)

[Using GetProcAddress 25](#_Toc139709114)

[Setting the Appcall options 25](#_Toc139709115)

[Calling functions that can cause exceptions 26](#_Toc139709116)

[Appcall related functions in Python 26](#_Toc139709117)

[Storing/Retrieving objects 26](#_Toc139709118)

[Accessing enum members as constants 28](#_Toc139709119)

# Introduction

Appcall is a mechanism to call functions under a debugger session in the context of the debugged program using IDA's CLI (Command Line Interpreter) or from a script.  
Such a mechanism can be used for seamless blackbox function calling and use, fuzzing, process instrumentation, DLL injection, testing or extending applications.

Appcall mechanism highly depends on the type information of the called function. For that reason, it is necessary to have a correct function prototype before doing an Appcall, otherwise different or incorrect results may be returned.

In a nutshell, Appcall works by first hijacking the current thread's stack (please do switch threads explicitly if you want to Appcall in a different context), then pushing the arguments, and then temporarily adjusting the instruction pointer to the beginning of the called function. After the function returns (or an exception occurs), the original stack, instruction pointer, and other registers are restored, and the result is returned to the caller.

Please note that while most of the examples in this document are illustrated using a Windows user mode application, Appcall is not limited to Windows and can be used with any platform supported by IDA debuggers.

## Quick start

Let's start explaining the basic concepts of Appcall using the IDC CLI. Let's imagine we have the following printf() in the disassembly somewhere:

.text:00000001400015C0 ; \_\_int64 printf(const char \*, ...)

.text:00000001400015C0 \_printf proc near

.text:00000001400015C0

.text:00000001400015C0

.text:00000001400015C0 arg\_0 = qword ptr 8

.text:00000001400015C0 arg\_8 = qword ptr 10h

.text:00000001400015C0 arg\_10 = qword ptr 18h

.text:00000001400015C0 arg\_18 = qword ptr 20h

.text:00000001400015C0

.text:00000001400015C0 mov [rsp+arg\_0], rcx

.text:00000001400015C5 mov [rsp+arg\_8], rdx

.text:00000001400015CA mov [rsp+arg\_10], r8

.text:00000001400015CF mov [rsp+arg\_18], r9

...

It can be called by simply typing the following in the IDC CLI (press "." to jump to the CLI):

\_printf("hello world\n");

As you noticed, we invoked an Appcall by simply treating \_printf as if it was a built-in IDC function. If the application had a console window, then you should see the message printed in it.

If you have a function with a mangled name or with characters that cannot be used as an identifier name in the IDC language, such as "\_my\_func@8", then you can use the LocByName function to get its address given its name, then using the address variable (which is callable) we issue the Appcall:

auto myfunc = LocByName("\_my\_func@8");

myfunc("hello", "world");

Or simply directly as:

LocByName("\_my\_func@8")("hello", "world");

# Using AppCall with IDC

Apart from calling Appcall naturally as shown in the previous section, it is possible to call it explicitly using the dbg\_appcall function:

// Call application function

// ea - address to call

// type - type of the function to call. can be specified as:

// - declaration string. example: "int func(void);"

// - typeinfo object. example: get\_tinfo(ea)

// - zero: the type will be retrieved from the idb

// ... - arguments of the function to call

// Returns: the result of the function call

// If the call fails because of an access violation or other exception,

// a runtime error will be generated (it can be caught with try/catch)

// In fact there is rarely any need to call this function explicitly.

// IDC tries to resolve any unknown function name using the application labels

// and in the case of success, will call the function. For example:

// \_printf("hello\n")

// will call the application function \_printf provided that there is

// no IDC function with the same name.

anyvalue dbg\_appcall(ea, type, ...);

The Appcall IDC function requires you to pass a function address, function type information (various forms are accepted) and the parameters (if any):

auto msgbox;

msgbox = LocByName("\_\_imp\_MessageBoxA");

// Pass "0" for the type to deduce it from the database

dbg\_appcall(msgbox, 0, 0, "Hello world", "Info", 0);

We've seen so far how to call a function if it already has type information, now suppose we have a function that does not:

user32.dll:00007FFF3AD730F0 user32\_FindWindowA proc near

user32.dll:00007FFF3AD730F0 mov r9, rdx

user32.dll:00007FFF3AD730F3 mov r8, rcx

user32.dll:00007FFF3AD730F6 xor edx, edx

user32.dll:00007FFF3AD730F8 xor ecx, ecx

user32.dll:00007FFF3AD730FA jmp sub\_7FFF3ADC326C

user32.dll:00007FFF3AD730FA user32\_FindWindowA endp

Before calling this function with dbg\_appcall we have two options:

1. Pass the prototype as a string
2. Or, parse the prototype separately and pass the returned type info object.

This is how we can do it using the first option:

auto window\_handle;

window\_handle = dbg\_appcall(

LocByName("user32\_FindWindowA"),

"long \_\_stdcall FindWindow(const char \*cls, const char \*wndname)",

0,

"Calculator");

msg("handle=%d\n", window\_handle);

As for the second option, we can use parse\_decl() first, then proceed as usual:

auto window\_handle, tif;

tif = parse\_decl("long \_\_stdcall FindWindow(const char \*cls, const char \*wndname)", 0);

window\_handle = dbg\_appcall(

LocByName("user32\_FindWindowA"),

tif,

0,

"Calculator");

msg("handle=%d\n", window\_handle);

*Note that we used parse\_decl() function to construct a typeinfo object that we can pass to dbg\_appcall*.

It is possible to permanently set the prototype of a function programmatically using apply\_type():

auto tif;

tif = parse\_decl("long \_\_stdcall FindWindow(const char \*cls, const char \*wndname)", 0);

apply\_type(

LocByName("user32\_FindWindowA"),

tif);

In the following sections, we are going to cover different scenarios such as calling by reference, working with buffers and complex structures, etc.

## Passing arguments by reference

To pass function arguments by reference, it suffices to use the **&** symbol as in the C language.

* For example to call this function:

void ref1(int \*a)

{

if (a == NULL)

return;

int o = \*a;

int n = o + 1;

\*a = n;

printf("called with %d and returning %d\n", o, n);

}

We can use this code from IDC:

auto a = 5;

msg("a=%d", a);

ref1(**&**a);

msg(", after the call=%d\n", a);

* To call a C function that takes a string buffer and modifies it:

/\* C code \*/

int ref2(char \*buf)

{

if (buf == NULL)

return -1;

printf("called with: %s\n", buf);

char \*p = buf + strlen(buf);

\*p++ = '.';

\*p = '\0';

printf("returned with: %s\n", buf);

int n=0;

for (;p!=buf;p--)

n += \*p;

return n;

}

We need to create a buffer and pass it by reference to the function:

auto s = strfill('\x00', 20); // create a buffer of 20 characters

s[0:5] = "hello"; // initialize the buffer

ref2(&s); // call the function and pass the string by reference

// check if the string has a dot appended to it

if (s[5] != ".")

{

msg("not dot\n");

}

else

{

msg("dot\n");

}

## \_\_usercall calling convention

It is possible to Appcall functions with non standard calling conventions, such as routines written in assembler that expect parameters in various registers and so on. One way is to use the **\_\_usercall** calling convention.

Consider this function:

/\* C code \*/

// eax = esi - edi

int \_\_declspec(naked) asm1()

{

\_\_asm

{

mov eax, esi

sub eax, edi

ret

}

}

And from IDC:

auto x = dbg\_appcall(

LocByName("asm1"),

"int \_\_usercall asm1@<eax>(int a@<edi>, int b@<esi>);",

1,

5);

msg("result = %d\n", x);

## Variadic functions

In C:

int va\_altsum(int n1, ...)

{

va\_list va;

va\_start(va, n1);

int r = n1;

int alt = 1;

while ( (n1 = va\_arg(va, int)) != 0 )

{

r += n1\*alt;

alt \*= -1;

}

va\_end(va);

return r;

}

And in IDC:

auto result = va\_altsum(5, 4, 2, 1, 6, 9, 0);

## Calling functions that might cause exceptions

Exceptions may occur during an Appcall. To capture them, use the try/catch mechanism:

auto e;

try

{

dbg\_appcall(some\_func\_addr, func\_type, args...);

// Or equally:

// some\_func\_name(arg1, arg2, ...);

}

catch (e)

{

// Exception occured .....

}

The exception object "e" will be populated with the following fields:

* description: description text generated by the debugger module while it was executing the Appcall
* file: The name of the file where the exception happened.
* func: The IDC function name where the exception happened.
* line: The line number in the script
* qerrno: The internal code of last error occurred

For example, one could get something like this:

description: "Appcall: The instruction at 0x401012 referenced memory at 0x0. The memory could not be written"

file: ""

func: "\_\_\_idcexec0"

line: 4. 4h 4o

qerrno: 92. 5Ch 134o

In some cases, the exception object will contain more information.

## Functions that accept or return structures

Appcall mechanism also works with functions that accept or return structure types. Consider this C code:

#pragma pack(push, 1)

struct UserRecord

{

int id;

char name[50];

struct UserRecord\* next;

};

#pragma pack(pop)

// Function to create a new record

UserRecord \*makeRecord(char name[], int id)

{

UserRecord\* newRecord = (UserRecord\*)malloc(sizeof(UserRecord));

strcpy(newRecord->name, name);

newRecord->id = id;

newRecord->next = NULL;

return newRecord;

}

void printRecord(UserRecord\* record)

{

printf("Id: %d ; Name: %s\n", record->id, record->name);

}

// Function to list all student records in the linked list

void listRecords(UserRecord\* head)

{

if (head == NULL)

{

printf("No records found.\n");

return;

}

printf("Records:\n"

"--------\n");

while (head != NULL)

{

printRecord(head);

head = head->next;

}

}

We can create a couple of records and link them up together:

auto rec1, rec2, rec3;

// Create records

rec1 = makeRecord("user1", 1);

rec2 = makeRecord("user2", 2);

rec3 = makeRecord("user3", 3);

// Link them up

rec1.next = rec2;

rec2.next = rec3;

// Display them

listRecords(rec1);

Because we issued an Appcall, when listRecords is called, we expect to see the following output in the console:

Records:

--------

Id: 1 ; Name: user1

Id: 2 ; Name: user2

Id: 3 ; Name: user3

We can then access the fields naturally (even the linked objects). We can verify that if we just dump the first record through the IDC CLI (or just by calling IDC's print function):

IDC>rec1

object

id: 1. 1h 1o

name: "user1\x00"

next: object

id: 2. 2h 2o

name: "user2\x00"

next: object

id: 3. 3h 3o

name: "user3\x00"

next: 0x0i64

Notice how when rec1 is dumped, its next field is automatically followed and properly displayed. The same happens for rec2 and rec3.

We can also directly access the fields of the structure from IDC and have those changes reflected in the debugee's memory:

rec1.id = 11;

rec1.name = "hey user1";

rec1.next.name = "hey user2";

rec1.next.id = 21;

rec1.next.next.name = "hey user3";

rec1.next.next.id = 31;

// Display them

listRecords(rec1);

Notable observations:

* Objects are always passed by reference (no need to use the **&**)
* Objects are created on the stack
* Objects are untyped
* Missing object fields are automatically created by IDA and filled with zero

## Calling an API that receives a structure and its size

Let us take another example where we call the GetVersionExA API function:

kernel32.dll:00007FFF3A0F9240 kernel32\_GetVersionExA proc near

kernel32.dll:00007FFF3A0F9240 jmp cs:off\_7FFF3A1645E0

kernel32.dll:00007FFF3A0F9240 kernel32\_GetVersionExA endp

This API requires one of its input fields to be initialized to the size of the structure. Therefore, we need to initialize the structure correctly before passing it to the API to be further populated therein:

// Create an empty object

auto ver = object();

// We need to initialize the size of the structure

ver.dwOSVersionInfoSize = sizeof("OSVERSIONINFO");

// This is the only field we need to have initialized, the other fields will be created by IDA and filled with zeroes

// Now issue the Appcall:

GetVersionExA(ver);

msg("%d.%d (%d)\n", ver.dwMajorVersion, ver.dwMinorVersion, ver.dwBuildNumber);

Now if we dump the **ver** object contents we observe something like this:

IDC>print(ver);

object

dwBuildNumber: 9200. 23F0h 21760o

dwMajorVersion: 6. 6h 6o

dwMinorVersion: 2. 2h 2o

dwOSVersionInfoSize: 148. 94h 224o

dwPlatformId: 2. 2h 2o

szCSDVersion: "\x00\x00\x00\x00\x00\x00...."

## Working with opaque types

Opaque types like FILE, HWND, HANDLE, HINSTANCE, HKEY, etc. are not meant to be used as structures by themselves but like pointers.

Let us take for example the FILE structure that is used with fopen(); its underlying structure looks like this (implementation details might change):

00000000 FILE struc ; (sizeof=0x18, standard type)

00000000 curp dd ?

00000004 buffer dd ?

00000008 level dd ?

0000000C bsize dd ?

00000010 istemp dw ?

00000012 flags dw ?

00000014 hold dw ?

00000016 fd db ?

00000017 token db ?

00000018 FILE ends

And the fopen() function prototype is:

msvcrt.dll:00007FFF39F1B7B0 ; FILE \*\_\_cdecl fopen(const char \*FileName, const char \*Mode)

msvcrt.dll:00007FFF39F1B7B0 fopen proc near

msvcrt.dll:00007FFF39F1B7B0 mov r8d, 40h ; '@'

msvcrt.dll:00007FFF39F1B7B6 jmp msvcrt\_\_fsopen

msvcrt.dll:00007FFF39F1B7B6 fopen endp

Let us see how we can get a "FILE \*"" and use it as an opaque type and issue an fclose() call properly:

auto fp;

fp = fopen("c:\\temp\\x.cpp", "r");

print(fp);

fclose(fp.\_\_at\_\_);

Nothing special about the fopen/fclose Appcalls except that we see the **\_\_at\_\_** attribute showing up although it does not belong to the FILE structure definition.  
This is a special attribute that IDA inserts into all objects, and it contains the memory address from which IDA retrieved the object attribute values. We can use the **\_\_at\_\_** to retrieve the C pointer of a given IDC object.

Previously, we omitted the **\_\_at\_\_** field from displaying when we dumped objects output, but in reality this is what one expects to see as part of the objects attributes used in Appcalls. Let's create a user record again:

auto rec;

rec1 = makeRecord("user1", 13);

rec2 = makeRecord("user2", 14);

rec1.next = rec2;

print(rec1);

..and observe the output:

object

\_\_at\_\_: 5252736. 502680h 24023200o

id: 13. Dh 15o

name: "user1\x00..."

next: object

\_\_at\_\_: 5252848. 5026F0h 24023360o

id: 14. Eh 16o

name: "user2\x00..."

next: 0x0i64

Please note that it is possible to pass as integer (which is a pointer) to a function that expects a pointer to a structure.

## FindFirst/FindNext APIs example

In this example, we call the APIs directly without permanently setting their prototype first.

static main()

{

auto fd, h, n, ok;

fd = object(); // create an object

h = dbg\_appcall(

LocByName("kernel32\_FindFirstFileA"),

"HANDLE \_\_stdcall FindFirstFileA(LPCSTR lpFileName, LPWIN32\_FIND\_DATAA lpFindFileData);",

"c:\\windows\\\*.exe",

fd);

if (h == -1) // INVALID\_HANDLE\_VALUE

{

msg("No files found!\n");

return -1;

}

for (n=1;;n++)

{

msg("Found: %s\n", fd.cFileName);

ok = dbg\_appcall(

LocByName("kernel32\_FindNextFileA"),

"BOOL \_\_stdcall FindNextFileA(HANDLE hFindFile, LPWIN32\_FIND\_DATAA lpFindFileData);",

h,

fd);

if ( (n > 5) || (ok == 0) )

break;

}

dbg\_appcall(

LocByName("kernel32\_FindClose"),

"BOOL \_\_stdcall FindClose(HANDLE hFindFile);",

h);

return n;

}

## Using LoadLibrary/GetProcAddress

In this example, we are going to initialize the APIs by setting up their prototypes correctly so we can use them later conveniently.

extern getmodulehandle, getprocaddr, findwindow, loadlib;

static init\_api()

{

loadlib = LocByName("kernel32\_LoadLibraryA");

getmodulehandle = LocByName("kernel32\_GetModuleHandleA");

getprocaddr = LocByName("kernel32\_GetProcAddress");

if (loadlib == BADADDR || getmodulehandle == BADADDR || getprocaddr == BADADDR)

return "Failed to locate required APIs";

// Let us permanently set the prototypes of these functions

apply\_type(loadlib, "HMODULE \_\_stdcall loadlib(LPCSTR lpModuleName);");

apply\_type(getmodulehandle, "HMODULE \_\_stdcall gmh(LPCSTR lpModuleName);");

apply\_type(getprocaddr, "FARPROC \_\_stdcall gpa(HMODULE hModule, LPCSTR lpProcName);");

// Resolve address of FindWindow api

auto t = getmodulehandle("user32.dll");

if (t == 0)

{

t = loadlib("user32.dll");

if (t == 0)

return "User32 is not loaded!";

}

findwindow = getprocaddr(t, "FindWindowA");

if (findwindow == 0)

return "FindWindowA API not found!";

// Set type

apply\_type(findwindow, "HWND \_\_stdcall FindWindowA(LPCSTR lpClassName, LPCSTR lpWindowName);");

return "ok";

}

static main()

{

auto ok = init\_api();

if (ok != "ok")

{

msg("Failed to initialize: %s", ok);

return -1;

}

auto hwnd = dbg\_appcall(findwindow, 0, 0, "Calculator");

if (hwnd == 0)

{

msg("Failed to locate the Calculator window!\n");

return -1;

}

msg("Calculator hwnd=%x\n", hwnd);

return 0;

}

## Retrieving application's command line

extern getcommandline;

static main()

{

getcommandline = LocByName("kernel32\_GetCommandLineA");

if (getcommandline == BADADDR)

{

msg("Failed to resolve GetCommandLineA API address!\n");

return -1;

}

apply\_type(getcommandline, "const char \*\_\_stdcall GetCommandLineA();");

msg("This application's command line:<\n%s\n>\n", getcommandline());

return 0;

}

## Specifying Appcall options

Appcall can be configured with set\_appcall\_options() and passing one or more options:

* APPCALL\_MANUAL: Only set up the appcall, do not run it (you should call cleanup\_appcall() when finished). Please Refer to the "Manual Appcall" section for more information.
* APPCALL\_DEBEV: If this bit is set, exceptions during appcall will generate IDC exceptions with full information about the exception. Please refer to the "Capturing exception debug events" section for more information.

It is possible to retrieve the Appcall options, change them and then restore them back. To retrieve the options use the get\_appcall\_options().

Please note that the Appcall options are saved in the database so if you set it once it will retain its value as you save and load the database.

## Manual Appcall

So far, we've seen how to issue an Appcall and capture the result from the script, but what if we only want to setup the environment and manually step through a function?

This can be achieved with manual Appcall. The manual Appcall mechanism can be used to save the current execution context, execute another function in another context and then pop back the previous context and continue debugging from that point.

Let us directly illustrate manual Appcall with a real life scenario:

1. You are debugging your application
2. You discover a buggy function (foo()) that misbehaves when called with certain arguments: foo(0xdeadbeef)
3. Instead of waiting until the application calls foo() with the desired arguments that can cause foo() to misbehave, you can manually call foo() with the desired arguments and then trace the function from its beginning.
4. Finally, one calls cleanup\_appcall() to restore the execution context

To illustrate, let us take the ref1 function (from the previous example above) and call it with an invalid pointer:

1. Set manual Appcall mode:

set\_appcall\_options(APPCALL\_MANUAL);

1. Call the function with an invalid pointer:

ref1(6);

Directly after doing that, IDA will switch to the function and from that point on we can debug:

.text:0000000140001050 ; void \_\_stdcall ref1(int \*a)

.text:0000000140001050 ref1 proc near

.text:0000000140001050 test rcx, rcx ; **<< RIP starts here**

.text:0000000140001053 jz short locret\_14000106A

.text:0000000140001055 mov edx, [rcx]

.text:0000000140001057 lea r8d, [rdx+1]

.text:000000014000105B mov [rcx], r8d

.text:000000014000105E lea rcx, aCalledWithDAnd ; "called with %d and returning %d\n"

.text:0000000140001065 jmp \_printf

.text:000000014000106A locret\_14000106A:

.text:000000014000106A retn

.text:000000014000106A ref1 endp

Now you are ready to single step that function with all its arguments properly set up for you. When you are done, you can return to the previous context by calling cleanup\_appcall().

### Initiating multiple manual Appcalls

It is possible to initiate multiple manual Appcalls. If manual Appcall is enabled, then issuing an Appcall from an another Appcall will push the current context and switch to the new Appcall context. cleanup\_appcall() will pop the contexts one by one (LIFO style).

Such technique is useful if you happen to be tracing a function then you want to debug another function and then resume back from where you were!

Manual Appcalls are not designed to be called from a script (because they don't finish), nonetheless if you use them from a script:

auto i;

printf("Loop started\n"); // appcall 1

for (i=0;i<10;i++)

{

msg("i=%d\n", i);

}

printf("Loop finished\n"); // appcall 2

We observe the following:

1. First Appcall will be initiated
2. The script will loop and display the values of i in IDA's output window
3. Another Appcall will be initiated
4. The script finishes. None of the two Appcalls actually took place
5. The execution context will be setup for tracing the last issued Appcall
6. After this Appcall is finished, we observe "Loop finished"
7. We issue cleanup\_appcall and notice that the execution context is back to printf but this time it will print "Loop started"
8. Finally when we call again cleanup\_appcall we resume our initial execution context

## Capturing exception debug events

We previously illustrated that we can capture exceptions that occur during an Appcall, but that is not enough if we want to learn more about the nature of the exception from the operating system point of view.

It would be better if we could somehow get the last **debug\_event\_t** that occured inside the debugger module. This is possible if we use the **APPCALL\_DEBEV** option. Let us repeat the previous example but with the **APPCALL\_DEBEV** option enabled:

auto e;

try

{

set\_appcall\_options(APPCALL\_DEBEV); // Enable debug event capturing

ref1(6);

}

catch (e)

{

// Exception occured ..... this time "e" is populated with debug\_event\_t fields (check idd.hpp)

}

And in this case, if we dump the exception object's contents, we get these attributes:

Unhandled exception: object

can\_cont: 1. 1h 1o

code: 3221225477. C0000005h 30000000005o

ea: 4198442. 40102Ah 20010052o

eid: 64. 40h 100o

file: ""

func: "\_\_\_idcexec0"

handled: 1. 1h 1o

info: "The instruction at 0x40102A referenced memory at 0x6. The memory could not be read"

line: 2. 2h 2o

pc: 11. Bh 13o

pid: 40128. 9CC0h 116300o

ref: 6. 6h 6o

tid: 36044. 8CCCh 106314o

## Appcall related functions

There are some functions that can be used while working with Appcalls.

### parse\_decl/get\_tinfo/sizeof

The get\_tinfo() function is used to retrieve the typeinfo string associated with a given address.

/// Get type information of function/variable as 'typeinfo' object

/// ea - the address of the object

/// type\_name - name of a named type

/// returns: typeinfo object, 0 - failed

/// The typeinfo object has one mandatory attribute: typid

typeinfo get\_tinfo(long ea);

typeinfo get\_tinfo(string type\_name);

The parse\_decl() function is used to construct a typeinfo string from a type string. We already used it to construct a typeinfo string and passed it to dbg\_appcall().

/// Parse one type declaration

/// input - a C declaration

/// flags - combination of PT\_... constants or 0

/// PT\_FILE should not be specified in flags (it is ignored)

/// returns: typeinfo object or num 0

typeinfo parse\_decl(string input, long flags);

And finally, given a typeinfo string, one can use the sizeof() function to calculate the size of a type:

/// Calculate the size of a type

/// type - type to calculate the size of

/// can be specified as a typeinfo object (e.g. the result of get\_tinfo())

/// or a string with C declaration (e.g. "int")

/// returns: size of the type or -1 if error

long sizeof(typeinfo type);

## Accessing enum members as constants

In IDC, it is possible to access all the defined enumerations as if they were IDC constants:

00000001 ; enum MACRO\_PAGE (standard) (bitfield)

00000001 PAGE\_NOACCESS = 1

00000002 PAGE\_READONLY = 2

00000004 PAGE\_READWRITE = 4

00000008 PAGE\_WRITECOPY = 8

00000010 PAGE\_EXECUTE = 10h

00000020 PAGE\_EXECUTE\_READ = 20h

00000040 PAGE\_EXECUTE\_READWRITE = 40h

Then one can type:

msg("PAGE\_EXECUTE\_READWRITE=%x\n", PAGE\_EXECUTE\_READWRITE);

This syntax makes it even more convenient to use enumerations when calling APIs via Appcall.

## Storing/Retrieving typed elements

It is possible to store/retrieve (aka serialize/deserialize) objects to/from the database (or the debugee's memory). To illustrate, let us consider the following memory contents:

0001000C dd 1003219h

00010010 dw 0FFEEh

00010012 dw 0FFEEh

00010014 dd 1

And we know that this maps to a given type:

struct X

{

unsigned long a;

unsigned short b, c;

unsigned long d;

};

To retrieve (deserialize) the memory contents into a nice IDC object, we can use the object.retrieve() function:

/// Retrieve a C structure from the idb or a buffer and convert it into an object

/// typeinfo - description of the C structure. Can be specified

/// as a declaration string or result of \ref get\_tinfo() or

/// similar functions

/// src - address (ea) to retrieve the C structure from

/// OR a string buffer previously packed with the store method

/// flags - combination of \ref object\_store[PIO\_...] bits

void object.retrieve(typeinfo, src, flags);

Here is an example:

// Create the typeinfo string

auto t = parse\_decl("struct X { unsigned long a; unsigned short b, c; unsigned long d;};", 0);

// Create a dummy object

auto o = object();

// Retrieve the contents into the object:

o.retrieve(t, 0x1000C, 0);

And now if we dump the contents of **o**:

IDC>print(o);

object

**\_\_at\_\_:** 65548. 1000Ch 200014o 00000000000000010000000000001100b

a: 16790041. 1003219h 100031031o 00000001000000000011001000011001b

b: 65518. FFEEh 177756o 00000000000000001111111111101110b

c: 65518. FFEEh 177756o 00000000000000001111111111101110b

d: 1. 1h 1o 00000000000000000000000000000001b

and again we notice the **\_\_at\_\_** which holds the address of the retrieved object.

To store (serialize) the object back into memory, we can use the object.store() function:

/// Convert the object into a C structure and store it into the idb or a buffer

/// typeinfo - description of the C structure. Can be specified

/// as a declaration string or result of \ref get\_tinfo() or

/// similar functions

/// dest - address (ea) to store the C structure

/// OR a reference to a destination string

/// flags - combination of PIO\_.. bits

void object.store(typeinfo, dest, flags);

Here's an example continuing from the previous one:

o.a++; // modify the field

o.d = 6; // modify another field

o.store(t, o.\_\_at\_\_, 0);

And finally to verify, we go to the memory address:

0001000C dd 100321Ah

00010010 dw 0FFEEh

00010012 dw 0FFEEh

00010014 dd 6

# Using Appcall with IDAPython

The Appcall concept remains the same between IDC and Python, nonetheless Appcall/Python has a different syntax (using references, unicode strings, etc.)

The Appcall mechanism is provided by ida\_idd module (also via idaapi) through the Appcall variable. To issue an Appcall using Python:

from idaapi import Appcall

Appcall.printf("Hello world!\n");

One can take a reference to an Appcall:

printf = Appcall.printf

# ...later...

printf("Hello world!\n");

* In case you have a function with a mangled name or with characters that cannot be used as an identifier name in the Python language, then use the following syntax:

findclose = Appcall["\_\_imp\_\_FindClose@4"]

getlasterror = Appcall["\_\_imp\_\_GetLastError@0"]

setcurdir = Appcall["\_\_imp\_\_SetCurrentDirectoryA@4"]

* In case you want to redefine the prototype of a given function, then use the Appcall.proto(func\_name or func\_ea, prototype\_string) syntax as such:

# pass an address or name and Appcall.proto() will resolve it

loadlib = Appcall.proto("\_\_imp\_\_LoadLibraryA@4", "int (\_\_stdcall \*LoadLibraryA)(const char \*lpLibFileName);")

# Pass an EA instead of a name

freelib = Appcall.proto(LocByName("\_\_imp\_\_FreeLibrary@4"), "int (\_\_stdcall \*FreeLibrary)(int hLibModule);")

* To pass unicode strings you need to use the Appcall.unicode() function:

getmodulehandlew = Appcall.proto("\_\_imp\_\_GetModuleHandleW@4", "int (\_\_stdcall \*GetModuleHandleW)(LPCWSTR lpModuleName);")

hmod = getmodulehandlew(Appcall.**unicode**("kernel32.dll"))

* To pass int64 values to a function you need to use the Appcall.int64() function:

/\* C code \*/

int64 op\_two64(int64 a, int64 b, int op)

{

if (op == 1)

return a + b;

else if (op == 2)

return a - b;

else if (op == 3)

return a \* b;

else if (op == 4)

return a / b;

else

return -1;

}

Python Appcall code:

r = Appcall.op\_two64(Appcall.**int64**(1), Appcall.**int64**(2), 1)

print("result=", r.value)

If the returned value is also an int64, then you can use the int64.value to unwrap and retrieve the value.

* To define a prototype and then later assign an address so you can issue an Appcall:

# Create a typed object (no address is associated yet)

virtualalloc = Appcall.typedobj("int \_\_stdcall VirtualAlloc(int lpAddress, SIZE\_T dwSize, DWORD flAllocationType, DWORD flProtect);")

# Later we have an address, so we pass it:

virtualalloc.ea = idc.get\_name\_ea(0, "kernel32\_VirtualAlloc")

# Now we can Appcall:

ptr = virtualalloc(0, Appcall.Consts.MEM\_COMMIT, 0x1000, Appcall.Consts.PAGE\_EXECUTE\_READWRITE)

print("ptr=%x" % ptr)

Things to note:

* We used the Appcall.Consts syntax to access enumerations (similar to what we did in IDC)
* If you replicate this specific example, a new memory page will be allocated. You need to refresh the debugger memory layout (with idaapi.refresh\_debugger\_memory()) to access it

## Passing arguments by reference

* To pass function arguments by reference, one has to use the Appcall.byref():

# Create a byref object holding the number 5

i = Appcall.byref(5)

# Call the function

Appcall.ref1(i)

# Retrieve the value

print("Called the function:", i.value)

* To call a C function that takes a string buffer and modifies it, we need to use the Appcall.buffer(initial\_value, [size]) function to create a buffer:

buf = Appcall.buffer("test", 100)

Appcall.ref2(buf)

print(buf.cstr())

* Another real life example is when we want to call the GetCurrentDirectory() API:

# Take a reference

getcurdir = Appcall.proto("kernel32\_GetCurrentDirectoryA", "DWORD \_\_stdcall GetCurrentDirectoryA(DWORD nBufferLength, LPSTR lpBuffer);")

# make a buffer

buf = Appcall.byref("\x00" \* 260)

# get current directory

n = getcurdir(260, buf)

print("curdir=%s" % buf.cstr())

* To pass int64 values by reference:

int64\_t ref4(int64\_t \*a)

{

if (a == NULL)

{

printf("No number passed!");

return -1;

}

int64\_t old = \*a;

printf("Entered with %" PRId64 "\n", \*a);

(\*a)++;

return old;

}

We use the following Python code:

# Create an int64 value

i = Appcall.int64(5)

# create a reference to it

v = Appcall.byref(i)

# appcall

old\_val = Appcall.ref4(v)

print(f"Called with {old\_val.value}, computed {i.value}")

* To call a C function that takes an array of integers or an array of a given type:

/\* C code \*/

int ref3(int \*arr, int sz)

{

if (arr == NULL)

return 0;

int sum = 0;

for (int i=0;i<sz;i++)

sum += arr[i];

return sum;

}

First we need to use the Appcall.array() function to create an array type, then we use the array\_object.pack() function to encode the Python values into a buffer:

# create an array type

arr = Appcall.array("int")

# Create a test list

L = [x for x in range(1, 10)]

# Pack the list

p\_list = arr.pack(L)

# appcall to compute the total

c\_total = Appcall.ref3(p\_list, len(L))

# internally compute the total

total = sum(L)

if total != c\_total:

print("Appcall failed!")

else:

print(f"Total computed using Appcall is {total}")

## Functions that accept or return structures

Like in IDC, we can create objects and pass them with at least two methods.

The first method involves using the Appcall.obj() function that takes an arbitrary number of keyword args that will be used to create an object with the arguments as attributes. The second method is by using a dictionary.

# Via dictionary

rec1 = {"id": 1, "name": "user1"}

# Via Appcall.obj

rec2 = Appcall.obj(id=2, name="user2")

Appcall.printRecord(rec1)

Appcall.printRecord(rec2)

And finally, if you happen to have your own object instance then just pass your object. The IDAPython object to IDC object conversion routine will skip attributes starting and ending with "\_\_".

## FindFirst/FindNext example

# For simplicity, let's alias the Appcall

a = idaapi.Appcall

getcurdir = a.proto(

"kernel32\_GetCurrentDirectoryA",

"DWORD \_\_stdcall GetCurrentDirectoryA(DWORD nBufferLength, LPSTR lpBuffer);")

getwindir = a.proto(

"kernel32\_GetWindowsDirectoryA",

"UINT \_\_stdcall GetWindowsDirectoryA(LPSTR lpBuffer, UINT uSize);")

setcurdir = a.proto(

"kernel32\_SetCurrentDirectoryA",

"BOOL \_\_stdcall SetCurrentDirectoryA(LPCSTR lpPathName);")

findfirst = a.proto(

"kernel32\_FindFirstFileA",

"HANDLE \_\_stdcall FindFirstFileA(LPCSTR lpFileName, LPWIN32\_FIND\_DATAA lpFindFileData);")

findnext = a.proto(

"kernel32\_FindNextFileA",

"BOOL \_\_stdcall FindNextFileA(HANDLE hFindFile, LPWIN32\_FIND\_DATAA lpFindFileData);")

findclose = a.proto(

"kernel32\_FindClose",

"BOOL \_\_stdcall FindClose(HANDLE hFindFile);")

def test():

# create a buffer

savedpath = a.byref("\x00" \* 260)

# get current directory

n = getcurdir(250, savedpath)

out = []

out.append("curdir=%s" % savedpath.value[0:n])

# get windir

windir = a.buffer(size=260) # create a buffer using helper function

n = getwindir(windir, windir.size)

if n == 0:

print("could not get current directory")

return False

windir = windir.value[:n]

out.append("windir=%s" % windir)

# change to windows folder

setcurdir(windir)

# initiate find

fd = a.obj()

h = findfirst("\*.exe", fd)

if h == -1:

print("no \*.exe files found!")

return False

found = False

while True:

fn = a.cstr(fd.cFileName)

if "regedit" in fn:

found = True

out.append("fn=%s<" % fn)

fd = a.obj() # reset the FD object

ok = findnext(h, fd)

if not ok:

break

#

findclose(h)

# restore cur dir

setcurdir(savedpath.value)

# verify

t = a.buffer(size=260)

n = getcurdir(t.size, t)

if t.cstr() != savedpath.cstr():

print("could not restore cur dir")

return False

out.append("curdir=%s<" % t.cstr())

print("all done!")

for l in out:

print(l)

if found:

print("regedit was found!")

else:

print("regedit was not found!")

return found

test()

### Using GetProcAddress

a = idaapi.Appcall

loadlib = a.proto("kernel32\_LoadLibraryA", "HMODULE \_\_stdcall LoadLibraryA(const char \*lpLibFileName);")

getprocaddr = a.proto("kernel32\_GetProcAddress", "FARPROC \_\_stdcall GetProcAddress(HMODULE hModule, LPCSTR lpProcName);")

freelib = a.proto("kernel32\_FreeLibrary", "BOOL \_\_stdcall FreeLibrary(HMODULE hLibModule);")

def test\_gpa():

h = loadlib("user32.dll")

if idaapi.inf\_is\_64bit():

h = h.value

if h == 0:

print("failed to load library!")

return False

p = getprocaddr(h, "FindWindowA")

if idaapi.inf\_is\_64bit():

p = p.value

if p == 0:

print("failed to gpa!")

return -2

findwin = a.proto(p, "HWND FindWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName);")

hwnd = findwin(0, "Calculator")

freelib(h)

if idaapi.inf\_is\_64bit():

hwnd = hwnd.value

print("%x: ok!->hwnd=%x" % (p, hwnd))

return 1

test\_gpa()

Please note that we used the idaapi.inf\_is\_64bit() method to properly unwrap integer values that depends on the bitness of the binary.

## Setting the Appcall options

In Python, the Appcall options can be set global or locally per Appcall.

* To set the global Appcall setting:

old\_options = Appcall.set\_appcall\_options(Appcall.APPCALL\_MANUAL)

* To set the Appcall setting per Appcall:

# take a reference to printf

printf = Appcall.\_printf

# change the setting for this Appcall

printf.options = Appcall.APPCALL\_DEBEV

printf("Hello world!\n")

Similarly, retrieving the Appcall options is done by either calling Appcall.get\_appcall\_options() or by reading the options attribute (for example: printf.options)

To cleanup after a manual Appcall use Appcall.cleanup\_appcall().

## Calling functions that can cause exceptions

An Appcall that generates an exception while executing in the current thread will throw a Python **Exception** object. This is inline with the IDC behavior we described above.

* Let us try when the Appcall options does not include the APPCALL\_DEBEV flag:

try:

idaapi.Appcall.cause\_crash()

except Exception as e:

print("Got an exception!")

This approach is useful if you want to know whether the Appcall passes or crashes.

Now if we want more details about the exception, then we use the APPCALL\_DEBEV flag, which will cause an **OSError** exception to be raised and have its **args[0]** populated with the last debug\_event\_t:

cause\_crash = idaapi.Appcall.cause\_crash

cause\_crash.options = idaapi.APPCALL\_DEBEV

try:

cause\_crash()

except OSError as e:

debug\_event = e.args[0]

print(f"Exception: tid={debug\_event.tid} ea={debug\_event.ea:x}")

except Exception as e:

print("Unknown exception!")

If the Appcall caused a crash, then the **debug\_event** variable will be populated with the last debug\_event\_t structure inside the OSError exception handler.

## Appcall related functions in Python

### Storing/Retrieving objects

Storing/Retrieving objects is also supported in Python:

1. Using the IDA SDK (through the idaapi Python module)
2. Using Appcall helper functions

In this example we show how to:

1. Unpack the DOS header at address 0x140000000 and verify the fields
2. Unpack a string and see if it is unpacked correctly

Let's start with the IDA SDK helper functions first:

# Struct unpacking

def test\_unpack\_struct():

name, tp, flds = idc.parse\_decl("IMAGE\_DOS\_HEADER;", 0)

ok, obj = idaapi.unpack\_object\_from\_idb(idaapi.get\_idati(), tp, flds, 0x140000000, 0)

return obj.e\_magic == 23117 and obj.e\_cblp == 144

# Raw unpacking

def test\_unpack\_raw():

# Parse the type into a type name, typestring and fields

name, tp, flds = idc.parse\_decl("struct abc\_t { int a, b;};", 0)

# Unpack from a byte vector (bv) (aka string)

ok, obj = idaapi.unpack\_object\_from\_bv(

idaapi.get\_idati(),

tp,

flds,

b"\x01\x00\x00\x00\x02\x00\x00\x00",

0)

return obj.a == 1 and obj.b == 2

print("test\_unpack\_struct() passed:", test\_unpack\_struct())

print("test\_unpack\_raw() passed:", test\_unpack\_raw())

Now to accomplish similar result using Appcall helper functions:

# Struct unpacking with Appcall

def test\_unpack\_struct():

tp = idaapi.Appcall.typedobj("IMAGE\_DOS\_HEADER;")

ok, obj = tp.retrieve(0x140000000)

return ok and obj.e\_magic == 23117 and obj.e\_cblp == 144

# Raw unpacking with Appcall

def test\_unpack\_raw():

global tp

# Parse the type into a type name, typestring and fields

tp = idaapi.Appcall.typedobj("struct abc\_t { int a, b;};")

ok, obj = tp.retrieve(b"\x01\x00\x00\x00\x02\x00\x00\x00")

return obj.a == 1 and obj.b == 2

print("test\_unpack\_struct() passed:", test\_unpack\_struct())

print("test\_unpack\_raw() passed:", test\_unpack\_raw())

When it comes to storing, instead of using the Appcall's typedobj.retrieve(), we can use the typedobj.store() function:

# Packs/Unpacks a structure to the database using appcall facilities

def test\_pack\_idb(ea):

print("%x: ..." % ea)

tp = a.typedobj("struct { int a, b; char x[4];};")

o = a.obj(a=16, b=17,x="abcd")

return tp.store(o, ea) == 0

ea = idc.here() # some writable area

if test\_pack\_idb(ea):

print("cool!")

idaapi.refresh\_debugger\_memory()

### Accessing enum members as constants

Like in IDC, to access the enums, one can use the Appcall.Consts object:

print("PAGE\_EXECUTE\_READWRITE=%x" % Appcall.Consts.PAGE\_EXECUTE\_READWRITE)

If the constant was not defined then an attribute error exception will be thrown. To prevent that, use the Appcall.valueof() method instead, which lets you provide a default value in case a constant was absent:

print("PAGE\_EXECUTE\_READWRITE=%x" % Appcall.valueof("PAGE\_EXECUTE\_READWRITE", **0x40**))

Please send your comments or questions to [support@hex-rays.com](mailto:support@hex-rays.com)