## Usage examples

To work with Stadeo, we first need to set up a RPyC (Remote Python Call) server inside IDA, which allows us to access the IDA API from an arbitrary Python interpreter. You can use this script to open an RPyC server in IDA.

In all the examples below, we set up an RPyC server listening on 10.1.40.164:4455 (4455 being the default port), and then communicate with the server from a Python console.

We will use two Stadeo classes:

- **CFFStrategies** for CFF deobfuscation
- **StringRevealer** for string deobfuscation

Both classes can be initialized for both 32- and 64-bit architecture.

### Deobfuscating a single function

SHA-1 of the sample: `791ad58d9bb66ea08465aad4ea968656c81d0b8e`

The code below deobfuscates the function at **0x1800158B0** with the parameter in the **R9** register set to **0x567C** and writes its deobfuscated version to the **0x18008D000** address.

The **R9** parameter is a control variable for function-merging the CFF loop (merging variable) and it has to be specified using Miasm expressions, which are documented here; note that one has to use **RSP_init/ESP_init** instead of **RSP/ESP** to refer to the stack parameters. For example, we would use `ExprMem(ExprId("ESP_init", 32) + ExprInt(4, 32), 32)` to target the first stack parameter on 32-bit architecture.

```
from stadeo.cff.cff_strategies import CFFStrategies
from miasm.expression.expression import ExprId, ExprInt

cs = CFFStrategies(64)
cs.solve_loop(0x1800158B0,
              0x18008D000,
              context={ExprId("R9", 64): ExprInt(0x567c, 64)},
              ip='10.1.40.164')
```

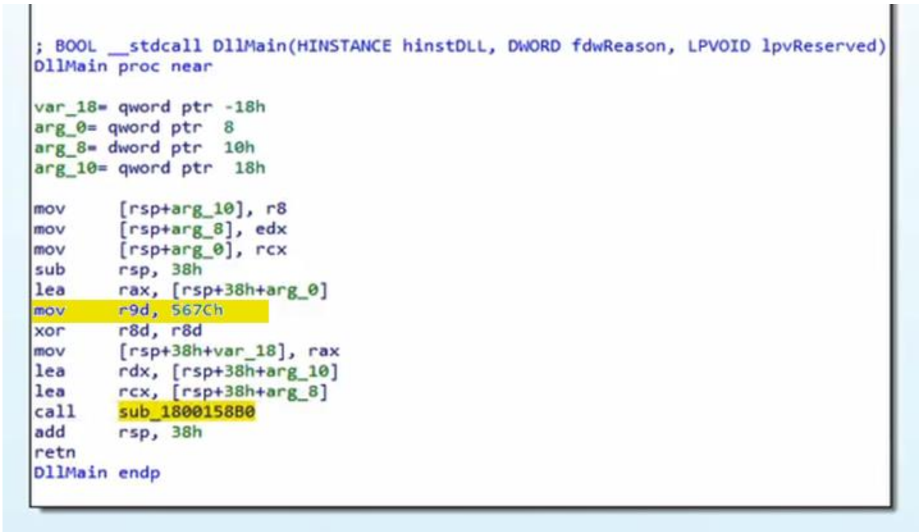



*Figure 1. Call to the obfuscated function*

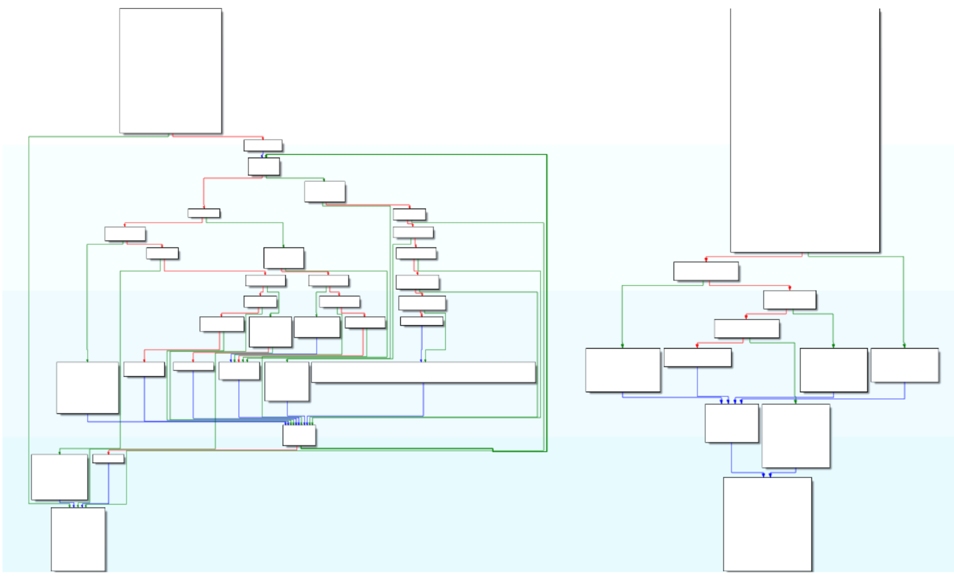


*Figure 2. Obfuscated (left) and deobfuscated (right) CFG*

## Processing reachable functions

SHA-1 of the sample: `e0087a763929dee998deebbcfa707273380f05ca`

The following code recognizes only obfuscated functions reachable from **0x1002DC50** and searches for candidates for merging variables. Successfully recognized and deobfuscated functions are starting at **0x10098000**.

```
from stadeo.cff.cff_strategies import CFFStrategies
strat = CFFStrategies(32)
res = strat.process_merging(0x1002DC50,
                            0x10098000,
                            ip="10.1.40.164")
```

Partial output:

```
skipping 0x1002dc50 with val None: 0xbadf00d

mapping: 0x1001ffc0 -> 0x10098000 with val @32[ESP_init + 0x8]: 0x6cef

skipping 0x100010f0 with val None: 0xbadf00d

mapping: 0x1000f0c0 -> 0x100982b7 with val @32[ESP_init + 0x8]: 0x2012

mapping: 0x1003f410 -> 0x10098c8a with val @32[ESP_init + 0x4]: 0x21a4

mapping: 0x1003f410 -> 0x10098f3d with val @32[ESP_init + 0x4]: 0x772a

skipping 0x1004ee79 (library func)

...
```

Mapped functions were deobfuscated properly and skipped ones weren't considered to be obfuscated. The format of the **mapping** lines in the output is:

```
mapping: %obfuscated_function_address% ->
%deobfuscated_function_address% with val %merging_variable%: %
merging_variable_value%
```

The default value for `merging_variable_value` is `0x0BADF00D`. The **skipping** lines follow the same pattern, but there's no `deobfuscated_function_address`. Library functions recognized by IDA are just skipped without further processing.

## Processing all functions

SHA-1 of the sample: `e575f01d3df0b38fc9dc7549e6e762936b9cc3c3`

We use the following code only to deal with CFF featured in Emotet, whose CFF implementation fits into the description of common control-flow flattening here.

We prefer this approach because the method `CFFStrategies.process_all()` does not attempt to recognize merging variables that are not present in Emotet and searches only for one CFF loop per function; hence it is more efficient.

Successfully recognized and deobfuscated functions are sequentially written from **0x0040B000**. Format of the output is the same as in the **`process_merging`** method used in the *Processing reachable functions* example, but naturally there won't be any merging variables.

```
from stadeo.cff.cff_strategies import CFFStrategies
strat = CFFStrategies(32)
res = strat.process_all(0x0040b000,
                        ip="10.1.40.164")
```

Partial output:

```
mapping: 0x00401020 -> 0x0040b000 with val None: 0xbadf00d

skipping 0x00401670 with val None: 0xbadf00d

mapping: 0x00401730 -> 0x0040b656 with val None: 0xbadf00d

...
```

```
v2 = 804876744;
v3 = 0;
while ( 1 )
{
  while ( 1 )
  {
    while ( v2 > 590244004 )
    {
      if ( v2 == 594130554 )
      {
        sub_402EC0(dword_40ACB8 + 36);
        v2 = 472512193;
      }
      else
      {
        v2 = 590244004;
      }
    }
    if ( v2 != 590244004 )
      break;
    v4 = getLibrary(-999761153, a2);
    v5 = (void (__stdcall *)(_DWORD, char *,
    v5(0, v7, 260);
    v2 = 594130554;
  }
  if ( v2 != 472512193 )
    break;
  v3 = sub_402DB0(v7, dword_40ACB8 + 36);
  v2 = 548948320;
}
sub_402D10(dword_40ACB8 + 36);
return v3;
```

```
v2 = getLibrary(-999761153, a2);
v3 = (void (__stdcall *)(_DWORD, char *,
v3(0, v6, 260);
sub_402EC0(dword_40ACB8 + 36);
v4 = sub_402DB0(v6, dword_40ACB8 + 36);
sub_402D10(dword_40ACB8 + 36);
return v4;
```

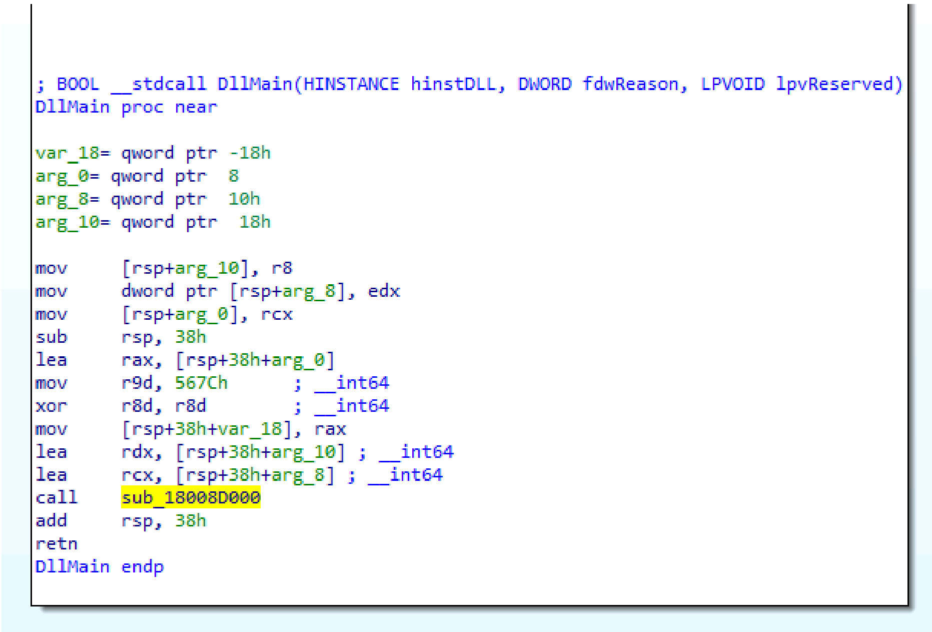
*Figure 3. Example of obfuscated (left) and deobfuscated (right) function in Emotet*

## Redirecting references to deobfuscated functions

Stadeo doesn't automatically update references to functions after their deobfuscation. In the example below, we demonstrate how to patch a function call in Figure 1 from the *Deobfuscating a single function* example. The patched reference is shown in Figure 4.

We use the following code to patch calls, whose fourth parameter is **0x567C**, to the obfuscated function at **0x1800158B0** with the deobfuscated one at **0x18008D000**. Note that one has to make sure that IDA has recognized parameters of the function correctly and possibly fix them.

```
from stadeo.utils.xref_patcher import patch_xrefs
patch_xrefs(0x1800158B0,
            0x18008D000,
            {3: 0x567c},
            ip='10.1.40.164')
```



```
; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
DllMain proc near

var_18= qword ptr -18h
arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h

mov     [rsp+arg_10], r8
mov     dword ptr [rsp+arg_8], edx
mov     [rsp+arg_0], rcx
sub     rsp, 38h
lea     rax, [rsp+38h+arg_0]
mov     r9d, 567Ch      ; __int64
xor     r8d, r8d        ; __int64
mov     [rsp+38h+var_18], rax
lea     rdx, [rsp+38h+arg_10] ; __int64
lea     rcx, [rsp+38h+arg_8] ; __int64
call    sub_18008D000
add     rsp, 38h
retn
DllMain endp
```

*Figure 4. Patched reference*

## Revealing obfuscated strings in a function

The function `StringRevealer.process_funcs()` reveals obfuscated strings in the specified function and returns a map of deobfuscated strings and their addresses.

Note that control flow of the target function has to have been deobfuscated already.

In the example below, we deobfuscate the strings of the function at **0x100982B7**, shown in Figure 5. The function was deobfuscated in the *Processing reachable functions* example.

```
from stadeo.string.string_revealer import StringRevealer
sr = StringRevealer(32)
strings = sr.process_funcs([0x100982B7],
                           ip="10.1.40.164")
```

```
...
v46 = 'k\0c';
*(_QWORD *)Dest = 'o\0L\0e\0S';
v47 = 0;
wcsncat(Dest, aM_1, 1u);
v48 = 'm\0e';
*(_DWORD *)String1 = 'o';
lstrcpyW(&String1[1], "r");
lstrcatW(Dest, L"yPr");
wcsncat(Dest, aI_0, 1u);
v51[0] = 'g';
v50 = 'e\0l\0i\0v';
*(_DWORD *)&v51[1] = 'e';
...
```

*Figure 5. Part of the function at 0x100982B7 which clearly assembles a string*

Content of the strings variable after the execution is:

```
{0x100982B7: {'SeLockMemoryPrivilege'}}
```