Analyzing an IDA Pro anti-decompilation code

antonioparata.blogspot.com/2022/01/analyzing-ida-pro-anti-decompilation.html

Twitter: @s4tan
GitHub: https://github.com/enkomio/

In this post I'll analyze a piece of code that induces IDA Pro to decompile the assembly in a wrong way. I'll propose a fix, but I'm open to more elegant solutions :)

The function that we want to decompile has the following assembly code (I'm using IDA Pro v7.6):

```
.text:1001BC95 56                   push    esi
.text:1001BC96 FF 74 24 10          push    [esp+4+arg_8]
.text:1001BC9A 8B 74 24 10          mov     esi, [esp+8+arg_4]
.text:1001BC9E 56                   push    esi
.text:1001BC9F FF 74 24 10          push    [esp+0Ch+arg_0]
.text:1001BCA3 52                   push    edx
.text:1001BCA4 51                   push    ecx
.text:1001BCA5 E8 57 20 FF FF       call    nullsub_1
.text:1001BCAA 8B 0A                mov     ecx, [edx]
.text:1001BCAC 83 C4 14             add     esp, 14h
.text:1001BCAF 89 4E 0C             mov     [esi+0Ch], ecx
.text:1001BCB2 8B 42 04             mov     eax, [edx+4]
.text:1001BCB5 03 C1                add     eax, ecx
.text:1001BCB7 89 46 04             mov     [esi+4], eax
.text:1001BCBA 5E                   pop     esi
.text:1001BCBB C3                   retn
```

The function uses two arguments with an unconventional calling convention. If we decompile the code, we obtain:

```
int __cdecl sub_1001BC95(int a1, int a2)
{
  int *v2; // edx
  int v3; // ecx
  int result; // eax

  nullsub_1();
  v3 = *v2;
  *(a2 + 12) = *v2;
  result = v3 + v2[1];
  *(a2 + 4) = result;
  return result;
}
```

In IDA Pro the **v2** variable (corrisponding to the line at address *0x1001BCAA*) is colored in red, since its value might be undefined.

Custom calling convention might cause some problems to the decompilation process (see this), but, in general, there exist an easy fix to it: it is enough to inform IDA Pro that the function uses a custom calling convention. By modifying the function, we can set the new type with the following definition:

```
int __usercall sub_1001BC95@<eax>(PUCHAR arg0@<edx>, int garbage, PUCHAR arg1)
```

with this new definition, the decompiled code now looks like the following:

```
int __usercall sub_1001BC95@<eax>(PUCHAR arg0@<edx>, int garbage, PUCHAR arg1)
{
  int *v1; // edx
  int v2; // ecx
  int result; // eax
  int v4; // [esp+Ch] [ebp+8h]

  nullsub_1();
  v2 = *v1;
  *(v4 + 12) = *v1;
  result = v2 + v1[1];
  *(v4 + 4) = result;
  return result;
}
```

We haven't done any progress at all. The only place where we haven't checked is the **nullsub_1** function, the problem must be in its call. If we analyze this function, we notice that it has an empty body, as shown below.

```
.text:1000DD01 C3                  retn
```

Why is this function causing problems? The answer is in the software convention used by the compiler. During the compilation, the compiler considers some registers as volatile. This means that the value of these registers, after a function call, should not be considered preserved ([1]). Among the volatile registers, there is **EDX**, which is exactly one of the registers used to pass a function parameter in the custom calling convention.

This code causes problem to the decompilation process that considers (correctly) the **EDX** register to have an undefined value after the function call.

I'm not aware of any particular IDA Pro command to inform the decompiler to not consider **EDX** as volatile, so the simpler solution that I found is to just remove the call instruction (I patched the bytes *E8 57 20 FF FF* with *90 90 90 90 90*). The result is a much cleaner decompiled code, as shown below.

```
int __usercall sub_1001BC95@<eax>(PUCHAR arg0@<edx>, int garbage, PUCHAR arg1)
{
  PUCHAR v3; // ecx
  int result; // eax

  v3 = *arg0;
  *(arg1 + 3) = *arg0;
  result = &arg0[1][v3];
  *(arg1 + 1) = result;
  return result;
}
```

Now we can proceed to further improve the decompilation code (we can clearly see the usage of a struct in the code) now that the decompiled code represents the real intent of the assembly code.

Update:

I received a message on twitter and reddit that suggests to have a look at the **__spoils** keyword mentioned in this *Igor's tip of the week* post [2] (shame on me for not having found it).

Its meaning is exactly what we need to solve the problem in a more elegant and generic way. It is enough to change the **nullsub_1** function definition by adding the **__spoils** keyword, as show below:

```
void __spoils<> nullsub_1(void)
```

The decompilation result of the function **sub_1001BC95** is the same as before with the exception that the call to the **nullsub_1** function is still there (it is not necessary to patch the bytes anymore).

Links:

[1] Register volatility and preservation
[2] Igor's tip of the week #51: Custom calling conventions