

Reverse Engineering with Ghidra

Dr. Alexandros Kapravelos
akaprav@ncsu.edu

John (Jack) Allison
President, HackPack
jeallis2@ncsu.edu

“What’s a Ghidra?”

“A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission”

*(It’s a tool for reverse engineering stuff)
(declassified after it’s existence was leaked by WikiLeaks ;))*

<https://ghidra-sre.org/CheatSheet.html>

What can it do?

- x86 **decompilation** (and ARM... and MIPS... and a lot more...)
 - Pretty much any architecture that's commonly used
- Debugging binaries under Windows and Linux (WinDbg and GDB)
- Scripting with Python (not covered here) to extend the feature set or automate tasks

Extending Ghidra

- <https://github.com/AllsafeCyberSecurity/awesome-ghidra>
- <https://github.com/topics/ghidra-scripts>
- <https://github.com/federicodotta/ghidra-scripts>

It's free and open source

- Possibly limited feature set compared to...
 - IDA Pro (multiple thousands), state of the art
 - Binary Ninja (\$300 personal license)
- But, it's **free**
- And now has a debugger, which was a missing feature till recently

What it can be used for

- Malware analysis
- CTF Challenges 🙄
- Learning how your favorite program works, under the hood

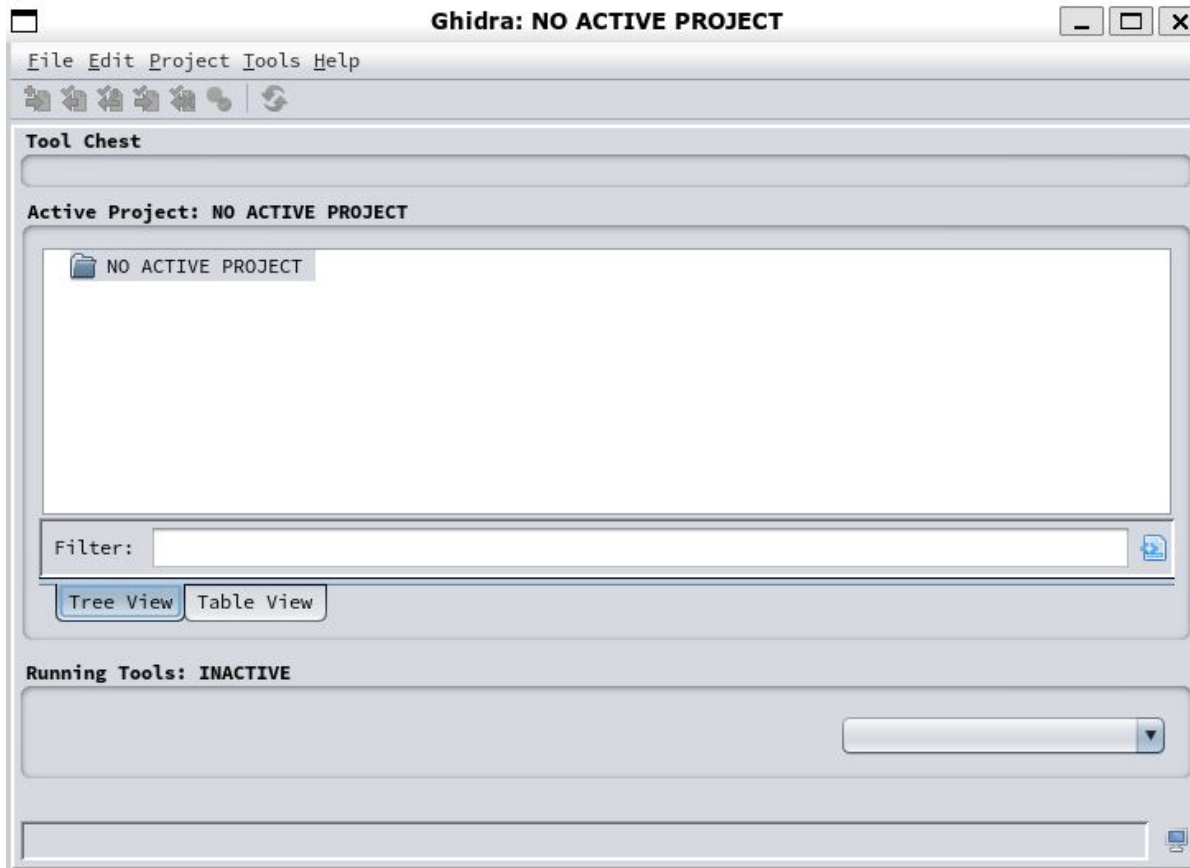
Let's do a quick demonstration.

Let's poke around a bit

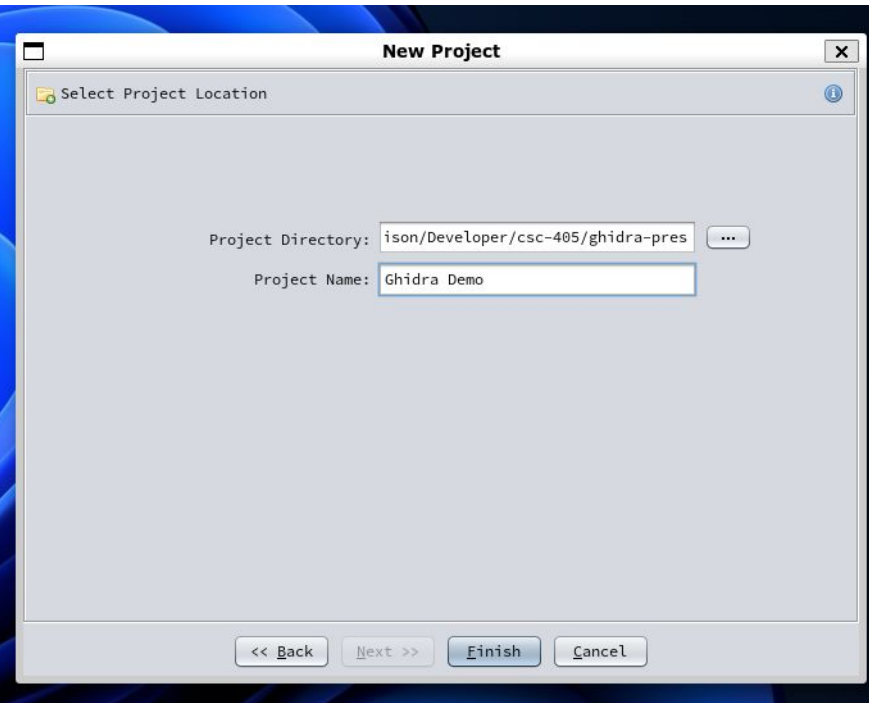
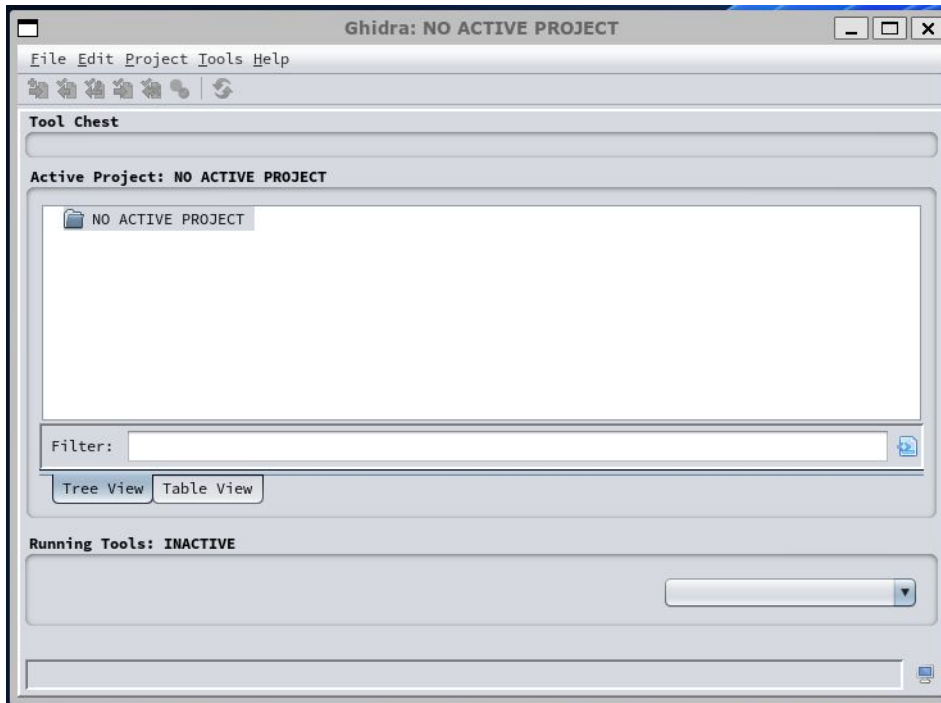
```
jallison at the-crag in ~/Developer/csc-405/ghidra-pres (master ..4)
λ ./hello_stripped
./hello_stripped (Executable, 15kB) ./hello_unstripped (Executable, 16kB)
```

*Note that the stripped binary (no debug symbols) is a tad bit smaller

Let's poke around a bit

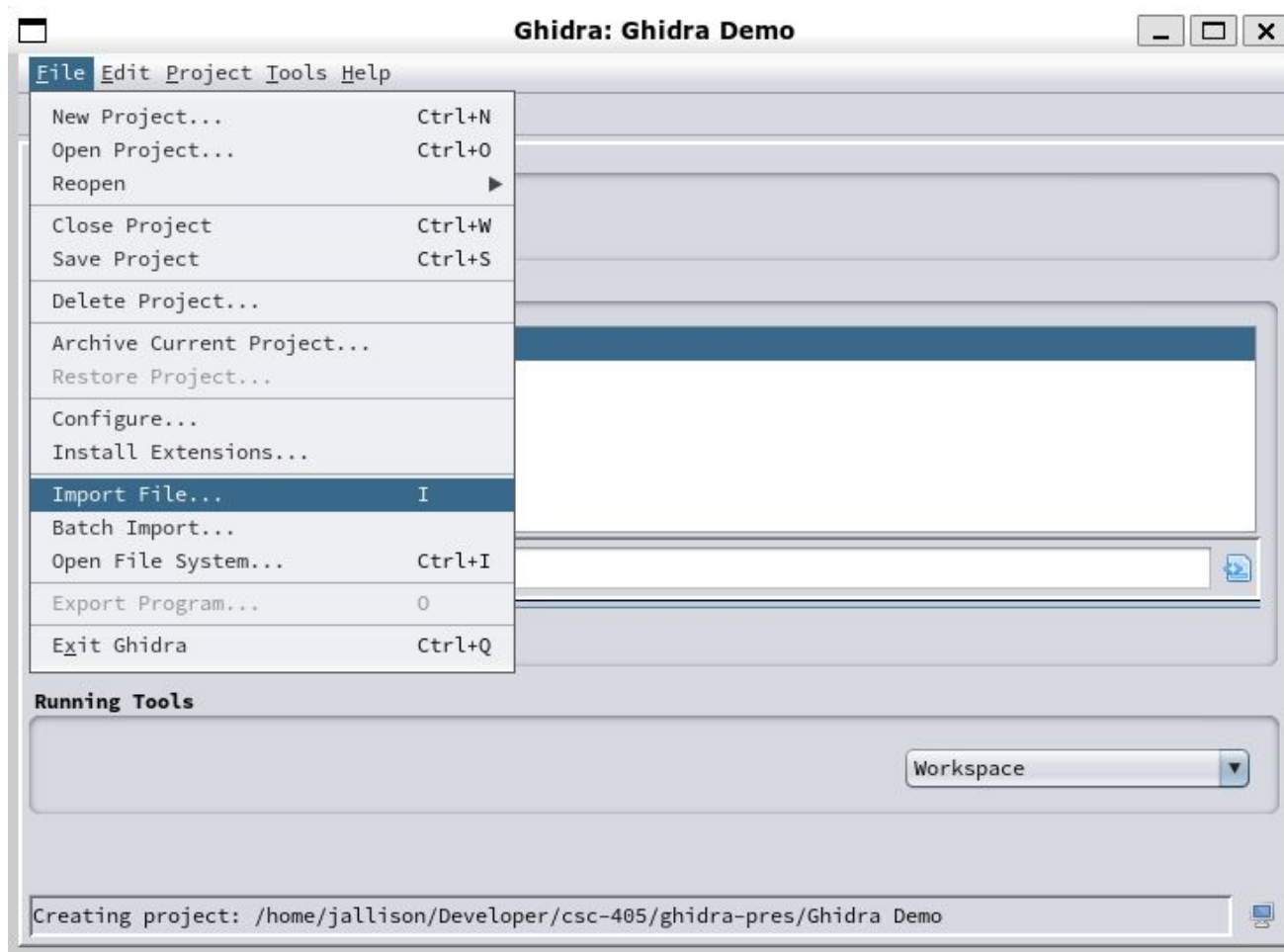


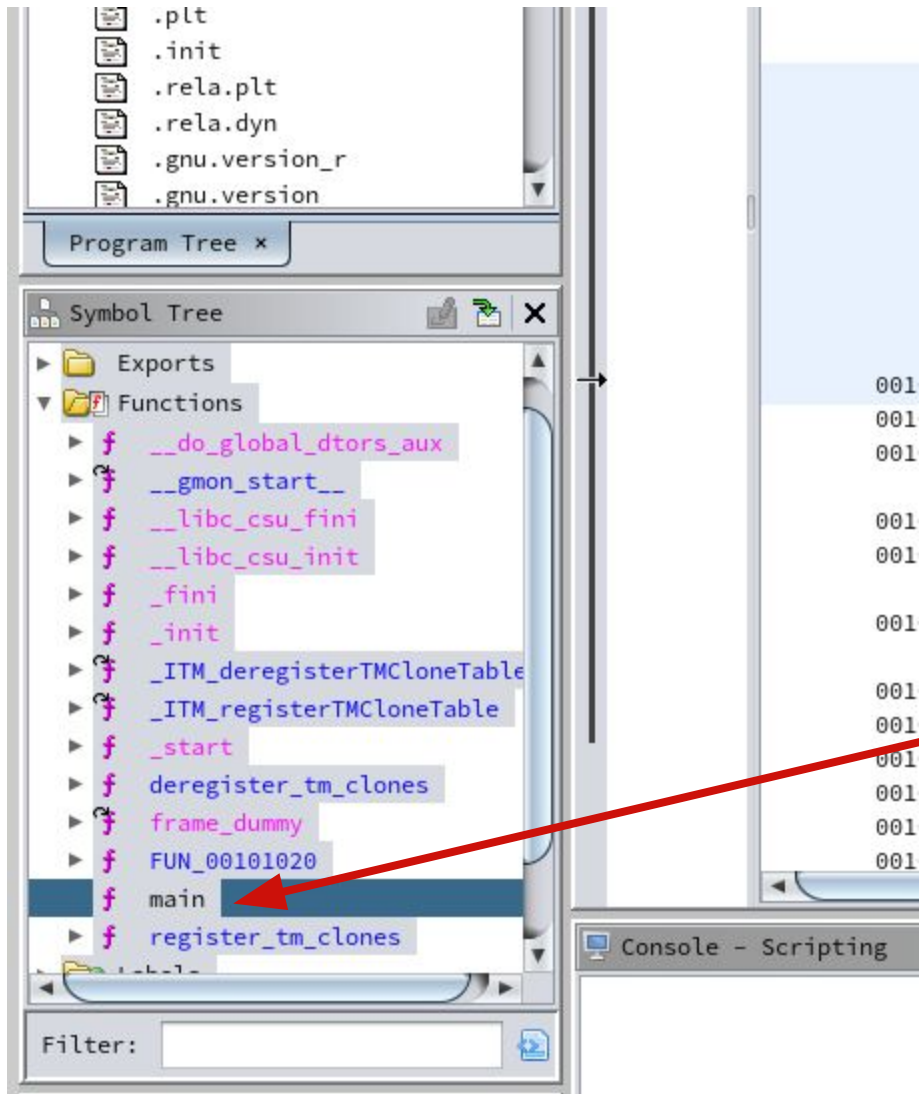
Create a project



Import your binary you want to analyze

(let's start with the unstripped binary)





On the left is a list of functions, including 'main'

Assembly and decompilation

```

trame_dummy
XREF[3]:  entry POINT(*),
          __libc_csu_init:001011a9(c),
          00103de8(*)

00101130 f3 0f 1e fa  ENDBR64
00101134 e9 67 ff    JMP      register_tm_clones
          ff ff

-- Flow Override: CALL_RETURN (CALL_TERMINATOR)
*****
*
* FUNCTION
*
*****
undefined main()
AL:1      <RETURN>
main
XREF[4]:  Entry Point(*),
          _start:00101061(*), 00102030,
          001020a8(*)

00101139 55          PUSH   RBP
0010113a 48 89 e5    MOV    RBP,RSP

```

```

1
2 undefined8 main(void)
3
4 {
5     puts("Hello, world!");
6     return 0;
7 }
8

```

**Note that this is PCode, Ghidra's IR specified by SLEIGH.
We'll call it that from now on.**

Decompiled code

```
8 #include <stdio.h>
7
6 int main(void)
5 {
4     printf("Hello, world!\n");
3     return 0;
2 }
1
```



```
Decompile: main - (hello_unstripped)
1
2 undefined8 main(void)
3
4 {
5     puts("Hello, world!");
6     return 0;
7 }
8
```

- Best effort attempt
- NOT 1:1
- Still, very helpful

Let's try something a bit more complex

```
jallison at the-crag in ~/Developer/csc-405/ghidra-pres (master ...9)
λ make
gcc more_complex.c -g -O0 -o more_complex_stripped
gcc more_complex.c -O0 -o more_complex_unstripped
```

What's the difference here?

- Compiling a stripped version
- Optimizations turned down with `-O0` to make it (ideally) more similar to the source once decompiled

Size differences, revisited

```
jallison at the-crag in ~/Developer/csc-405/ghidra-pres (master ...11)
└─λ ./more_complex_stripped
..._complex_stripped (Executable, 17kB) ..._complex_unstripped (Executable, 15kB)
```


What do we think this does?

```
jallison at the-crag in ~/Developer/csc-405/ghidra-pres (master ...11)
└─λ ./more_complex_stripped
Hm... I wonder what this thing is?
Oh, it's THIS THING...
```

It probably...

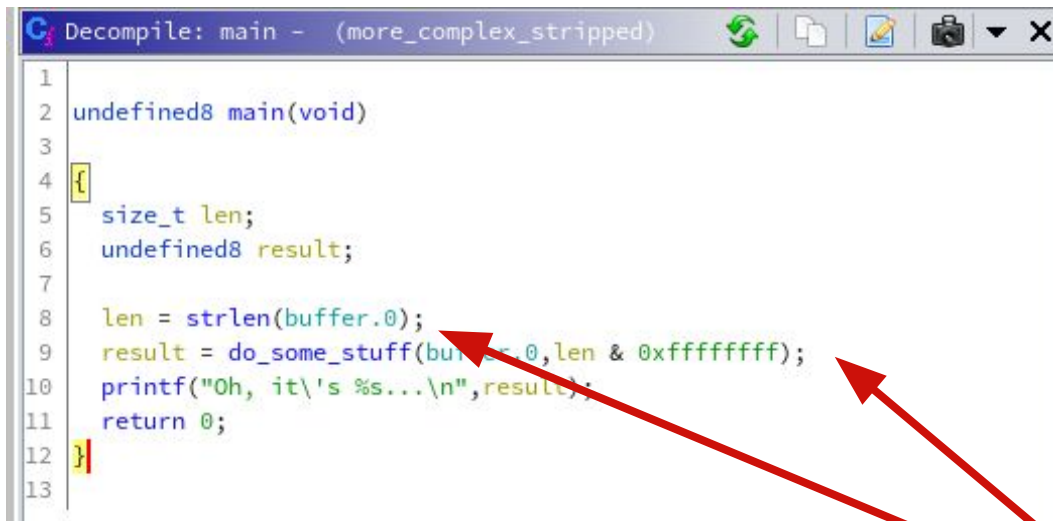
- Takes an input string
- Returns it, with modifications

Our main function

A screenshot of a decompiled C function named 'main'. The window title is 'Decompile: main - (more_complex_stripped)'. The code is as follows:

```
1
2 undefined8 main(void)
3
4 {
5     size_t sVar1;
6     undefined8 uVar2;
7
8     sVar1 = strlen(buffer.0);
9     uVar2 = do_some_stuff(buffer.0,sVar1 & 0xffffffff);
10    printf("Oh, it\'s %s...\n",uVar2);
11    return 0;
12 }
13
```

We can rename stuff!



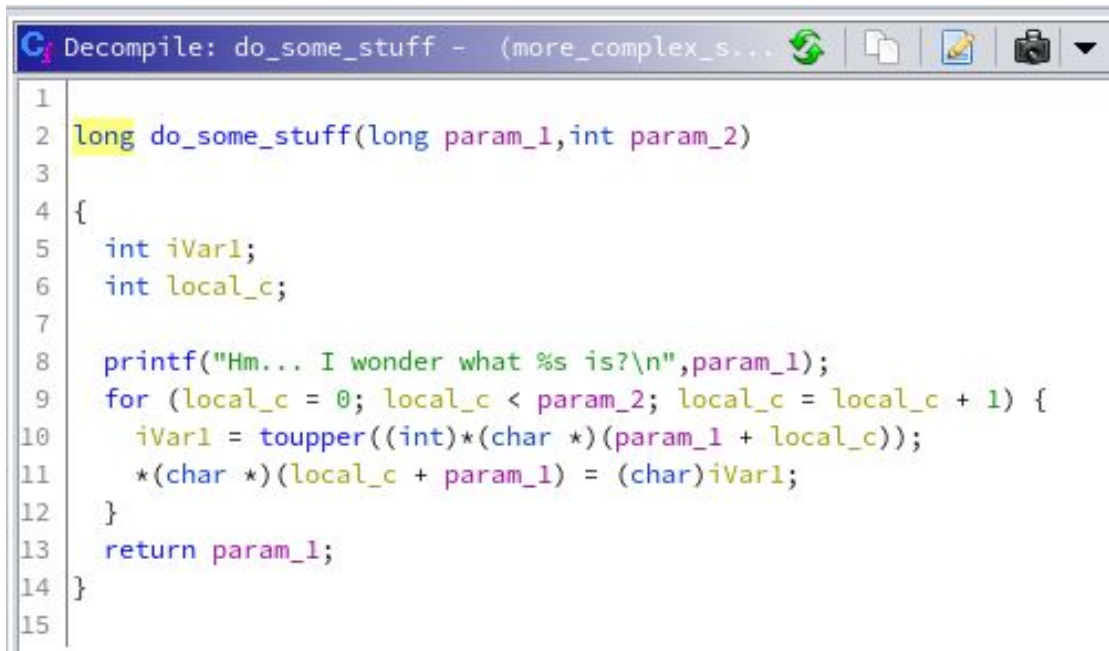
The screenshot shows a decompiler window titled "Decompile: main - (more_complex_stripped)". The code is as follows:

```
1
2 undefined8 main(void)
3
4 {
5     size_t len;
6     undefined8 result;
7
8     len = strlen(buffer.0);
9     result = do_some_stuff(buffer.0, len & 0xffffffff);
10    printf("Oh, it's %s...\n", result);
11    return 0;
12 }
13
```

Two red arrows originate from the bottom right of the slide. One arrow points to the variable `len` in line 9, and the other points to the function call `do_some_stuff` in line 9.

- Rename selected variables with 'l' (lowercase L)
- Can double-click that 'do_some_stuff' function to jump to it
- Lots of random compiler stuff going on, ignore it

Do Some Stuff?



```
Decompile: do_some_stuff - (more_complex_s...
1
2 long do_some_stuff(long param_1, int param_2)
3
4 {
5     int iVar1;
6     int local_c;
7
8     printf("Hm... I wonder what %s is?\n", param_1);
9     for (local_c = 0; local_c < param_2; local_c = local_c + 1) {
10        iVar1 = toupper((int)*(char *) (param_1 + local_c));
11        *(char *) (local_c + param_1) = (char) iVar1;
12    }
13    return param_1;
14 }
15
```

- Ghidra obviously got some stuff wrong
- We know the first parameter was a char *, but the function decompilation lists it as a long
- Let's clean this up.
*we can retype variables with CTRL-L

Do Some Stuff?

A screenshot of a decompiled C function named 'do_some_stuff'. The window title is 'Decompile: do_some_stuff - (more_complex_s...'. The code is as follows:

```
1  
2 char * do_some_stuff(char *buffer,int buflen)  
3  
4 {  
5     int upperChar;  
6     int i;  
7  
8     printf("Hm... I wonder what %s is?\n",buffer);  
9     for (i = 0; i < buflen; i = i + 1) {  
10        upperChar = toupper((int)buffer[i]);  
11        buffer[i] = (char)upperChar;  
12    }  
13    return buffer;  
14 }  
15
```

Now we have a significantly easier to decipher function

What does it do? Any guesses?

Here's some of the PCode of do_some_stuff

```

undefined4      Stack[-0x24]:4 local_24
do_some_stuff
XREF[2]:        001011c8(R)
                00101165(W),
                001011c3(R)
XREF[4]:        Entry Point(*), main:001011f1(c),
                0010204c, 001020d0(*)

00101159 55      PUSH    RBP
0010115a 48 89 e5   MOV     RBP,RSP
0010115d 48 83 ec 20 SUB     RSP,0x20
00101161 48 89 7d e8 MOV     qword ptr [RBP + local_20],buffer
00101165 89 75 e4   MOV     dword ptr [RBP + local_24],bufflen
00101168 48 8b 45 e8 MOV     RAX,qword ptr [RBP + local_20]
0010116c 48 89 c6   MOV     bufflen,RAX
0010116f 48 8d 05   LEA    RAX,[s_Hm..._I_wonder_what_%s_is?_00102004] = "Hm... I wonder what %s is?\n"
                8e 0e 00 00
00101176 48 89 c7   MOV     buffer=>s_Hm..._I_wonder_what_%s_is?_00102004,... = "Hm... I wonder what %s is?\n"
00101179 b8 00 00   MOV     EAX,0x0
                00 00
0010117e e8 cd fe   CALL   <EXTERNAL>::printf                          int printf(char * __format, ...)
                ff ff
00101183 c7 45 fc   MOV     dword ptr [RBP + i],0x0
                00 00 00 00
0010118a eb 34     JMP     LAB_001011c0

LAB_0010118c
XREF[1]:        001011c6(j)
0010118c 8b 45 fc   MOV     EAX,dword ptr [RBP + i]
0010118f 48 63 d0   MOVSXD RDX,EAX
00101192 48 8b 45 e8 MOV     RAX,qword ptr [RBP + local_20]
00101196 48 01 d0   ADD    RAX,RDX
00101199 0f b6 00   MOVZX  EAX,byte ptr [RAX]
0010119c 0f be c0   MOVSX  EAX,AL
0010119f 89 c7     MOV     buffer,EAX
001011a1 e8 8a fe   CALL   <EXTERNAL>::toupper                          int toupper(int __c)
                ff ff
001011a6 88 45 fb   MOV     byte ptr [RBP + local_d],upperChar
001011a9 8b 45 fc   MOV     upperChar,dword ptr [RBP + i]
001011ac 48 63 d0   MOVSXD RDX,upperChar
001011af 48 8b 45 e8 MOV     upperChar,qword ptr [RBP + local_20]
001011b3 48 01 c2   ADD    RDX,upperChar

```