Advanced Security - Code (de-)Obfuscation

Hatim Ridaoui

November 2025

1 Part 1

Both files were compiled without PIE so that the function addresses remain fixed, and the memory page containing the function was changed to rwx using `mprotect`.

For the first program, `foo_add.c`, the function `foo` increments a local variable and prints its value. After the first call, we located the immediate value of the `add` instruction in the machine code and replaced it with `0xFF`, which turns the `i++` operation into `i--`. As a result, the second call prints `i = -1` instead of `1`.

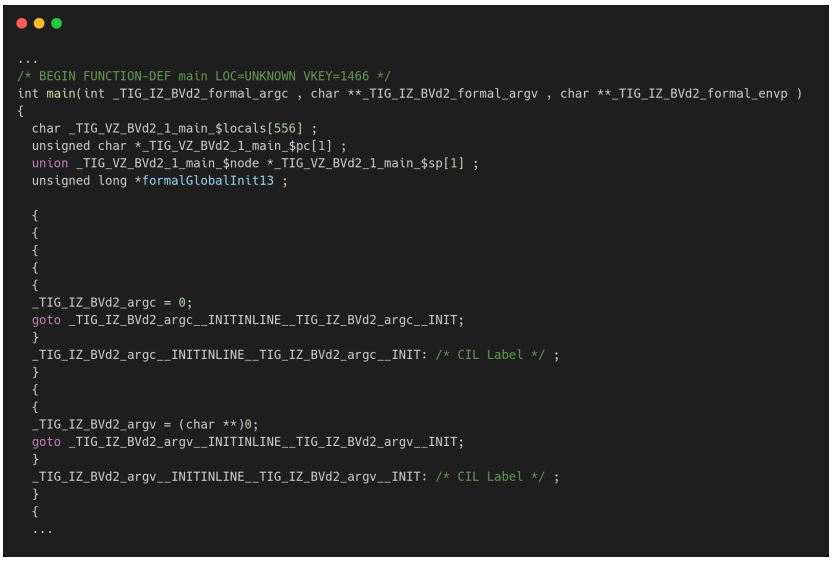
For the second program, `foo_shell.c`, the goal was to make the second call to `foo` execute the provided shellcode. Once the page permissions were updated, the shellcode bytes were copied directly to the address of `foo`. The first call behaved normally, while the second call executed the injected payload and spawned a shell.

2 Part 2

For this part we used Tigress to apply several obfuscation passes on small C programs. A first simple example was created with one integer assignment and an increment. Using the Virtualize transformation on the `main` function produced a source file in which the original code is replaced by a virtual machine.

The result contains a bytecode array, a program counter, temporary variables acting as a virtual stack, and a dispatcher loop with a `switch` statement that interprets each bytecode instruction. Running the transformed binary yields the same output as the original program, but stepping through it in `gdb` shows that execution follows the dispatcher instead of the original control flow.

We then built a second program with a non-trivial control flow: a loop, nested conditional statements, and a final check after the loop. On this example we applied two transformations to the function `f`: AddOpaque and Flatten. The opaque predicates introduce additional conditional branches whose truth values remain fixed but cannot be inferred by static analysis. The flattening pass removes the structured control flow and replaces it with a single dispatcher



```
...
/* BEGIN FUNCTION-DEF main LOC=UNKNOWN VKEY=1466 */
int main(int _TIG_IZ_BVd2_formal_argc , char **_TIG_IZ_BVd2_formal_argv , char **_TIG_IZ_BVd2_formal_envp )
{
  char _TIG_VZ_BVd2_1_main_$locals[556] ;
  unsigned char *_TIG_VZ_BVd2_1_main_$pc[1] ;
  union _TIG_VZ_BVd2_1_main_$node *_TIG_VZ_BVd2_1_main_$sp[1] ;
  unsigned long *formalGlobalInit13 ;

  {
  {
  {
  {
  {
  _TIG_IZ_BVd2_argc = 0;
  goto _TIG_IZ_BVd2_argc__INITINLINE__TIG_IZ_BVd2_argc__INIT;
  }
  _TIG_IZ_BVd2_argc__INITINLINE__TIG_IZ_BVd2_argc__INIT: /* CIL Label */ ;
  }
  {
  {
  _TIG_IZ_BVd2_argv = (char **)0;
  goto _TIG_IZ_BVd2_argv__INITINLINE__TIG_IZ_BVd2_argv__INIT;
  }
  _TIG_IZ_BVd2_argv__INITINLINE__TIG_IZ_BVd2_argv__INIT: /* CIL Label */ ;
  }
  {
  ...
```

Figure 1: Excerpt of the virtualized `main` showing bytecode and dispatcher

controlled by a state variable. The output of the transformed program remains unchanged.

Finally, we applied a data transformation on the same function. Using EncodeLiterals modifies the way constants are represented by replacing them with more complex expressions or helper variables. This changes the data layout without affecting the computation.

```
┌──(blackjack㉿Blackjack)-[~/Documents/Projects/Code-de--Obfuscation/src]
└─$ tigress \
  --Transform=InitEntropy --Functions=main \
  --Transform=InitOpaque  --Functions=main \
    --InitOpaqueStructs=list,array \
    --InitOpaqueCount=3 \
  --Transform=EncodeLiterals \
    --Functions=f \
  ex2.c --out=ex2_data.c

Environment defaulted to x86_64, Linux, Gcc.
Use the "--Environment=" option to override the default.
<<Tigress>> Using compiler:
<<Tigress>> Loading definitions of machine dependent types from: /usr/local/bin/tigresspkg/4.0.11/machdeps_json/x86_64_Linux_Gcc_0.json

<<Tigress.WARNING>> Dropping '__attribute__ ((__malloc__ (..., 1)))' attributes from 'reallocarray'.

┌──(blackjack㉿Blackjack)-[~/Documents/Projects/Code-de--Obfuscation/src]
└─$ gcc -no-pie -fno-stack-protector ex2_data.c -o ex2_data
./ex2_data

res = 5
```

Figure 2: Output after applying a data transformation (EncodeLiterals)

The resulting code stays functionally correct but becomes significantly harder to understand or analyse statically.

3 Part 3

The last part was a small CTF based on a single file called `payload` in the `Ctf1` directory. Using `head` and `file` showed that it was base64 text. After decoding it with `base64 -d payload > payload.dec` and renaming, we used `gunzip` and `tar` to obtain a POSIX tar archive containing two files, a stripped ELF binary `ctf` and a bitmap file `67b8601`

The bitmap was recognised as a 512x512 24-bit Windows BMP, but a search for the ELF magic using `grep -oba ELF 67b8601` revealed an embedded ELF header at offset 53. We then used `dd` to extract the library, first for analysis and then with the correct size into `lib5ae9b7f.so`. The `file` and `nm -D --demangle` commands confirmed that it was a small 64-bit shared object containing RC4-related functions.

We then used `ltrace` to follow the library calls: `ltrace -i -C -s 100 ./ctf show_me_the_flag`. The trace showed a first call to `rc4_decrypt` on a constant string and a subsequent call to `getenv("GUESSME")`, followed by another `rc4_decrypt` that used the environment variable as input. With `GUESSME=test` we obtained the message `guess again!`, confirming that the program compared the decrypted value to a hidden reference

Following the tutorial, we set a breakpoint near the second decryption and used `gdb` to inspect the decrypted content in memory . This revealed that the corect value for the environment variable was the string `"Crackers Don't Matter"`.

Finally, running `GUESSME="Crackers Don't Matter" ./ctf show_me_the_flag` produced the final flag `flag = 84b34c124b2ba5ca224af8e33b077e9e`.

```
┌──(blackjack㉿Blackjack)-[~/…/Code-de--Obfuscation/src/CTF/Ctf1]
└─$ gdb ./ctf
Breakpoint 1, 0x0000000000400dc8 in ?? ()
(gdb) info registers rcx
rcx            0x6156e0            6379232
(gdb) info registers rax
rax            0x0                 0
(gdb) x/s 0x400dc8
0x400dc8:       ":\024\001t\023\277\257\021@"
(gdb) x/s 0x6156e0
0x6156e0:       "Crackers Don't Matter"
(gdb) exit
A debugging session is active.

        Inferior 1 [process 336377] will be killed.

Quit anyway? (y or n) y

┌──(blackjack㉿Blackjack)-[~/…/Code-de--Obfuscation/src/CTF/Ctf1]
└─$ GUESSME="Crackers Don't Matter" ./ctf show_me_the_flag

checking 'show_me_the_flag'
ok
flag = 84b34c124b2ba5ca224af8e33b077e9e
```

Figure 3: Flag successfully captured

For the second CTF (ctf2), we tried to run the provided oracle binary but it requires `libcrypto.so.1.0.0` with the old `OPENSSL_1.0.0` ABI. This version is no longer available on a standard Kali install and loading it manually would require installing an obsolete OpenSSL 1.0 runtime or rebuilding the binary in a controlled environment. We did not want to break the system or mix incompatible OpenSSL versions, so we stopped there. The intended approach would be similar to ctf1: load the correct library, inspect the behaviour of the oracle, and reverse the logic used to validate the input.