hardcode

Unlike previous binaries, we won't do any disassembly using `gdb`. Rather, we will use this challenge to demonstrate Ghidra instead. This does not replace using `gdb` for dynamic analysis, but is a good tool for static analysis.

We first use `checksec` to see what security features are enabled:

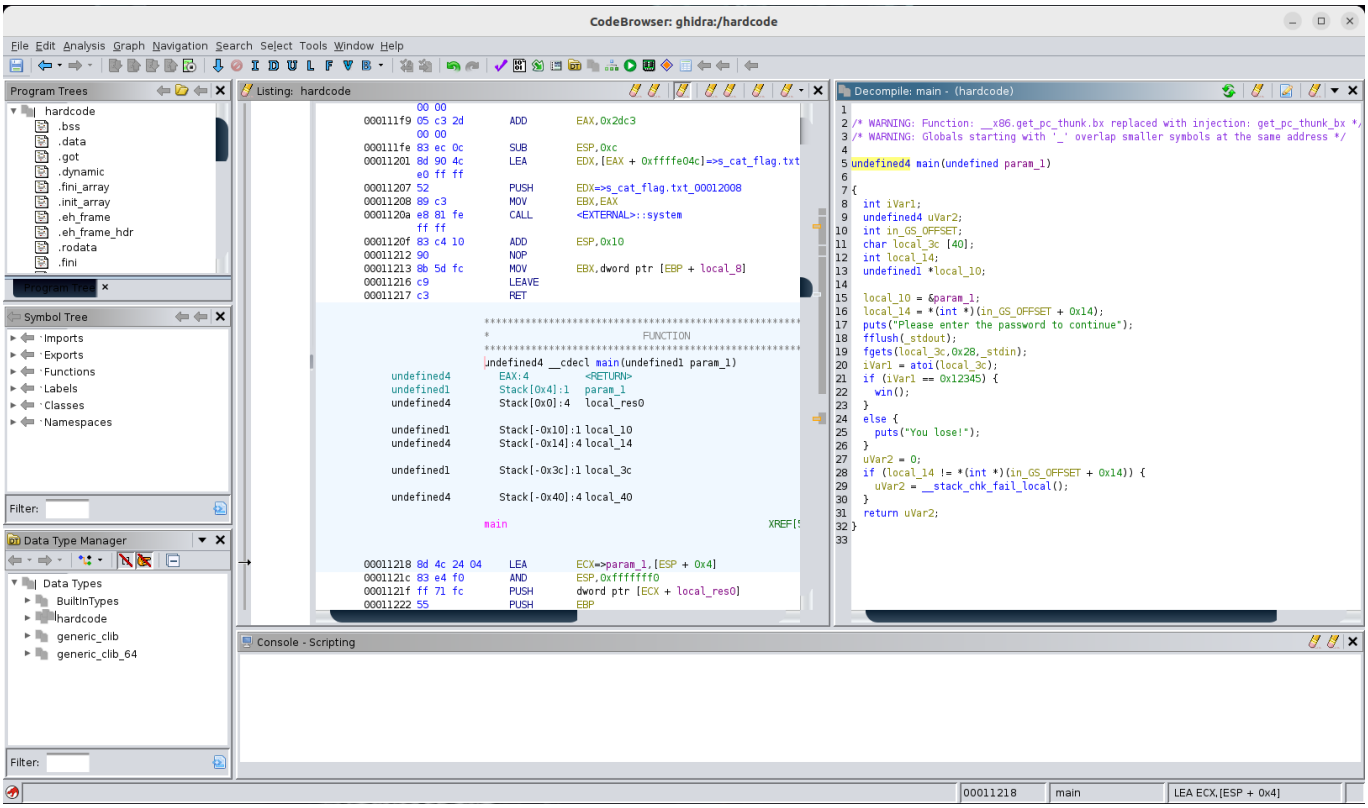
```
$ checksec hardcode
[*] '/home/joybuzzer/Documents/vunrotc/public/reverse-engineering/11-ghidra/hardcode/src/hardcode'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

This binarty is compiled with all protections enabled. We can't stack smash, we can't overwrite the GOT table, and we can't use absolute addressing. What can we do? *Not much.*

The most often solution to dealing with fully-protected binaries is by using the binary as expected, and crafting input that reaches an unexpected state. Ghidra helps us understand the control flow of the program.

Static Analysis

We can use Ghidra to disassemble the binary. If we open the binary in Ghidra, we see the following:



When looking at Ghidra, we notice a few things:

- Ghidra resolves all the function arguments for us.
- It chooses names of variables based on their location on the stack. This is helpful, but we can rename them to make them more meaningful.
- Ghidra always declares variables at the top of the function and then uses them later.
- Ghidra shows the `main()` function at first. The *Functions* folder on the left side lets us access the rest of the function. We can also double-click a function to jump to it.
- As we click on lines of code, Ghidra highlights the corresponding assembly code. This is helpful for understanding what the decompilation is doing.
- Sometimes Ghidra doesn't understand what data type a variable is. It will name it `undefined` plus the number of bytes it thinks the type is.
- Ghidra will not disassemble library functions by default because we did not provide the library file.

Below is the default disassembly of `main()` provided by Ghidra:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */
/* WARNING: Globals starting with '_' overlap smaller symbols at the same
address */
undefined4 main(undefined param_1)
{
  int iVar1;
  undefined4 uVar2;
  int in_GS_OFFSET;
  char local_3c [40];
  int local_14;
  undefined1 *local_10;
```

```
  local_10 = &param_1;
  local_14 = *(int *)(in_GS_OFFSET + 0x14);
  puts("Please enter the password to continue");
  fflush(_stdout);
  fgets(local_3c,0x28,_stdin);
  iVar1 = atoi(local_3c);
  if (iVar1 == 0x12345) {
    win();
  }
  else {
    puts("You lose!");
  }
  uVar2 = 0;
  if (local_14 != *(int *)(in_GS_OFFSET + 0x14)) {
    uVar2 = __stack_chk_fail_local();
  }
  return uVar2;
}
```

We can clean this function by renaming variables, defining undefined types, and removing unnecessary variables. We can also remove the canary check at the end and simply track that the binary has a canary. Since we see that `fgets()` is a secure call (`0x28 == 40`), we know there is no chance this function overflows the buffer. Here is the cleaned up version of `main()`:

```
int main(void)
{
  int intBuffer;
  int canaryOffset;
  char buffer [40];

  puts("Please enter the password to continue");
  fflush(stdout);
  fgets(buffer, 0x28, stdin);
  intBuffer = atoi(buffer);
  if (intBuffer == 0x12345) {
    win();
  }
  else {
    puts("You lose!");
  }

  return 0;
}
```

This function is easy to dissect. We are offered `40` bytes of input. This input is converted to an integer and compared with `0x12345`. If our input matches, it calls `win()`. Let's check out `win()`:

```
void win(void)
{
```

```
    system("cat flag.txt");
    return;
  }
```

This does exactly what we'd expect it to.

Beating the Binary

Unlike the binex section, our goal here isn't really to exploit the binary. Our goal is to craft the right input to pass all the checks and reach the `win()` function. In this case, we know we need to pass `0x12345`.

If we try to pass `0x12345` as input, we get the following:

```
$ nc vunrotc.cole-ellis.com 11100
Please enter the password to continue
0x12345
You lose!
```

Why doesn't this work? `atoi()` converts from a string to an integer. This function does not understand hex. Instead, we can pass the integer equivalent of `0x12345`:

```
$ nc vunrotc.cole-ellis.com 11100
Please enter the password to continue
74565
flag{that_was_easy}
```

This gives us the flag! we can also write this script using pwntools:

```
from pwn import *

proc = remote('vunrotc.cole-ellis.com', 11100)

key = 0x12345

proc.sendline(str(key).encode())
proc.interactive()
```

If we print our payload (i.e., `print(str(key).encode())`), we'd see our payload is `b'74565'`.