Create a detailed report solving all level 1-3 difficulty.

Report must contains:

- How binary was vulnerable to buffer overflow?  
- What is PIE option on checksec and What happens if it's disabled and enabled?

- Debugging with Ghidra (optional)

- Debugging with GDB (Finding offset)

- Exploit Code

Try explaining what was the bug and the idea to exploit the bug in detail

How binary was vulnerable to buffer overflow?

Figure Number of static function

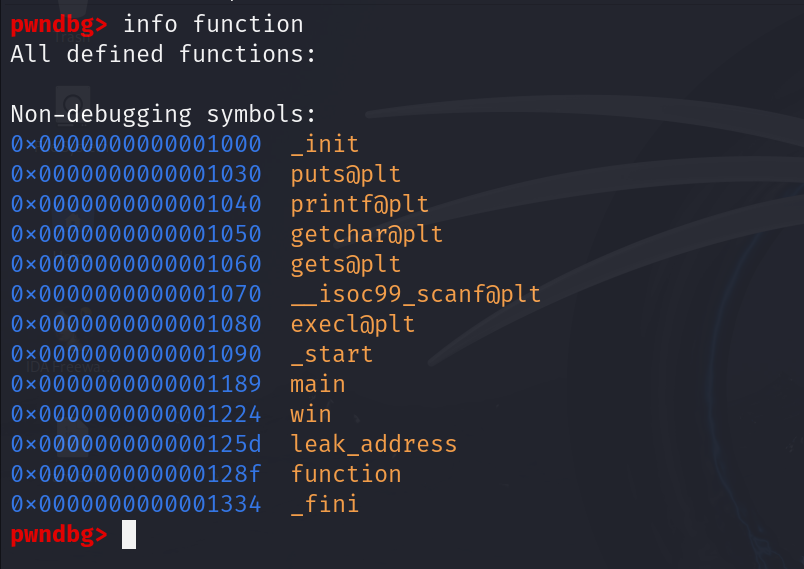


Figure : Disassemble instruction of 'function' gets

A screen shot of a computer code

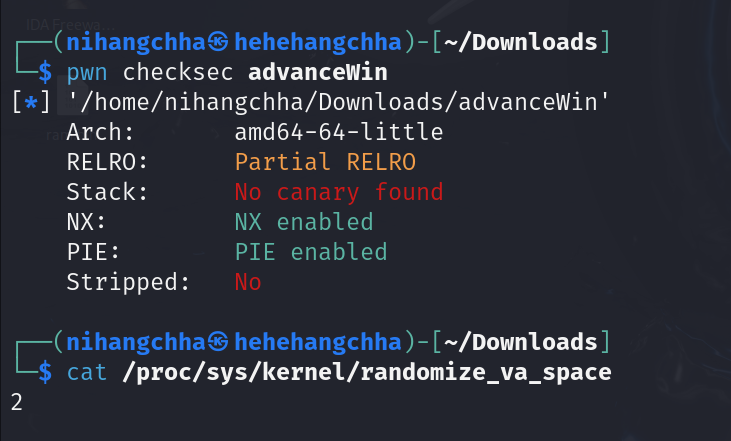
Description automatically generated

The provided ELF executable have 4 static function, ‘main’, ‘win, ‘leak\_address’, and ‘function’ as show in above figure 1. Among them the ‘function’ function was a vulnerable function because of the use of implicit function ‘gets’, it does not have limit of input take from the stdin. Function which was pushed into the stack, its memory region of local variable, old base pointer and even return address can be overwritten.

What is PIE option on checksec and What happens if it's disabled and enabled?

PIE (Position-Independent Executable) option in the checksec tool indicates whether a binary is built as a position-independent executable. PIE takes the advantage of Address Space Layout Randomization (ASLR**)** techniques, which makes it harder for attackers to predict memory addresses during exploitation attempts.

Figure : Pwntool checksec of binary executable when PIE and ASLR enabled



**When PIE is enabled**, the binary executable is loaded and its stack, heap, code and data segments are mapped into virtual address space at random memory addresses every time it is executed. This means it’s hard to predict the correct memory address of certain segments.

**When PIE is disabled**, the binary executable stack, heap, code and data segments loaded and mapped into virtual address space are always at the same memory address. This means all of the segments are placed at predictable locations.

**LEVEL1:**

Figure : Generating cyclic pattern and sending them as input

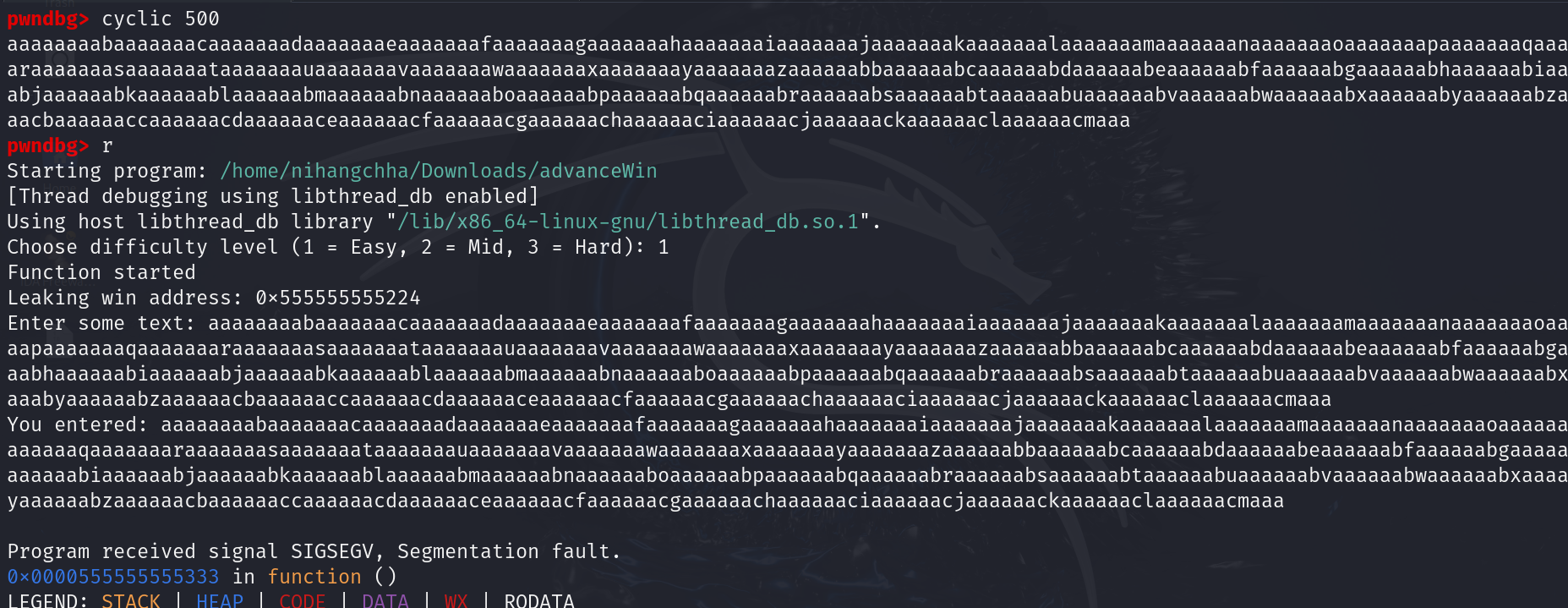


Figure : Overwriting the data of return 'function'

A computer screen shot of text

Description automatically generated

Figure : Finding the offset of function return address

A purple text on a black background

Description automatically generated

From the figure 4, as we can see the address of win function was being leaked which make come handy while creating a payload.

Figure : Creating a lvl1 payload

A computer screen shot of a computer code

Description automatically generated

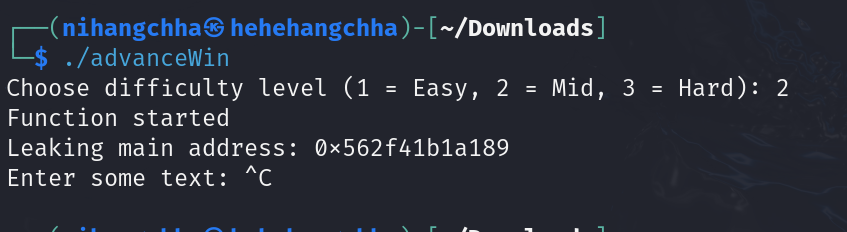
Figure : Shell spawn

A screenshot of a computer

Description automatically generated

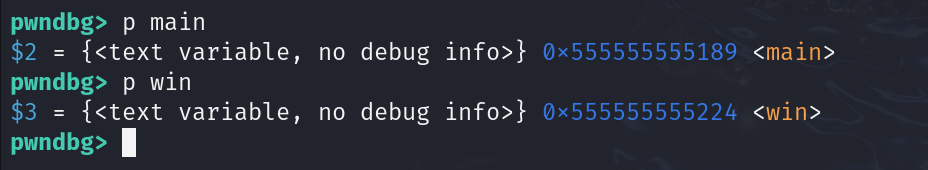
**LEVEL 2**

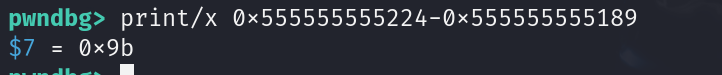
Figure : Leak address in LVL2

****

In level 2, the address of main function was being leaked so, we need to find the offset between main and win function.

Figure : Offset between main and win function





The main function was ‘0x9b’ from the ‘win’ function as show in figure 10. Now creating a payload that can trigger the ‘win’ function.

Figure : Payload of lvl2



Figure : Spawning the shell by launching the python file

A screen shot of a computer

Description automatically generated

**LEVEL 3**