Info Security Project-1

Sandesh Katta

Vulnerability 1

1. This program takes in a single command line argument. It then takes this argument and passes it to the `call_vul` function. The `call_vul` function then calls `vulnerable` function with this argument. The `vulnerable` function contains a local char array of size 100 bytes (`buf`). It then uses `strcpy` to copy the argument into the buffer `buf`.

2. The vulnerability here is in line 7 (specifically the call `strcpy(name, arg)`). `buf` is only of size 100 bytes, but the program can take in an input of arbitrary size and copy it into the buffer `buf`. If this value is too large (and larger than 100 bytes), this could overwrite adjacent memory locations and potentially important information like the return address, thus causing unexpected behaviour.

3. Our goal is to load the value of buffer address into the location of the vulnerable function's return address to open the shell. In order to do this, I created a python file (sol1.py) and crafted an input string of 116 bytes, as this allowed me to overrun the `buf` enough to write the return address. I figured these sizes and return address using GDB and print statements. The first 25 bytes of my string contained NOPS (0x90) because those are pretty useful filler bytes for exploit code, then next 53 bytes contain the shell code, followed by 34 bytes of NOPS (0x90) followed by the buffer address (This value overrides the return address).

4. Please see my submitted code.

5. For this specific program, a number of things could be done to prevent this from happening. One solution is to check that the length of the program argument is less than 100 bytes, so that we are not able to copy more data than the buffer can fit. A more permanent and general solution would be to use `strncpy` instead of `strcpy` and specify the max number of characters to copy.

Vulnerability 2

1. This program takes in a single command line argument. It then takes this argument and passes it to the `call_vul` function. The `call_vul` function then calls `vulnerable` function with this argument. The `vulnerable` function contains a local char array of size 1024 (`buf`), int pointer `p`, int `a`. It then uses `strncpy` to copy the program argument into the name `buf` and load value of `int a` to address pointed by `p`.

2. The vulnerability here is in line 12 (specifically the `strncpy(buf, arg,sizeof(buf) + 8);`). the program will take in an input of arbitrary size and copy it into the buffer `buf` but given limit is 8 bytes from mort than size of destination(`buf`) . If input is value is too large (and larger than 2048 bytes), this will overwrite adjacent memory locations int pointer `p` and int `a`, thus causing unexpected behaviour when line `*p = a;` executes.

3. In order to do this, I created a python file (sol2.py) and crafted an input string of 2056 bytes, as this allowed me to overrun the `buf` enough to overwrite int pointer `p` and int `a`. I loaded `vulnerable` function return address into `p` and buffer address into `a` values I figured out using GDB and print statements.

   The first 27 bytes of my string contained NOPS (0x90) because those are pretty useful filler bytes for exploit code, then next 53 bytes contain the shell code, followed by 1968 bytes of NOPS (0x90) followed by the return address and buffer address (This value needs to be overridden in the return address).

4. Please see my submitted code.

5. For this specific program, a number of things could be done to prevent this from happening. One solution is to check that the length of the program argument is less than 2048, so that we are not able to copy more data than the buffer can fit. A more permanent and general solution would be to set write right limits in `strncpy` . Limit should be always less than size of destination, in this case limit should not more than 2048. And line `*p =a` has no functional use hence it should be avoided since even though if we avoid buffer overflow `p` and `a` contain garbage values and could potentially corrupt stack or heap.

Vulnerability 3

1. This program takes in a single command line argument. It then takes this argument and passes it to the `read_file` function. The `read_file` function reads the first 4 bytes from the file and loads into a variable `count`. It then allocates the memory of size of unsigned integer times `count` buffer(`buf`) in the stack, afterwards it calls the function `read_elements` where it reads elements from the file and loads into `buf` for `count*4` bytes or untill `EOF` is reached.

2. The vulnerability here is in line 26 (specifically the `unsigned int *buf = alloca(count * sizeof(unsigned int));`). As the value of `count` times the size of an unsigned integer is sent as an argument, this multiplication can lead to integer overflow and thus causing disparity between size of buffer created and `count*4` bytes . This disparity can cause buffer overflow in `read_elements` function call.

   other vulnerability even if `count*4` doesn't overflow if this value is near `0xFFFFFFFF` `alloca` would return rotate whole ram and return address which is higher than this `read_file` function frame itself and enter `call_vul` function frame. then one can overwrite the `call_vul` return address and unexpected behaviour.

3. Our goal is to give a large value to the count such that when multiplied by 4, it causes an overflow(more than 4 bytes). In order to do this, I created a python file (sol3.py) and crafted an input integer `\x4000019` (when multiplied by 4, it causes an overflow and the value of the unsigned integer will be 1024). Then input string is taken such that the first 15 bytes contain NOPS (0x90), followed by 53 bytes of shell code, followed by 1000 bytes of NOPS (0x90). Then the address of the buffer to be placed into the return address (which I figured out using gdb and checking the values of $esp and buff)

4. Please see my submitted code.

5. For this specific program, a number of things could be done to prevent this from happening. One solution is to check for the integer overflow before allocating the memory using `alloc` function and also making sure requested memory is within limits of process memory.

Vulnerability 4

1. This program takes in a single command line argument. Then using file `/dev/urandom` and creates a random number `r` which is between 0 to 255 and it allocates this amount of byte space on stack Then it passes our argument to `vulnerable` function. `vulnerable` function copies this argument to buffer(`buf[1024]`) using `strcpy` similar to code1.

2. The vulnerability here is in line 7 (specifically the call `strcpy(buf, arg)`). `buf` is only of size 1024, but the program will take in an input of arbitrary size and copy it into the name buf. If this value is too large (and larger than 1024 bytes), this could overwrite adjacent memory locations and potentially important information like the return address, thus causing unexpected behaviour.

3. our goal is to modify return address of `vulnerable` function and point to our shellcode. But the catch here is due to randomization buffer address changes with every run.

   In order to do this, I created a python file (sol4.py) and crafted an input string of 1040 bytes ,as this allowed me to overrun the but enough to write the return address (I figured this out using GDB and print statements). The first 500 bytes of my string contained NOPS (0x90) because those are pretty useful filler bytes for exploit code, then next 53 bytes contain the shell code, followed by 39 bytes of NOPS (0x90) followed by the buffer address (This value needs to be overridden in the return address) .

   I calculated the buffer address using gdb and print statements. I added 255 NOP to this address this gives enough buffer so that even if random number is max(255) my start of my shell codes is greater than my return address I just calculated s In any case this facilates execution of my shell code.

4.

5. For this specific program, a number of things could be done to prevent this from happening. One solution is to check that the length of the program argument is less than 1024, so that we are not able to copy more data than the buffer can fit. A more permanent and general solution would be to use `strncpy` instead of `strcpy` which specifies the max number of characters to copy.

Vulnerability 5

1. This program take the input file(`sol5_input.txt`)as command line argument. We have buffer same as size of this input file(size is calculated using `get_file_size` function). Then, `launch()` is used to read the data from the file and then accept a set of user commands to read or write onto the file through this buffer.
2. In the write operation, the attacker can select an offset such that it performs a return-to-libc attack that opens the shell as root user.
3. Since DEP is enabled, buffer overflow cannot not help us to load the return address to an executable code. So, we need to find the offset such that we move beyond the buffer and override the location with 4 bytes of system call (0xb7e20b40) ,4 bytes of exit() (0xb7e13b40) and 4 bytes of address of /bin/sh (0xb7f60aaa).
4.
5. For this specific program, Just DEP is not sufficient to prevent the attack. one way to avoid this could be by using a shadow stack with DEP