CSE545 Project REPORT

Omkar Totade (ASU ID- 1209345380)

Mohit Nangare (ASU ID- 1209409068)

Format Strings Vulnerabilities Prevention

Format Strings Vulnerabilities-

Format Strings vulnerabilities is a type of security vulnerability that exploits the 'printf' family of functions, and can either crash the program or can execute arbitrary (mostly harmful) code. The 'printf' family of functions perform formatting on the user input using various format tokens like '%x', '%u', '%d', etc. While all other format specifiers will only print data from the call stack, the '%n' format token, used with the 'printf' family of functions, performs the task of writing the number of bytes that are formatted, to arbitrary memory locations on the stack. An exploit involves use of such format specifiers, more importantly the '%n' specifier to write the address of our shell code to a memory location on the stack, so that the program will somehow jump to that address and execute our shell code.

Describe the problem that you are solving-

Our project aims at handling the phenomenon of programs writing to arbitrary memory locations using the '%n' format specifier inside one of the 'printf' family functions. Also, by doing this we do not let the execution of the program be affected and hence prevent the program from crashing (a successful format strings exploit is

also used to crash programs). We implement this, by overloading the existing functions from the ‘printf’ family, with our own versions of the ‘printf’ family functions.

Describe the approach you took to implementing the project-

Letting the program write some arbitrary value to memory is the root cause of format string vulnerabilities. Writing to memory in itself is not bad, but deciding what is to be written to memory based on user input was something we did not want to happen. We thought of an approach that would attack this problem at its root cause, which is using the format specifier ‘%n’ in a ‘printf’ family function. If we could simply sanitize user input before providing it to the original ‘printf’ function, we could stop malicious users from providing unwanted ‘%n’ to the program and thus prevent the program from writing to any memory on the stack. While doing this, we also ensured that the functionality of other format specifiers like ‘%d’, ‘%x’, ‘%u’, etc. does not change and the program does not behave in an unexpected manner.

We implemented all the above test cases in our own version of the ‘printf’ family functions which we developed using the LD_PRELOAD interface, so that it will be easy for the developer to use our library. The developer could simply load our ‘safeprintf’ library before the standard C library is loaded, and run his program using our preloaded ‘printf’ family functions, and it would make his program safe from format strings vulnerabilities.

We first implemented the ‘printf’ function using LD_PRELOAD. The ‘printf’ function is a variadic function and thus we need to start and end a variable argument list to access the variable arguments one at a time. The way we went about it was, we checked for the occurrence of a ‘%’ character in the format, and checked the value that came immediately after the ‘%’. If it matched one of the valid format specifiers (d, x, u, c, s, etc.) we extracted an argument of the corresponding type (‘int’ for character ‘d’ and so on) from the va_arg list (variable argument list) and called the original ‘printf’ function with it. We parsed the entire format string in this manner

and mapped the corresponding format specifiers to their particular types, so that the original functionality of the program remains same.

We sanitize the user input to remove all excess number of '%n' from it, before providing it to the original 'printf' function which would then behave normally. We also covered cases where the user can use various other valid characters in between '%' and 'n', characters like '*' and '.' and numbers from 0-9. So, our program would ignore all the combinations of characters that would execute like a '%n'. For example,

printf ("cse545%3.n\n");

The printf statements given above would still write the length of "cse545" to a memory location on the stack. Our solution handles this case, sanitizes the input and removes the characters "%3.n" from the string, thus preventing 'printf' from writing to any memory location. If there are any characters between '%' and 'n', which are not valid characters, our library will consider it as a part of the string and print it.

We also handled the case where "%hn" writes to a short int argument and "%hhn" can be used to write to an unsigned char argument. In both these cases, the 'printf' function will write to a certain stack location if there is no corresponding argument. For example,

printf ("cse545%hn\n");

In the above code, printf would write the length of "cse545" as short int to a memory location on the stack, thus making the program vulnerable. Our library cleans the string to remove all possible combinations of "%hn" and "%hhn" and prevent the program (actually the attacker) from misusing these to write to memory.

A '%' character is used as an escape character in C, and thus can be exploited by the attacker. He can provide an input string that has "%%n" and attempt to write his shell code address to memory. But our library also handles that case, wherein if there is an escape character, we would print the character after it as a string, thus sanitizing the input and preventing memory write.

Thus, our approach was to anyhow prevent all memory writes by removing and sanitizing the "%n" format specifiers and all types of inputs that the user could possibly use to write to some memory on the stack, and not letting the functionality of other format specifiers be affected, so that the program could run smoothly.

Once we got everything working for the 'printf' function, we could similarly construct the 'fprintf' function wherein he just had to provide a file argument, we could construct the 'sprintf' function wherein we had to provide a string argument, and finally we constructed the 'snprintf' function which was very similar to the 'sprintf' function just that we had to copy only the given size of data. Further, working on implementing the 'vprintf', 'vfprintf', 'vsprintf', and 'vsnprintf' was similar too, with a few minor modifications because of the non-variadic nature of these functions.

Describe the limitations of your tool: what applications can it be applied to?

There is only one significant limitation in our tool, which was actually a decision that we took as a trade-off since we chose that it is better that our program would run with incorrect output rather than crashing and terminating. Consider the following case,

```
int main (int argc, char *argv[]) {

int len;

printf ("cse545isthebest%n\n", &len);

printf ("%d\n", len); }
```

In this case too our solution would sanitize the format string and remove the '%n' from it. The problem here is, we need to calculate the length of the format string and store the value in the variable 'len'. Due to our implementation approach, we can calculate the length of the format string, but we are unable to set it back to the 'len' pointer. This happens because, we cannot always be sure whether a corresponding pointer argument

will be present for a particular '%n' used in the format string, and thus in the case when a corresponding pointer argument does not exist, assigning the length value to some arbitrary pointer would give us unexpected results (segmentation fault). As a result, we decided to do a trade-off here, such that we would read the corresponding argument for the '%n' that is used, but we would not write anything to memory, since we are of the view that writing to memory is harmful and vulnerable. A program that runs completely with an incorrect output is always better than a program that crashes or gives a segmentation fault. Thus, with our library the above code would execute completely without crashing, but would provide an incorrect output-

cse545isthebest

0

Thus, in this case, the developer will not be able to use our library and get correct output, and he might have to use some other function like 'strlen' to calculate the length of the string and assign it to the 'len' variable.

Also, due to time constraint and complexity of the implementation, we decided to not support the flags, width, precision and field length used along with format specifiers in the 'printf' family functions. Hence we ignore all the numbers and some special characters that may be present between the '%' sign and a valid format specifier alphabet. For example the following code,

printf ("%1000d%.2f", 12345,1234.1234);

would treat "%1000d" as a normal "%d" and display, **12345,** and treats "%.2f" as normal "%f" and prints, **1234.1234.**

Describe the future work that you can see which would improve your tool-

We were unable to include the functionality of flags, width, precision and field length due to a time constraint but that is something that we definitely can work on in the future, but since it was not critical to the core

problem (writing to memory) we decided to put it aside for a while. This can be included in the future work on the tool, so that the tool will eventually behave exactly like the original 'printf' family functions, and would prevent the format string vulnerabilities attack. Another thing which we can work on in the future is the memory leakage part which was a part of the stretch goal. Once we figure out a way to calculate the number of arguments passed to the 'printf' function, working on the memory leakage part would be easy.

References-

1. https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

2. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.2725&rep=rep1&type=pdf

3. https://homes.cs.washington.edu/~djg/papers/format_string.pdf

4. http://fluxius.handgrep.se/2011/10/31/the-magic-of-ld_preload-for-userland-rootkits/