Project Write-up: Automating Exploitation of Format String Vulnerabilities

Prateek Jain

April 29, 2015

1 Introduction

A number of binary exploitation challenges in CTFs require an attacker to alter the control flow of a program and make it do things which it should not. Format string vulnerabilities are one such class of problems. A format string vulnerability can be exploited by feeding specially crafted user-inputs to the program which can help the attacker to perform attacks ranging from viewing the stack contents to writing arbitrary data at arbitrary locations.

Exploiting a format string vulnerability is generally simple and straightforward. A number of basic problems in this area can be solved by following similar steps. The main task for the attacker is to identify if the program has such a vulnerability. Sometimes this can be very obvious, but at other times it might take the attacker a long time to find one.

I proposed to build a tool which would help to automate the process of finding and exploiting format string vulnerabilities. The tool takes the program binary as input and presents the user with an interactive menu. The user can select options ranging from - finding if the binary has a format string vulnerability to creating exploit strings if the user provides the program with values such as address to overwrite, value to write and the offset between the format string and its address. The tool also provides the user with an option to automatically certain values like the value of the offsets and the addresses to overwrite (in case of a GOT or DTOR) address overwrite.

In all, the tool helps the user to exploit format string vulnerability in a fewer number of steps than required otherwise.

2 Approach

As mentioned in the introduction, the main task for an attacker would be to find a format string vulnerability in the program. For this, one has to check if the first argument supplied to a format function, such as a printf, can be user controlled. For e.g.,

Program:

```c
1. void main(int argc, char **argv)
2. {
3.     printf("Hi there!");   // Safe
4.     printf(argv[1]);       // Unsafe
5. }
```

In the above example, the printf at line 3 is safe as the argument passed to printf is not controlled by the user and is a constant string literal. However, the printf at line 4 is unsafe as the first and only argument that is passed to it fully user controlled. In this case, the attacker can craft an input in such a way that the return address stored over the stack frame of the main() function is overwritten with an address that the attacker wants. Once the program returns from main() function, the control flow is now in the hands of the attacker.

The tool separates the safe use of format functions from the unsafe use of these functions. To do this, I had planned to follow one of the following two approaches:

a. Taint Checking: Done at runtime, the basic concept behind taint analysis is to flag anything that has been input by a user and consider the input as something which poses a potential risk. Any other value, which is derived from this user input, is also flagged or monitored. If any of these variables can be used to execute a dangerous command, the taint checker lets the user, in this case the attacker, know of the potentially dangerous tainted variables which he can control.

b. Dynamic Analysis of calls to printf (and other format functions): When the first argument to printf is a constant format string, it is stored in the read only memory section. It thus, cannot be modified and a format attack is not possible. However, if the memory location used as the format string is in the read-write memory, it can be modified and used in an attack. The tool in this case, will then run the program and present the attacker with the printf statements that had their first argument in the read-write memory section and not in the read-only memory section (or .rodata).

After some analysis I decided to use **(b)** to find the vulnerable calls to format functions.

Using LD_PRELOAD:

To do dynamic analysis of the calls to printf and other format functions I took the help of the LD_PRELOAD environment variable. I wrote a C program with a modified implementation of printf and exported that as a shared library using the LD_PRELOAD variable. Whenever the binary executed after this, any calls to the printf function reached the function defined by my library. In this function, I saved the addresses of the first argument of the function to a file and thereafter called the actual printf function.

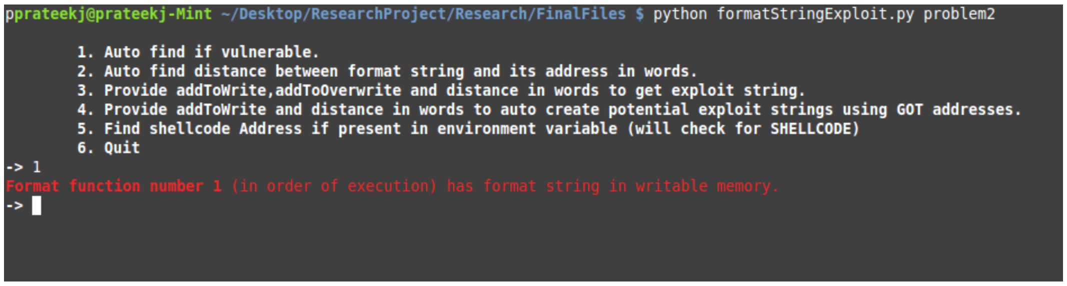
Once the tool gets this addresses, next, it compares these addresses to those found by the tool in the /proc/pid/maps file of the binary when it was being executed. The tool reports a function to be vulnerable if the addresses of the first argument passed to printf had a w bit set in the maps file of the binary.

Shown below are snippets of the output of the maps file and the shared library:

```
08048000-08049000 r-xp 00000000 08:01 298101
08049000-0804a000 r--p 00000000 08:01 298101
0804a000-0804b000 rw-p 00001000 08:01 298101
f7e07000-f7e08000 rw-p 00000000 00:00 0
f7e08000-f7fb1000 r-xp 00000000 08:01 917910
....
```

```c
static int (*orig_printf)(const char *format, ...) = NULL;

int printf(const char *format, ...)
{

        if (orig_printf == NULL)
        {
                orig_printf = (int (*)(const char *format,
                ...)) dlsym(RTLD_NEXT, "printf");
        }

        FILE *fp;
        fp = fopen("address.txt", "a+");
        fprintf(fp,"%p\n",format);
        fclose(fp);

        return orig_printf(format);
}
```

Once a vulnerability is detected, the tool outputs which of the format functions is vulnerable and presents the user with options to do further actions. Below is a snapshot of a sample output of the tool.



```
pprateekj@prateekj-Mint ~/Desktop/ResearchProject/Research/FinalFiles $ python formatStringExploit.py problem2

        1. Auto find if vulnerable.
        2. Auto find distance between format string and its address in words.
        3. Provide addToWrite,addToOverwrite and distance in words to get exploit string.
        4. Provide addToWrite and distance in words to auto create potential exploit strings using GOT addresses.
        5. Find shellcode Address if present in environment variable (will check for SHELLCODE)
        6. Quit
-> 1
Format function number 1 (in order of execution) has format string in writable memory.
->
```

3 Implementation

The tool has been implemented using python. It is a standalone tool and executes subprocesses when necessary. Also, a shared library has been implemented which intercepts the calls to printf, saves the addresses of the first argument to a file and then call the actual printf. This library is implemented in C. Another component that the tool uses is a python-gdb script which is responsible for saving the contents of the maps file which contains the address ranges in the process space and the permissions that have been applied to them.

For the exploit generation, the tool does multiple executions of the program to provide the user with as much useful information as possible, such as distance between the format string and its address on the stack [4], so that the number of values the user has to find and input to the tool are reduced to a minimum.

Limitations:

1. The tool works for problems which takes their input via the command line. It currently does not handle the problems which take inputs via other means such as scanf() or gets().

2. The tool sometimes generates false positives for the vulnerable printf in case a user controlled buffer has been overwritten with a constant value during the execution.

4 Evaluation

Though there are a not a lot of problems related to format string vulnerabilities, I tested it on problems like "Game of Tables" and "Protostar Format 4". Apart from that, I tested the tool on other format string vulnerable problems that I found online. In almost all the cases, the tool was able to save some amount of time which is generally required to solve such problems.
Below is the code for Protostar Format 4. (Modified a little to take the input from command line instead of gets())

Program:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int target;

void hello()
{
 printf("code execution redirected! you win\n");
 _exit(1);
}

void vuln(char* str)
{
 char buffer[512];

 strncpy(buffer, str, sizeof(buffer)-1);
 buffer[sizeof(buffer)]='\0';

 printf(buffer);

 exit(1);
}

int main(int argc, char **argv)
{
 vuln(argv[1]);
}
```

It can be seen that this problem has a format string vulnerability. The tool will try to find this vulnerability dynamically, by running the binary and by analyzing call to the format functions.

In the above example, once the format string vulnerability is found, the tool will provide the user with different options as shown in the snapshot in the **Approach Section**. In the trial run, the tool was able to find the presence format string vulnerability, find the distance in words between the format string and its address and also generate a number of exploit strings by using the address of GOT entries as the addresses to overwrite. These exploits were exported to different files in a directory. A close look at the program shows that the GOT entry for the exit() function is to be overwritten. Selecting that exploit and feeding that to the binary caused the hello() function to be called.

"Game of tables" can be exploited in a similar manner. In case, the exploit is not possible using the GOT overwrite, the user can provide the tool with the address that the user wants to be overwritten, the address to write to and the offset(if the tool was not able to calculate that for some reason). The tool then generates the exploit and saves it to a text file which can be used by the user to provide an input to the binary.

5 Related Work

[1]. Pierre Bourdon. Python GDB tutorial for reverse engineering
http://blog.lse.epita.fr/articles/10-pythongdb-tutorial-for-reverse-engineering—part-.html
This is a nice article which explains how to write a python plugin for GDB. The plugin helps find vulnerable format functions by dynamically analyzing calls to various format functions.

[2]. Thyago Silva. Format Strings (Gotfault Security Community)
http://www.exploit-db.com/papers/13239/
This article explains nicely about format string concepts including direct parameter accesses, format string vulnerabilities and the different ways that they can be exploited in. E.g., writing to memory addresses, overwriting the Dtors Section, Overwriting the Global

Offset table entries etc. The article also shows how to develop a format string builder. This will help in developing the format string exploit generator of my tool.

[3]. Paul Haas. Advanced Format String Attacks
http://www.redspin.com/blog/wp-files/defcon-18-paul_haas-advanced_format_string_attacks_final.pdf
This presentation gives an insight into advanced attacks on format strings. This is of interest as the author mentions a few ways in which format string exploits can be automated, like calculations for the distance between the format string and its address on the stack.