Project Proposal: Automating Exploitation of Format String Vulnerabilities

Prateek Jain

April 15, 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 wish to build a tool which helps to automate the process of finding and exploiting format string vulnerabilities.

The tool, which would just take the program binary as an input, would help the attacker find the vulnerability and present the attacker with the vulnerable call. It will then provide the attacker with an interactive menu, which can be used perform certain tasks. Using this interactive menu, an attacker may instruct the program to try an end-to-end attack which if successful, will present the user with a shell. The menu will also provide the attacker the freedom to provide an address to overwrite to, an address to overwrite and even shellcode to execute once the control flow is hijacked.

In all, the tool will help an attacker to exploit format string vulnerabilities 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:

```
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 then, would have to separate the safe use of format functions from the unsafe use of these functions. To do this, I can follow one of the two approaches mentioned below:

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).

Once a vulnerability is detected, it will be followed by exploitation. For this, as mentioned in the introduction, either the user can choose to let the tool try a full end-to-end exploitation of the problem and present the attacker with a shell. Or, the attacker can choose a number of options which help him exploiting the vulnerability. Following are the options which I plan to include in my tool:

A: Insert a custom shellcode
B: Insert default shellcode, hardcoded in the tool
C: Address to overwrite
D: Value to write
E: Distance in words between the format string and its address on the stack
F: Insert shellcode in environment variable or the format string itself
G: Function name for which GOT entry is to be overwritten.
H: Bruteforce (This will try to exploit the program automatically and give a shell)

3 Implementation

I plan to implement the tool using python. It can be a standalone tool or a python plugin for GDB.I might use the Binary Analysis Platform (BAP) tool [1] to perform the binary analysis and taint tracking for my tool. I can also use the tool developed in [2] which helps identifying vulnerable format functions based on the location (read-only/read-write memory) of the first argument to the format function.

For the exploit generation, I plan to do multiple executions of the program to provide the attacker 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 that an attacker has to find and input to the tool are reduced to a minimum.

4 Evaluation

I will be aiming to automatically exploit a couple of problems using the tool to check for its usefulness. One of the problems is Game of Thrones:

Program:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define MAX_LENGTH 512

void getShell()
{
    gid_t gid = getegid();
    setresgid(gid, gid, gid);
    system("/bin/sh -i");
}

int main(int argc, char **argv) {
    char name[MAX_LENGTH+1];
    if(argc != 2)
       return -1;
    strncpy(name, argv[1], MAX_LENGTH);
    name[MAX_LENGTH] = '\0';
    printf("Welcome back: ");
    printf(name);
    printf("\nToday's quote is:\nWith great power comes great
    responsibility\n");
    exit(0);
}
```

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 using taint analysis/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 mentioned in the approach section. If the user selects the Bruteforce option, the tool will run the binary repeatedly with different inputs and try to provide the user with a shell. By running the program multiple times, the tool can calculate information like the number of words between the format string and its address on the stack. Next, the tool will try to place the shell in the format string itself or an environment variable. As a final step, the tool will search for addresses which can be overwritten to cause the control flow to jump to the inserted shell code. The easiest of the approaches would be to overwrite the GOT entry of a function that is being used after the vulnerable format function.

In case the tool fails to automatically provide a shell, it will return to the interactive mode, showing the user all the values that it was able to calculate during the multiple runs and ask the user to enter the remaining values. The user can choose to ignore the values found and enter custom values for all the fields instead. In this case, the user will be presented with exploit strings that the tool feels would be able to exploit the present vulnerability.

Similarly, for the problem Format:

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int secret = 0;

void give_shell(){
    gid_t gid = getegid();
    setresgid(gid, gid, gid);
    system("/bin/sh -i");
}

int main(int argc, char **argv){
  int *ptr = &secret;
  printf(argv[1]);
  if(secret==1337){
    give_shell();
  }
  return 0;
}
```

In this problem, the value in the secret variable is to be overwritten. The tool will generate an exploit for this problem if the user provides it with three values. The address of the secret variable, the value of 1337 to write, and the distance in words between the format string and its address on the stack.

Alternatively, the user might choose the option to export a different shell code, and jump to that instead of calling the function present in the program. Thus, the tool helps the user

to exploit the same problem in different ways.

Once I am able to use the tool successfully with the above mentioned problems, I will find other format strings problems online and try to exploit them with the tool.

5 Related Work

[1]. David Brumley, Alex Rebert, Edward J. Schwartz, and Edward J. Schwartz "The Binary Analysis Platform"
https://github.com/BinaryAnalysisPlatform/bap
A tool which can be used to perform binary analysis and can be used from python, shell as well as other languages. I can use this in conjunction with my tool to get help in identifying vulnerable format functions by using taint analysis.

[2]. 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.

[3]. 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.

[4]. 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.

6 Updates

6.1 Update 1

I started with the binary analysis phase. I took the help of the tutorial and tool developed in [2] to find arguments to the format functions which might be user derived. The script is a python script which can be used with gdb to analyze the binary. The basic concept is to parse the /proc/pid/maps file of the binary when it is being run. The file consists of regions of virtual memory in a process. Each listing contains the starting and the ending addresses of the region. Also, the permissions for each region is listed (r/w/x/s). The idea then is to find the region in which the format address lies. If the permissions of the region have the write flag set, the script breaks and notifies the user about the format string

being in a writable section. The only drawback is that it can give false positives in some cases. For e.g.,

Program:

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

#define MAX_LENGTH 512

int main(int argc, char **argv){

        char name[MAX_LENGTH+1];
        strncpy(name, argv[1], MAX_LENGTH);
        name[MAX_LENGTH] = '\0';
        strcpy(name,"Hello");
        printf(name);
}
```

In this case, even though the printf(name) is not vulnerable to a format string exploit, still the script reports this call as unsafe. I am looking for ways in which this can be made more reliable. Also, I plan to make further tweaks to make it more user friendly and add the selection menu to perform further operations.

As suggested in the comments last time, I researched about the LD_PRELOAD variable. It specifies a list of user specified shared libraries that are to be loaded before all others. I still have to figure out how to use LD_PRELOAD to find out arguments that are in the bss. I explored the possibility of defining a custom library which would have a custom implementation of different format functions (which would be called instead of the actual format functions by taking the help of LD_PRELOAD), but I did not find it to be useful in this scenario.

As another option for performing the taint tracking, I went through the documentation of [1]. I have installed BAP and looking into the various API's and the PIN trace tool that it provides to see if it can be used in my project. I plan to talk to people who have worked with BAP to get a better understanding of the same. Also, I plan to work with rajivk who is trying to use BAP for the same use case.