Generating exploits for Format String Vulnerabilities

Prateek Jain (prateekj@andrew.cmu.edu)

16th March, 2015

Introduction:

I wish to write a program/script which will try to find format string vulnerabilities (and possibly buffer overflow vulnerabilities) in a binary and try to generate exploits for them in an automated manner.

In its simplest form, the program/script will be able to tell the user if the binary is vulnerable to a format string exploit. Next, it will ask the user to enter the address to overwrite, the value to write to that address, and the number of words required to reach the beginning of the format string. With this information, it will present the user with the exploit string which can be input to the binary. This will save all the calculations that are otherwise required to be done in order to write a value to a location. (I will try to see the number of words required to reach the beginning of the format string can be found in an automated way).

This can then be extended to use shell-codes within format strings or as environment variables. A user can chose to go with a default shell-code available with the program or provide his own shell code. For the default shell code, I am planning to use a shell-code with as few bytes as possible so that it can fit in most of the string buffers. If the user chooses to insert the shell code as an environment variable, then I will try to automate and find the address of this inserted shell-code and present the user with the exploit string having this address.

Furthermore, an option can be provided to the user to overwrite a GOT address. For this, the program will ask the user for the function whose GOT entry is to be overwritten. It will then try to find out the address of this function, and use that in the exploit.

The features mentioned above, when complete will save quite some time when solving format string exploits. I feel this deserves to be automated as format string exploits are generally straightforward and such a tool might be able to help in almost all the problems having such a vulnerability. Though there was only a single problem related to format string exploit in the picoCTF, I will try adding problems from other sources, including the one I created.

Approach:

The tool will provide the user with a number of flags which will perform various operations. E.g., one of the flags might be '-e or --examine' flag. This can be used to present the tool with the binary file in question. On execution, the tool will output if a format string vulnerability exists.

Similarly, I plan to incorporate all the features mentioned in the introduction, and present the user with options/flags to choose from.

Considering, the problem "Game of Tables" which has the following code:

```c
#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. Also, I know that a GOT entry is to be overwritten to solve this (Mentioned in the write-up). If this problem is to be done in a regular way, then first the user would have to start inputting a format string repeatedly till the user finds the distance between the format string and its address on the stack. Next, the user will have to find the address of either the printf() or the exit() function in order to start building the exploit. Once this address is available, the user next finds the address of getShell(). Finally, the user will build the actual exploit by doing calculations that are required to be done in order to write arbitrary values to addresses. A simple calculation mistake leads to rework.

However, if the tool mentioned is used, this exploit string will be generated automatically. Also, if I am able to automate the process of finding the distance between the format string and its address on the stack, then the user will only have to enter 2 addresses, the address which is to be overwritten and the address which is to be written.

Also, in this program, there is a function which spawns a shell. In case such a function is not present, the tool will help by exporting a shell code in an environment variable and then use that address when preparing the exploit.

Implementation and Evaluation:

I plan to implement the tool using python. It can be a standalone tool or a python plugin for GDB. The user will invoke the tool by using one or more of the flags available. E.g.,

exploitFormat.py –e binaryName

This can be used to examine if the binary is vulnerable to a format string exploit or not. I am looking into a couple of approaches that can be taken to implement this. Either I can make a simple parser, which will take the source-code as an input and check if a format function is being called without using format specifiers. Or, I will dynamically analyze calls to the format functions, and find if all the arguments passed to it are in the read-only memory(1). If any of the arguments is not in the read only memory, then a possible format string exploit exists.

Some other options which I plan to include in the script are:

A: Insert a custom shellcode

B: Insert the default shellcode

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)

Suppose, the user selects the flag which does a Bruteforce.

First, the script will try to run the binary by giving it different inputs. The inputs will essentially be some dummy characters followed by either a series of %x's or %x in the form of Direct Parameter Access. After each run, it will compare the output given by the program to the dummy characters inserted at the beginning of the exploit. Once a match is found, the program will have the distance of in words between the format string and its address on the stack.

Next, the script will export the default shellcode in an environment variable. I then plan to use functions like popen() and bash commands like grep, cut etc. to find the address of this inserted shell code.

Once, the address of the shellcode is found, the program now has 2 of the required values. The third value needed is the address to overwrite. Here again, a list of all the library functions being used and their addresses can be found out and extracted from the binary. After that, the program can try to use these addresses one by one, in different runs. I plan to improve this by first finding out the functions that are being used after the vulnerable format function, and use the addresses of those functions.

Lastly, after having all the three values needed, the program will generate a script by performing all the calculations that are required to write an address to an address. This string can then be input to the binary by the program. If this succeeds, the user will be presented with a shell. Otherwise, it will tell the user that automatic exploitation failed. In this case, the user will have to use the other flags available in the program and try to exploit the problem.

I will be evaluating the picoCTF problem, 'Format' first.

```c
#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. This problem should be solved by providing three things to the program. 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.

Another way to solve this would be to use the Bruteforce flag, which will insert its own shell code and spawn a shell automatically. Once this succeeds, I will proceed to solve the next problem, 'Game of Tables'.

I will then try to find other format strings problems online and try to exploit them with the tool.

Related Work:

1. Python GDB tutorial for reverse engineering
Pierre Bourdon
This is a nice article which explains how to write a python plugin for GDB. The article tries to find printf's that are vulnerable to format string exploits in a binary.

2. AEG: Automatic Exploit Generation
Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao and David Brumley

(Just stumbled open it. Have to check if this can be helpful with the tool.)

Updates: