CSE 543 Fall 2013 - Assignment 2

October 8, 2013

1 Dates

- **Out**. *8th Oct 2013*
- **Due**. *22nd Oct 2013 (Tuesday), 11:59PM*

2 Introduction

In this project, your aim is to exploit a format-string vulnerability in the given network echo server to get program control using Global Offset Table (GOT) hijacking. **Note: This project is for educational purposes only, and should *never* be used outside of this class**.

3 Background

Recall that a format string vulnerability allows attackers to read and write arbitrary data. Thus, a format string vulnerability in conjunction with some information leak about addresses in the program can be used to circumvent defenses such as ASLR. With increasing deployment of ASLR, such information leak vulnerabilities are likely to be the future of exploitation [6].

In this exercise, you are given a simple network echo server that contains both a format string vulnerability and an information leak through a function pointer on the stack. You will write a client program that interacts with the server and exploits its format string vulnerability to launch a shell at the server. The simple technique of subverting control flow by overwriting the return address will not work in this case because the server function never returns! Thus, we will use a different technique – global offset table (GOT) hijacking [2] – to subvert the control flow. You will use the leaked information of the function pointer to find the base location of the randomized libc, and redirect the function pointer's GOT entry to a function of the attacker's choice in libc (usually, `system()`).

4 Knowledge Gain

In completing this exercise, you will learn:

- The scope and limitations of various existing defenses and how they can be bypassed by information flow leaks,
- How a real-world format string vulnerability is detected and exploited, and
- An understanding of the GOT hijacking technique to subvert control flow

5 Exercise Steps

In this exercise, you will be making extensive use of `gdb` on the server program to understand the address-space layout and what the exploit code should do. Refer the online documentation for GDB [4] as well as cheat sheets (e.g., [5]). In particular, we will be using the commands `break` (to set a breakpoint), `run` (to run the program), `next` (to execute next line of code), `continue` (to continue the program from the current line), `print` (to print values of variables/memory), `x` (to dereference and print values of variables/memory), and `disas` (to disassemble code).

At a high-level, the steps in this exercise involve: (1) following the function pointer into the PLT and GOT tables (by using the format string vulnerability to read arbitrary memory), and (2) overwriting the GOT table's entry for the function pointer to the `system()` libc function (by using the format string vulnerability to write arbitrary memory).

5.1 Finding the offset of the function pointer

The first step is to find the value of the function pointer `printf_p` on the stack. For this, we will use the format string vulnerability to print out the values on the stack. Finding out the format string vulnerability and how to print values on the stack is part of the exercise (see 13.7.1 of reference [3]). To confirm, we run the program in `gdb` as follows, and print out the stack. The values from `gdb` should match what we get from the format string vulnerability.

```
user@machine$ gdb server
GNU gdb (GDB) 7.5.91.20130417-cvs-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/hayawardh/misc/ass2/server...done.
(gdb) break main
Breakpoint 1 at 0x8048920: file server.c, line 49.
(gdb) run
Starting program: /home/hayawardh/misc/ass2/server

Breakpoint 1, main () at server.c:49
49      {
(gdb) next
52              int (*printf_p) (const char *, ...) = &printf;
(gdb) next
53              char pre[4] = "srv";
(gdb) next
55              memset(&hints, 0, sizeof hints);
(gdb) print printf_p
$1 = (int (*)(const char *, ...)) 0x8048650 <printf@plt>
(gdb) x/20x $esp
0xbfffed10:     0x00000004      0x6474e551      0x00000000      0x00000000
0xbfffed20:     0x00000000      0x00000000      0x00000000      0xb7fe6fad
0xbfffed30:     0xb7e27f59      0x08048650      0x00767273      0xb7fff000
0xbfffed40:     0xbfffefa8      0x00000000      0xbfffefd8      0x0d696910
0xbfffed50:     0xb7e27f41      0xb7fdcac0      0x00000000      0xb7fff000
(gdb)
```

5.2 Dereference the function pointer's location in the PLT

The next step is to dereference the function pointer (which is a pointer to procedure linkage table (PLT) code), to get the address into the GOT table. Refer [1] for details of how the PLT and GOT work to enable position-independent code.

In our case, the function pointer points to the address `0x8048650`. Let us see the code at that address:

```
(gdb) disas 0x8048650
Dump of assembler code for function printf@plt:
   0x08048650 <+0>:     jmp    *0x804a010
   0x08048656 <+6>:     push   $0x8
   0x0804865b <+11>:    jmp    0x8048630
End of assembler dump.
```

As we can see, the code simply jumps into the global offset entry at `0x804a010`. To extract this address, we have to get the argument for the indirect jump instruction. The opcode for that instruction takes two bytes, so we need the contents of `0x8048650`:

```
(gdb) x/x 0x8048652
0x8048652 <printf@plt+2>:       0x0804a010
```

Refer section 13.7.2 of [3] for how to dereference using the format string vulnerability.

5.3 Dereference the GOT entry to get the function's address in libc

If we dereference the GOT entry, we obtain the function's entry in libc. However, the entry is initially unpatched, so we wait until the function pointer is called first (line 108). When this happens, the dynamic linker will patch the GOT entry to be the actual function's address in libc.

```
(gdb) n
56              hints.ai_family = AF_UNSPEC;
(gdb) n
57              hints.ai_socktype = SOCK_STREAM;
(gdb) n
58              hints.ai_flags = AI_PASSIVE; // use my IP
(gdb) n
60              if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
(gdb) n
66              for(p = servinfo; p != NULL; p = p->ai_next) {
(gdb) n
68                                      p->ai_protocol)) == -1) {
(gdb) n
67                      if ((sockfd = socket(p->ai_family, p->ai_socktype,
(gdb) n
73                      if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
(gdb) n
79                      if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
(gdb) n
85                      break;
(gdb) n
88              if (p == NULL)  {
(gdb) n
93              freeaddrinfo(servinfo); // all done with this structure
(gdb) n
95              if (listen(sockfd, BACKLOG) == -1) {
(gdb) n
100             sa.sa_handler = sigchld_handler; // reap all dead processes
(gdb) n
101             sigemptyset(&sa.sa_mask);
(gdb) n
102             sa.sa_flags = SA_RESTART;
(gdb) n
103             if (sigaction(SIGCHLD, &sa, NULL) == -1) {
(gdb) n
108             printf_p("%s: waiting for connections...\n", pre);
(gdb) n
srv: waiting for connections...
111                     sin_size = sizeof their_addr;
(gdb) x/x 0x804a010
0x804a010 <printf@got.plt>:     0xb7e637f0
```

Thus, we see that the printf function's address is `0xb7e637f0`. We have now de-randomized libc using the information leaked from the function pointer!

5.4 Get the absolute address of `system()`

We will now have to overwrite the GOT entry for `printf()` with `system()`, so that the next time `printf()` is called in the program, `system()` will be called instead. For this, we first need the absolute address of `system()`. Now that we know the absolute address of `printf()` in the current program execution, we can use that information to calculate the absolute address of `system()` by simply adding the offset between `system()` and `printf()` in the library.

Let us first find out what library file is loaded for libc:

```
$ cat /proc/‘pidof server‘/maps
08048000-08049000 r-xp 00000000 08:01 5381080    /home/hayawardh/misc/ass2/server
```

```
08049000-0804a000 r--p 00001000 08:01 5381080    /home/hayawardh/misc/ass2/server
0804a000-0804b000 rw-p 00002000 08:01 5381080    /home/hayawardh/misc/ass2/server
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e14000-b7e15000 rw-p 00000000 00:00 0
b7e15000-b7fc2000 r-xp 00000000 08:01 5505861    /lib/i386-linux-gnu/libc-2.17.so
b7fc2000-b7fc4000 r--p 001ad000 08:01 5505861    /lib/i386-linux-gnu/libc-2.17.so
b7fc4000-b7fc5000 rw-p 001af000 08:01 5505861    /lib/i386-linux-gnu/libc-2.17.so
b7fc5000-b7fc8000 rw-p 00000000 00:00 0
b7fda000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 5505837    /lib/i386-linux-gnu/ld-2.17.so
b7ffe000-b7fff000 r--p 0001f000 08:01 5505837    /lib/i386-linux-gnu/ld-2.17.so
b7fff000-b8000000 rw-p 00020000 08:01 5505837    /lib/i386-linux-gnu/ld-2.17.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
```

Let us examine the file to find the offsets of `system()` and `printf()`:

```
$ objdump -T /lib/i386-linux-gnu/libc-2.17.so  | grep " printf$"
0004e7f0 g    DF .text  00000034  GLIBC_2.0   printf
hayawardh@mantra:~/misc/ass2$ objdump -T /lib/i386-linux-gnu/libc-2.17.so  | grep " system$"
00041280  w   DF .text  0000008d  GLIBC_2.0   system
```

As we can see, `printf()` is loaded at an offset of `0x4e7f0`, and `system()` at an offset of `0x41280`. Use this to calculate the absolute address of `system()`. You can assume this offset as a constant in your exploiting program.

5.5 Hijack `printf()` GOT entry

After having gleaned information from the program about the absolute address of `system()`, we will overwrite the GOT entry (at address `0x804a010` in our example) with the address of `system()`. Details of how to do this using the format string vulnerability are in reference [7]. Once this is done, the next time `printf()` is called (line 136), `system()` will execute instead. To simplify the exercise, the arguments for `printf()` are already `/bin/sh`, which will simply launch a shell with `system()`. In real exploitation, the adversary will setup the arguments to `system()` on the stack herself, and then launch a reverse shell or bind a shell to a port using a command such as `netcat`. We are done!

6 Deliverables

You can use your client machines in assignment 1 to do the exercise. You will be provided with the server program's code, and will be required to write a client that will interact with the server and exploit the format string vulnerability as above, to launch a shell on the server. Your exploit code should work even if the server program is compiled with stack canaries and the system has ASLR enabled. You can use *any* language of your choice for the client.

Submit your well-commented client code as a tarred and gzipped archive file with the following filename: `yourfirstname_yourlastname_cse543_assn2.tar.gz`. The archive should include a `Makefile` and any other files required to run the code.

In addition to the code, include in the archive a file named `ANSWERS` with an answers for the question below.

1. Format String Vulnerability. Which line in the server code has the format string vulnerability?

2. Attack. Suppose the linked `libc` did not have `system()` or any other useful function that could be used directly. Can the program still be exploited? If so, how?

3. Defense. Find out which `gcc` flags prevent GOT hijacking. Briefly explain what the flags are, and what they do. Verify that the server program cannot be exploited using GOT hijacking if compiled with these flags.

7 Evaluation

- Client code that can generate format strings to read the stack (step 1) **- 5 points**
- Client code that can generate format strings to derference an address (steps 2, 3) **- 10 points**
- Finding the offset of `system()` from `printf()` in the library (step 4) **- 3 points**
- Client code that can generate format strings to write an address (step 5) **- 20 points**
- Complete working client code that exploits the server (steps 1-5) **- 2 points**

- Answer to question 1 **- 2 points**
- Answer to question 2 **- 3 points**
- Answer to question 3 **- 5 points**
- Total **- 50 points**

Hayawardh Vijayakumar

References

[1] Eli Bendersky. Position independent code (pic) in shared libraries. `http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/`, 2011.

[2] c0ntex. How to hijack the global offset table with pointers for root shells. `http://www.open-security.org/texts/6`, 2012.

[3] eTutorials.org. Format string bugs. `http://etutorials.org/Networking/network+security+assessment/Chapter+13.+Application-Level+Risks/13.7+Format+String+Bugs/`, 2013.

[4] GNU. Debugging with gdb. `http://sourceware.org/gdb/current/onlinedocs/gdb/`, 2013.

[5] Greg Ippolito. Gnu gdb debugger command cheat sheet. `http://www.yolinux.com/TUTORIALS/GDB-Commands.html`, 2013.

[6] F.J. Serna. The info leak era of software exploitation. `http://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf`, 2012.

[7] sloth@nopninjas.com. Format string technique. `http://julianor.tripod.com/bc/NN-formats.txt`, 2011.