Homework 6

CIS 4930 / CIS 5930
Offensive Computer Security
Spring 2014

Due March 24th, 2014, by *MIDNIGHT*
Worth: 100 points

Electronic turn in (Turn in via email to: redwood@cs.fsu.edu)
The email must be titled in the following format:
[OCS2014] hw6 <your last name>
(where <your last name> is your last name)
i.e.: [OCS2014] hw6 redwood

This homework is entirely focused on 32-bit (intel architecture) exploitation development. ALL problems have been statically compiled in a Virtual Machine running the liveCD .iso file provided with the Hacking: The Art of Exploitation textbook for this course. Thus you should develop all your exploits in that environment, or a similar one (a 32bit debian environment with ASLR disabled). Exploits developed outside of that environment are not guaranteed to work, as different linux distros perhaps having ASLR / N^X / GCC-stack-cookie mitigations can interfere with your work.

Go here to download all the files required for this HW: http://www.cs.fsu.edu/~redwood/OffensiveComputerSecurity/hw6_files.zip

1) [20 points (5 each)]

a) return to lib c exploits require shellcode (True / False)

b) In the case of a non executable stack, an attacker can still have an ***injected*** `jmp` or `call style` instruction on the stack point to his/her shellcode on the heap (or elsewhere). In other words, a `jmp/call` instruction that has been placed on the stack as part of an overflow attack, and somehow EIP has been directed to point to it. (True / False).

c) In linux systems with ASLR enabled, the address space for a process is randomized each time the process gets input (True / False).

d) Even when the stack is not executable, an attacker can still use shellcode placed in environment variables (True / False)

2) [15 points]

You are required to exploit the following code:

You can see that in the main function, there is no call to the win() function. Your goal is to find the input to give this function to exploit its vulnerability, and hijack the control flow to point to target().

stack1.c:

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

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[123];

  gets(buffer);
}
```

<u>Your answer is expected to be in the form:</u>

```
perl -e ‘print “your exploit here”’ | ./stack1
python -c ‘print “your exploit here”’ | ./stack1

(or some variant).

Also it is OK if your answer causes a segfault
```

Here are some hints to get you started:

- You'll need to find the address of target() in memory, and this can be done with objdump like this:
  - `objdump -d stack1 | grep win`
- Main has a return address stored on the stack, just like any other function. It will be near $ebp
- GDB will come in handy to debug this code, to develop your exploit. Here are some sample commands that will surely get you started if you are new to GDB:
  - To debug this binary with GDB, use the following command:
    - `gdb -q stack1`
  - Once in gdb, the command prompt appears as:
    - `(gdb)`
  - To set a breakpoint on the main function:

        - `(gdb) break main`
    - To see the assembly code at any address:
        - (gdb) disassemble main
            - will disassemble the code that the `main` symbol points to
        - `(gdb) disassemble *0xbffffff7`
            - will disassemble any instructions (if applicable) at the address `0xbffffff7`
    - To examine the processor's registers at any point:
        - `info registers`
    - Some examples to examine the stack:
        - `(gdb) info frame`
            - this will detail information about the current stack frame
        - `(gdb) x/30x $esp`
            - this means e<u>x</u>amine the contents of thirty values in memory starting at wherever $esp points at (The top stack pointer). Each value is 32-bits on 32-bit architecture.

Your requirement for this problem is to submit the program input that correctly hijacks control flow to point to the target() function to print the success command. Make sure you test it!

3) [25 points]
You are required to exploit the following code to login without providing a valid password:
heap_logon.c:

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

struct auth {
  char name[32];
  int auth;
};

struct auth *auth;
char *service;

int main(int argc, char **argv)
{
  char line[64];

  while(1) {
    printf("[ auth = %p, service = %p ]\n", auth, service);

    if(fgets(line, sizeof(line), stdin) == NULL) break;

    if(strncmp(line, "auth ", 5) == 0) {
      auth = malloc(sizeof(auth));
      memset(auth, 0, sizeof(auth));
      if(strlen(line + 5) < 31) {
        strcpy(auth->name, line + 5);
      }
    }
    if(strncmp(line, "reset", 5) == 0) {
      free(auth);
    }
    if(strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if(strncmp(line, "login", 5) == 0) {
      if(auth->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
      }
    }
    }
  }
```

Your goal is to craft an input that exploits the heap buffer overflow vulnerability break the authentication logic of this code. Use the techniques presented in the book and the lectures. There is certainly more than one right way to direct the program to the winning conditions. Your answer must be in the format of a single command line. For example:

```
./heap_logon “your string here”
or
echo $(python -c ‘print “XYZ………”) | ./heap_logon
```

4) [25 points] format string exploit
You are required to exploit the following code:
format.c:

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

int dummy = 1234;
int wrong_target;
int target;

void printbuffer(char *string)
{
  printf(string);
}

void vuln()
{
  char buffer[512];

  fgets(buffer, sizeof(buffer), stdin);

  printbuffer(buffer);
  wrongtarget = 1234;

  if(target == 0x01023931) {
    printf("you have modified the target :)  You Win!\n");
  } else {
    printf("target is %08x :(\n", target);
  }
}

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

Your goal is to craft an input that exploits the format string vulnerability to write a specific value to a global variable. Format string exploitation is a technique covered in the book (HAOE pages 167-179). Use the techniques presented in the book and the lectures. There is certainly more than one right way to direct the program to the "you have modified the target :)" printf win statement. <u>Your answer must be in the format of a single command line. For example:</u>

```
./format "your string here"
or
echo $(python -c 'print "XYZ………") | ./format
```

5) [15 points] anti-reverse engineering N-day binary exploitation
In order to compile a C/C++ program on a linux system to be conveniently debugged by GDB, one must use the `gcc -ggdb` flag. It has been theorized for a while now that an attacker might look into the extra stuff that gcc adds into the binary from compiling with the `-ggdb` flag to perhaps find some weaknesses in debuggers such as GDB and IDA. A recent exploit has been released that does just this: http://blog.ioactive.com/2012/12/striking-back-gdb-and-ida-debuggers.html

This exploit was used in last year's Nullcon CTF in their RE500 problem. The above URL provides the exploit code one can use to corrupt the debugging sections that are added to a binary by the GCC compiler. This particular exploit causes GDB and IDA both to seg-fault upon loading the binary, and still the bug that this exploit is leveraging has not been fixed by the GDB or IDA teams. Thus this GDB/IDA exploit is effectively a unpatched n-day (by a small stretch of the definition). Incident responders have to respond to malware equipped with techniques like this, that are specifically designed to make their life difficult. Thus it is important for you to know how to mitigate such anti-reverse engineering techniques... and the best way to start is on a recent 0day!

I was able to mitigate the zero-day on that RE500 challenge problem through the use of binary patching. Binary patching is the technique of modifying a binary (with a hexeditor) to zero out, or NOP out undesirable data or instructions. From reading the above blog post description of the exploit code, it modifies a number of sections in the binary that begin with ".debug". From that information alone you should be able to proceed and do the same for this problem. The file for this problem is called: **`antire`**

It has been modified with the aforementioned 0day technique, (and some minor other techniques) to prevent debugging. Also this time you get no source code. Your goal is to binary patch it so that you can debug it, and then reverse engineer and exploit the binary to find the key inside. The source code in this case contains a number of vulnerabilities (format string, buffer overflow, etc) that you will need to exploit in order to get the key. Good luck!

> **a) [15 points]** Find the correct section headers to binary patch out, provide me a patched version of the binary for me to verify your work (if it segfaults upon loading in gdb, you get zero points)
> **c) [EXTRA CREDIT: 5 points]** What is the address of levelthree()?
>
> **c) [EXTRA CREDIT: 20 points]** What is the key? How did you find it (provide exploit code and brief explanation/writeup)

----

6) [EXTRA CREDIT 5 points]
Given what we've covered so far about x86 reverse engineering, stack/heap/format exploitation, writing shellcode (for linux), return to libc and return to library style attacks, and exploit mitigation security (ASLR/ stack cookies/ etc....), list anything we've covered that you feel you don't understand. Also list anything that is related, that might not have been covered that you would like to learn about.