Paxson
Spring 2011

CS 161
Computer Security

Homework 1

Due: Wednesday, February 9, at 9:59pm

Instructions. Submit your solution by Wednesday, February 9, at 9:59pm, in the drop box labelled CS161 in 283 Soda Hall. Print your name, your class account name (e.g., `cs161-xy`), your TA's name, the discussion section time where you want to pick up your graded homework, and "HW1" prominently on the first page. Staple all pages together. Your solutions must be legible and the solution to each problem must be labelled clearly. You must work on your own on this homework.

Problem 1 ***Memory safety*** **(20 points)**

Alice has decided to write her diary in digital form. To make sure that the secrets of her life stay safe, she wants to encrypt the diary. She downloads from the web a command-line utility called `encryptor` for encrypting text. `encryptor` takes two arguments: a *key* and a *filename* to store the encrypted text. It reads the text to encrypt from the standard input and writes its encryption using *key* to the given file.

Alice decides it would be a good idea to store each day's diary in a separate file with a separate key. In the case of compromise of one key, her diary for other days will remain secure. In order to simplify the task, she has written the following code:

```
struct date
{
   int day;
   char* month;
   int year;
};

void write_diary(char* text, struct date today)
{
   FILE* diary;
   char buf[200];
   int key = today.day + today.year * 365;

   sprintf(buf, "encryptor -k %d -f \"mydiary_%d-%s-%d.txt\"",
       key, today.day, today.month, today.year);

   diary = popen(buf, "w");
   if (! diary)
       /* something about the command failed, give up */
       return;
```

```
   fprintf(diary, text);
   pclose(diary);
}
```

Unfortunately, Alice developed the code in a rush, and did not write secure or robust code. One problem concerns her computation of the encryption key: it is both easy to guess, and will sometimes repeat. *Ignoring these encryption issues*, identify at least 3 security problems with her code. For each problem, describe an example of input that an attacker could provide (in terms of the arguments in a call to `write_diary`) that would cause the security problem to occur.

HINT: Familiarize yourself with the workings of `popen()` and `pclose()` if they are new to you. You can read the manual pages for `popen()` by typing `man popen` at a shell prompt on a Unix system.

Problem 2 ***Frame Pointer Overwrite*** **(20 points)**

The C code below has an off-by-one error; the loop in the `vuln()` function iterates one more time than it should.

```
void vuln(char* s)
{
    char buffer[256];
    int i;
    int n = strlen(s);

    if (n > 256)
      n = 256;
    for (i = 0; i <= n; i++)
      buffer[i] = s[i];
}

int main(int argc, char* argv[])
{
    if (argc < 2)
    {
       printf("missing args\n");
       exit(-1);
    }

    vuln(argv[1]);
}
```

In Section #0 we discussed the layout of stack, including different types of information that is stored on the stack during function calls. Different implementations can vary in the particulars of the stack layout, but for this problem assume a layout that corresponds to the specific example given in Section.

(a) You will likely find it helpful to sketch the stack for this program. (You do *not* have to include the sketch in your writeup.) Can the attacker overwrite the saved frame pointer (`SFP` in the Section notes)? Can the attacker overwrite the return instruction pointer (`RIP`)? Explain why for each.

(b) Explain how the attacker can exploit the opportunity to overwrite a single byte to modify the program's flow of execution.

HINT: Pay close attention to how returning from a function works; popping a return address from the stack has a dependency on `SFP`. You may find it helpful to read the discussion in the Section materials about the modification of registers by the `leave` and `ret` instructions of the `x86` instruction set.

Problem 3 ***Heap Overflow*** **(20 points)**

Stack smashing attacks generally work by modifying a program's control flow because information regarding control flow is stored in the same way as data. Similarly, heap overflow vulnerabilities arise because attackers can cause data they supply to be interpreted as control flow information.

The slides from Section #0 discuss general approaches to implementing heap-based storage. Review the chunk structure in those materials to understand how it differs for allocated chunks versus free chunks.

When using heap memory, a program releases a buffer by calling the `free` function. `free` adjusts the pointer passed to it to point to the beginning of the chunk and checks whether the surrounding chunks are allocated. If they are not, it merges the chunk being freed with the already free ones into a bigger chunk.[1] The merge process involves removing the free chunks from their bin, then consolidating the chunks, and finally placing the single new chunk into a bin according to its size. In this problem, we focus on a heap overflow that can be triggered during the removal of a chunk from its bin, which is implemented by the `unlink` macro:

```
/* P: Chunk being unlinked
 * BK: Previous chunk
 * FD: Next chunk
```

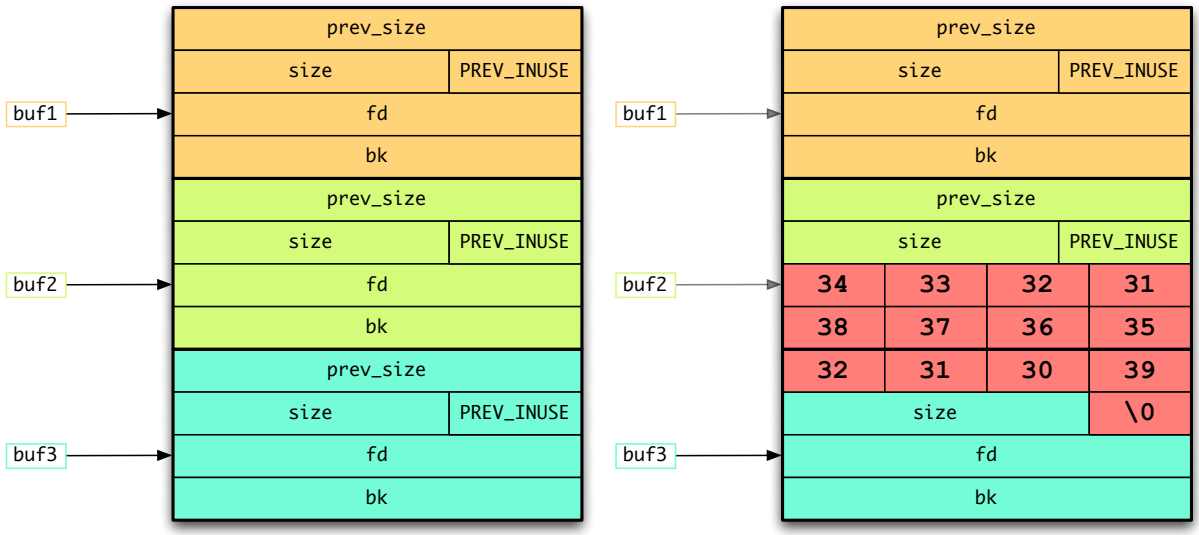
[1] There are other possibilities for how a system might implement `malloc` and `free`. Here we focus on a concrete implementation approach, namely the one presented in the Section materials.

```
 */
#define unlink(P, BK, FD)
{
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;  /* equivalent to *(P->fd + 12) = P->bk */
    BK->fd = FD;  /* equivalent to *(P->bk + 8) = P->fd */
}
```

Consider the following code example along with the corresponding heap layout (where we allocate zero-length buffers to keep the accompanying diagrams a bit simpler):

```
char* buf1 = malloc(0); /* empty just to keep the diagrams simple */
char* buf2 = malloc(0);
char* buf3 = malloc(0);
...
gets(buf2);
...
free(buf1);
free(buf2);
```

In the code, we allocate three zero-sized buffers. The code then copies user input from standard input into `buf2`, and finally frees the first two buffers. In the following, the left figure shows the heap layout before reading the input, and the right figure after having read the string `"123456789012"`:



(a) Describe what happens when `free(buf1)` is called, and in particular how this results in corrupting the heap memory.

(b) How can an attacker exploit this vulnerability to inject code? For your analysis, assume that the second line of the code instead reads:

```
char* buf2 = malloc(256);
```

so that it has enough space to hold the injected code.[2]

Problem 4 ***Security Principles*** **(20 points)**

Identify one or more security principles relevant to each of the following scenarios, giving a one or two sentence justification for your answer:

(a) You are given the task to code a mail server that will run on the standard SMTP port 25 on a UNIX system. On UNIX systems, a program must have "root" (super-user) privileges to run a service on a port less than 1024.

(b) The Windows API enables programmers to control access to process objects by specifying a security descriptor when calling the `CreateProcess` function. The security descriptor is a detailed structure defining who can perform what actions. In the API, passing a value of NULL means that the process uses the default security descriptor, which inherits the properties of the creator of the process. Many programmers indeed use NULL, given its convenience.

(c) BankOBits is a local bank that offers its customers access to a number of conveniently located ATMs. Normally, when a customer inserts their ATM card into a BankOBits ATM, the ATM will contact the BankOBits central server to validate the ATM card inserted into it and check that the corresponding account has sufficient funds before issuing money. If the server does not respond or the network connection is down, the BankOBits ATM allows the customer to withdraw up to $300, keeps a record of the transaction, and uploads that information to the BankOBits server when connectivity comes back. As a result of this design decision, a gang of criminals steals from the bank by cutting the network connection on BankOBits ATMs and repeatedly withdrawing $300 from them even though the account does not have that much money in it.

(d) A kiosk at the SFO airport lets you access the web for a fee. To use the kiosk, you enter your credit card information at a welcome screen before the kiosk will give you access to a web browser. However, an observant hacker discovered that if you press F1 to invoke the "help" screen, the Windows help subsystem pops up a window with generic help information about the login screen. The help text happens to

[2] This extra space would exist between the `bk` field of chunk #2 and the `prev_size` field of chunk #3.

contain a link to an external web site with more help information, and if you click on *that* link, the kiosk opens the Internet Explorer web browser to display contents of the link. At that point, you can change the URL in the Internet Explorer address bar to gain full access to the web, all without paying.

Problem 5 ***Reasoning About Code*** **(20 points)**

Consider the following C code:

```
void dectobin(unsigned int decimal, char* binary)
{
   char temp[20];
   int j = 0;
   int k = 0;

   while (decimal > 0)
   {
       temp[j++] = (decimal % 2) + '0';
       decimal = decimal / 2;
   }

   while (j >= 0)
       binary[k++] = temp[--j];

   binary[k - 1] = '\0';
}
```

Is the above code is memory safe? If yes, prove it by writing the precondition and invariants. If not, describe the modifications required and prove that the modified code is memory safe.

Problem 6 ***Feedback*** **(0 points)**

The feedback we received from Homework #0 was highly helpful, and further feedback would be great to have. So, optionally, feel free to include comments about the course, such as *What's the single thing we could to make the class better?*, or *What did you find most difficult or confusing from lectures or the rest of class, and would like to see explained better?*

If you have feedback, create a text file called `q6.txt` with your comments.