Princeton University
COS 217: Introduction to Programming Systems

Assignment 5: Buffer Overrun

Purpose

The purpose of this assignment is to help you learn (1) how programs are represented in ARMv8 machine language, (2) how ARMv8 stack frames are structured in memory, and (3) how ARMv8 programs can be vulnerable to buffer overrun attacks.

As noted in the next section, this is a teams-of-two assignment.

Rules

You may work with one teammate on this assignment. You need not work with a teammate, but we prefer that you do. Your preceptor will help you with your teammate search, if you so desire.

Your teammate must be from your precept. So make sure you find a teammate as soon as you can. After all, if your precept has an odd number of students, then someone must work alone; don't let that person be you! Any exceptions to this policy must be made by the lead instructor.

The "A+" attack (as defined below) is the challenge part of this assignment. While doing the challenge part of the assignment, you are bound to observe the course policies regarding assignment conduct as given in the course Policies web page, plus one additional policy: you may not use any "human" sources of information. That is, you may not consult with the course's staff members, the lab teaching assistants, other current students via Piazza, or any other people while working on the challenge part of an assignment, except for clarification of requirements. However you may consult with your teammate, with no limitations.

As noted below, the challenge part is worth 10 percent of this assignment. So if you don't do the challenge part and all other parts of your assignment solution are perfect and submitted on time, then your grade for the assignment will be 90 percent.

Background

We will provide a "grader" program, both source code (`grader.c`) and executable binary code (`grader`). The file `grader` was produced from `grader.c` using this command:

```
gcc217 -O -fomit-frame-pointer grader.c -o grader
```

In short, the `-O` and `-fomit-frame-pointer` options make sure that the grader program is vulnerable to buffer overflow attacks. See the course instructors if you would like more information about those options.

The program asks you for your name, and writes something like this (where the `user input` and *`program output`* are indicated by font style):

```
$ ./grader
What is your name?
Bob
D is your grade.
Thank you, Bob.
```

However, the author of the program inexplicably forgot to do bounds-checking on the array into which the program reads the input, and so the program is vulnerable to a *buffer overrun* (alias *buffer overflow*) attack.

Your Task

Your task is to attack the given program by exploiting its buffer overrun vulnerability. More specifically, your job is to provide input data to the program so that it writes something more like this:

```
$ ./grader < dataB
What is your name?
B is your grade.
Thank you, Bob.
```

As you can see from reading the program, it is designed to give the B grade if the user's name is Andrew Appel, but not if the user's name is Bob. However, it is programmed sloppily: it reads the input into a buffer, but forgets to check whether the input fits. This means that a too-long input can overwrite other important memory, and you can trick the program into giving you a B even though your name is not Andrew Appel.

Here's another example

```
$ ./grader < dataA
What is your name?
A is your grade.
Thank you, Bob.
```

As you can see from reading the program, it is designed not to give anyone an A under any circumstances. However, again, it is programmed sloppily. A too-long input can overwrite other important memory, and you can trick the program into giving you an A.

In fact, because of the program's buffer overrun vulnerability, you even can trick the program into giving you an A+:

```
$ ./grader < dataAplus
What is your name?
A+ is your grade.
Thank you, Bob.
```

The Procedure

Develop on armlab. Use `emacs` to create source code. Use `gdb` to debug.

The armlab `/u/cos217/Assignment5` directory contains files that you will need. Subsequent parts of this document describe them. Create a project directory, and copy all files from the `/u/cos217/Assignment5` directory to your project directory. Then complete the parts of the assignment given below, in the order of their appearance.

> Study the given `Makefile`. Using it could save you lots of typing.

The precept document entitled *A Linux File Sharing Trick* describes a mechanism that you can use to share Assignment 5 files with your teammate.

Step 1: Legal Matters (2% of the grade)

Copy these sentences to your `readme` file, and fill in the blanks so the sentences are correct:

> According to 18 U.S. Code 1030, if you were to use a buffer overrun attack to commit fraud or related activity in connection with computers, but did not attempt to cause death and did not knowingly or recklessly cause death, then you could receive a maximum penalty of _____ in prison.
>
> According to 18 U.S. Code 1030, if you were to use a buffer overrun attack to commit fraud or related activity in connection with computers, and attempted to cause death or knowingly or recklessly caused death, then you could receive a maximum penalty of _____ in prison.

It's fine to do a web search to complete Step 1 of the assignment.

Step 2: The Memory Map (37% of the grade)

Create a text file named `memorymap`. Begin your `memorymap` file with your name and your teammate's name.

Take the `grader` executable binary file that we have provided you, and use `gdb` to analyze its sections.

Step 2.1: The Memory Map of the Text Section

Analyze the **text** section by issuing this `x` command:

```
$ gdb grader
(gdb) x/71i readString
```

Then copy the resulting 71 lines of assembly language code into your `memorymap` file. (Be careful: `gdb` displays the lines one windowfull at a time, so you must press the <Enter> key to see all 71 lines.) Then annotate the lines to explain them.

Do not annotate every line of assembly language code. Instead, cluster the lines of assembly language code into "paragraphs," and annotate each paragraph. Your analysis must have this format:

```
Annotation
Line of assembly language code
Line of assembly language code
...
<blank line>
Annotation
Line of assembly language code
Line of assembly language code
...
<blank line>
...
```

Use these 7 annotations (and only these 7 annotations) in the `readString` function:

```
Prolog
First loop setup
First loop
buf[i] = '\0'
Second loop setup
Second loop
Epilog and return
```

Use these 4 annotations (and only these 4 annotations) in the `getName` function:

```
Prolog
printf("What is your name?\n");
readString();
Epilog and return
```

Use these 8 annotations (and only these 8 annotations) in the `main` function:

```
Prolog
mprotect(...);
getName();
if (strcmp(name, "Andrew Appel") != 0) skip assignment to grade
grade = 'B';
printf("%c is your grade.\n", grade);
printf("Thank you, %s.\n", name);
Epilog and return 0
```

Step 2.2: The Memory Map of the Data Section

-

Analyze the **data** section by issuing these `gdb` commands:

```
$ gdb grader
(gdb) break main
(gdb) run
(gdb) print &grade
(gdb) x/x &grade
```

Place a table in your `memorymap` file showing the layout of the data section. The table must have three columns: *Address (in hex)*, *Content (in hex)*, and *Description*. The table must contain one row for each byte in the data section. Since the data section contains exactly one byte, the table must contain exactly one row.

Step 2.3: The Memory Map of the BSS Section

- Analyze the **bss** section by issuing this `gdb` command:

```
$ gdb grader
(gdb) print &name
```

Place a table in your `memorymap` file showing the layout of the bss section. The table must have two columns: *Address (in hex)* and *Description*. The table must contain one row for each byte in the bss section, that is, one row for each byte of the `name` array.

At the start of program execution, the content of the `name` array will be zeros. Later during program execution, the `name` array will contain more interesting data.

Compose your memory map of the bss section before you implement your "A" attack (as described below). The table in your memory map must describe the content of the `name` array **as you wish it to be** during your "A" attack. Thus your memory map of the bss section will help you to compose your "A" attack.

For your sake, it's fine to add another column to your memory map describing the content of the `name` array as you wish it to be during your "A+" attack. Thus your memory map of the bss section will help you to compose your "A+" attack. But you're not required to add that column.

Step 2.4: The Memory Map of the Stack Section

Using your analysis of the text section, compose an analysis of the **stack** section. Place a table in your `memorymap` file showing the layout of the stack. The table must have two columns: *Offset* and *Description*. Each row must represent 8 bytes. Each offset must be expressed as a positive offset relative to the SP register. The first row must have offset 0, the second row must have offset 8, the third row must have offset 16, and so forth. The table must show the content of the stack from the top value through the value of the `X30` register that was pushed onto the stack by the `getName` function.

You'll discover that the stack begins with the content of some registers; each row must have a description which is the name of the register whose content is stored at that spot in the stack. Then the stack contains the `buf` array; each row that comprises the `buf` array must have the description "buf". Finally the stack contains the content of the `X30` register that was stored by the `getName` function; that row must have the description "X30".

Step 3: The "B" Attack (12% of the grade)

Compose a C program named `createdataB.c` that produces a file named `dataB`, as short and simple as possible, that causes the `grader` program to write your name and recommend a grade of "B". You can see by reading the program that, if your name is Andrew Appel, that's very easy to do. But that's not easy to do if your name isn't Andrew Appel! To receive full credit the `dataB` file must cause the `grader` program to generate output whose format is indistinguishable from normal output.

The `createdataB.c` program must write to the `dataB` file; it must not write to `stdout`.

Your `createdataB.c` file must contain these comments:

- A *file* comment, that is, a comment at the beginning of the file providing the file name, your name, and your teammate's name.
- A *function* comment describing the "I/O behavior" of the `main` function. That is, the function comment must describe what the `main` function accepts as command-line arguments, reads from `stdin` or any other stream, writes to `stdout`, `stderr`, or any other stream, and returns.
- *Local* comments, that is, comments throughout your program that thoroughly describe the bytes that the program writes to the `dataB` file.

> It is acceptable to use "magic numbers" throughout your `createdataB.c` file.

> For the "B" attack it's OK to truncate your name(s) if necessary.

> Suggestion: After creating your `dataB` file, issue the command `xxd dataB` to confirm that the file contains exactly the bytes that you want it to contain.

Step 4: A MiniAssembler Module (15% of the grade)

The given `miniassembler.h` interface file declares four functions that generate machine language instructions. Comments in the file describe the functions. The given `miniassembler.c` implementation file defines one of those functions. Define the other three, thus completing the `miniassembler.c` implementation file.

The given `testminiassembler.c` file tests the `MiniAssembler` module. Use the MiniAssembler module and `testminiassembler.c` to build a program named `miniassembler`. Run the program, and compare its output to the given `testminiassembler.out` file to make sure the function definitions in your MiniAssembler module are correct.

Step 5: The "A" Attack (24% of the grade)

Compose a C program named `createdataA.c` that produces a file named `dataA`, as short and simple as possible, that causes the `grader` program to write your name and recommend a grade of "A". To receive full credit the `dataA` file must cause the `grader` program to generate output whose format is indistinguishable from normal output. Your program must call the four functions that are declared in `miniassembler.h`.

The `createdataA.c` program must write to the `dataA` file; it must not write to `stdout`.

Your `createdataA.c` file must contain these comments:

- A *file* comment, that is, a comment at the beginning of the file providing the file name, your name, and your teammate's name.
- A *function* comment describing the "I/O behavior" of the `main` function. That is, the function comment must describe what the `main` function accepts as command-line arguments, reads from `stdin` or any other stream, writes to `stdout`, `stderr`, or any other stream, and returns.
- *Local* comments, that is, comments throughout your program that thoroughly describe the bytes that the program writes to the `dataA` file.

> It is acceptable to use "magic numbers" throughout your `createdataA.c` file.

> For the "A" attack it's OK to truncate your name(s) if necessary.

> Suggestion: After creating your `dataA` file, issue the command `xxd dataA` to confirm that the file contains exactly the bytes that you want it to contain.

Step 6: The "A+" Attack (10% of the grade)

Compose a C program named `createdataAplus.c` that produces a file named `dataAplus`, as short and simple as possible, that causes the `grader` program to write your name and recommend a grade of "A+". To receive credit the `dataAplus` file must cause the `grader` program to generate output whose format is indistinguishable from normal output.

The `createdataAplus.c` program must write to the `dataAplus` file; it must not write to `stdout`.

Your `createdataAplus.c` file must contain:

- A *file* comment, that is, a comment at the beginning of the file providing the file name, your name, and your teammate's name.
- A *function* comment describing the "I/O behavior" of the `main` function. That is, the function comment must describe what the `main` function accepts as command-line arguments, reads from `stdin` or any other stream, writes to `stdout`, `stderr`, or any other stream, and returns.
- *Local* comments, that is, comments throughout your program that thoroughly describe the bytes that the program writes to the `dataAplus` file.

> It is acceptable to use "magic numbers" throughout your `createdataAplus.c` file.

> For the "A+" attack it's OK to truncate your name(s) if necessary.

> For the "A+" attack it's OK to declare additional functions in the `miniassembler.h` file and define additional functions in the `miniassembler.c` file.

Finishing Up

Edit your copy of the given `readme` file by answering each question that is expressed therein.

One of the sections of the `readme` file requires you to list the authorized sources of information that you used to complete the assignment. Another section requires you to list the *unauthorized* sources of information that you used to complete the assignment. Your grader will not grade your submission unless you have completed those sections. To complete the "authorized sources" section of your `readme` file, copy the list of authorized sources given in the *Policies* web page to that section, and edit it as appropriate.

Provide the instructors with your feedback on the assignment. To do that, issue this command:

```
FeedbackCOS217.py 5
```

and answer the questions that it asks. (When answering the numeric questions, please enter the average of the responses that you and your teammate would provide individually.) That command stores its questions and your answers in a file named `feedback` in your working directory.

Submit your work electronically on armlab. If you worked with a teammate, then one of the teammates must submit all of your team's files, and the other teammate must submit none of your team's files. Your `readme` file, your `memorymap` file, and your source code files must contain both your name and your teammate's name. Use these commands to submit:

```
submit 5 createdataB.c
submit 5 miniassembler.h miniassembler.c
submit 5 createdataA.c
submit 5 createdataAplus.c
submit 5 memorymap readme feedback
```

Notes

On some versions of Linux, every time the program is executed the initial stack pointer is in a different place. This makes it difficult to make an attack in which the return address points into the same data that was just read into the buffer on the stack. (Indeed, that is the purpose of varying the initial stack pointer!) However, you will note that the grader program copies data from `buf` (which is in the stack section) into `name` (which is in the bss section). You'll find that `name` is reliably in the same place every time you (or we) run the grader program.

On some versions of Linux, executing instructions from the bss section causes a segmentation fault. The purpose of this is to defend against buffer overrun attacks! The `mprotect` call in the grader program disables that protection. You're not required to understand or explain how that line works. Note, however, that this mechanism (even if we didn't disable it) would not defend against the "B" attack.

If you work hard, you could create input that exploits the buffer overrun to take over the grader's Linux process and do all sorts of damage. DON'T DO THAT! Any deliberate attempt of that sort is a violation of the University's disciplinary code, and also is a violation of the Computer Fraud and Abuse Act (see Step 1 above).

Program Style

In part, good program style is defined by the `splint` and `critTer` tools, and by the rules given in *The Practice of Programming* (Kernighan and Pike) as summarized by the Rules of Programming Style document.

The more course-specific style rules listed in the previous assignment specifications also apply, as do the rules given in previous sections of this assignment specification.

Grading

Minimal requirement to receive credit for creatdataB.c: the `createdataB` program must build.

Minimal requirement to receive credit for miniassembler.c: the `testminiassembler` program must build.

Minimal requirement to receive credit for createdataA.c: the `createdataA` program must build.

Minimal requirement to receive credit for createdataAplus.c: the `createdataAplus` program must build.

When we grade this assignment, we will take the recommendation of the `grader` program into account. But that will not be the only criterion. In particular, see the grade percentages noted above.

To receive the full 10 percent credit for your "A+" attack, your "A+"attack must work perfectly. If it doesn't work perfectly, then we will award up to 5 percent partial credit, but only if your `createdataAplus.c` program contains a very thorough comment explaining the principles of operation of your attack.

We will grade your work on two kinds of quality:

- Quality from the *user's* point of view. From the user's point of view, your code has quality if it successfully implements the attacks described above.

- Quality from the *programmer's* point of view. From the programmer's point of view, your code has quality if it is well styled and thereby easy to maintain. Good program style is defined by the previous sections of this assignment specification.

To encourage good coding practices, we will deduct points if `gcc217` generates warning messages.

Debugging Tips

While debugging your attacks you might find it useful to use `gdb` to step through the execution of the `grader` program at the machine language level. These commands are appropriate for doing that:

- `display/i $pc`

  `gdb` maintains a *display list*. You can issue the `display` command to place items on the display list. Typically you place variables on the display list. At each pause in execution, `gdb` displays the values of all those variables. Thus the display list is a handy way to track the values of variables throughout the execution of your program.

  That particular `display` command tells `gdb` to place the pc (program counter) register on the display list. Thus at each pause in execution `gdb` displays the content of the memory to which pc points. That is, at each pause in execution `gdb` displays the instruction that is about to be executed.

  Moreover, that particular `display` command tells `gdb` to use the `i` (instruction) format when displaying. Thus at each pause in execution `gdb` interprets the content of the memory to which pc points as a machine language instruction, and displays that instruction in assembly language.

  In short, that command tells `gdb` to display the instruction that is about to be executed.

- `stepi`

  As you know, in `gdb` the `step` command (abbreviated `s`) executes the next line of C code. Since the `grader` executable binary file was built without the `-g` option, `gdb` has no knowledge of how the machine language instructions of the `grader` executable binary file correspond to lines of C code. So the `step` command is useless when analyzing the `grader` executable binary file.

  Instead you can use the lower-level `stepi` command. The `stepi` command (abbreviated `si`) tells `gdb` to execute the next machine language instruction.

This assignment was written by Andrew Appel and Robert M. Dondero, Jr.

grader.c (Page 1 of 1)

```
 1: #include <stdio.h>
 2: #include <string.h>
 3: #include <stdlib.h>
 4: #include <sys/mman.h>
 5:
 6: enum {BUFSIZE = 48};
 7:
 8: char grade = 'D';
 9: char name[BUFSIZE];
10:
11: void readString(void)
12: {
13:    char buf[BUFSIZE];
14:    int i = 0;
15:    int c;
16:
17:    for (;;)
18:    {
19:       c = fgetc(stdin);
20:       if ((c == EOF) || (c == '\n'))
21:          break;
22:       buf[i] = c;
23:       i++;
24:    }
25:    buf[i] = '\0';
26:
27:    for (i = 0; i < BUFSIZE; i++)
28:       name[i] = buf[i];
29: }
30:
31: void getName(void)
32: {
33:    printf("What is your name?\n");
34:    readString();
35: }
36:
37: int main(void)
38: {
39:    mprotect((void*)((unsigned long)name & 0xfffffffffffff000), 1,
40:             PROT_READ | PROT_WRITE | PROT_EXEC);
41:
42:    getName();
43:
44:    if (strcmp(name, "Andrew Appel") == 0)
45:       grade = 'B';
46:
47:    printf("%c is your grade.\n", grade);
48:    printf("Thank you, %s.\n", name);
49:
50:    return 0;
51: }
```

Princeton University
COS 217: Introduction to Programming Systems
Writing Binary Data

Example 1

We wish to five 0 bytes (alias null characters, alias '\0' characters) to a file named `data`. That is, we wish to write these five bytes to the file:

```
00000000 00000000 00000000 00000000 00000000
```

Open the File
```
FILE *psFile;
psFile = fopen("data", "w");
```

Attempt 1 (Incorrect)
```
fprintf(psFile, "00000");  /* Writes 00110000 00110000 00110000 00110000 00110000 */
```

Attempt 2 (Incorrect)
```
for (i = 0; i < 5; i++)
   fprintf(psFile, "%c", '0');  /* Writes 00110000*/
```

Attempt 3 (Incorrect)
```
for (i = 0; i < 5; i++)
   putc('0', psFile);  /* Writes 00110000 */
```

Attempt 4 (Correct)
```
for (i = 0; i < 5; i++)
   fprintf(psFile, "%c", '\0');  /* Writes 00000000*/
```

Attempt 5 (Correct)
```
for (i = 0; i < 5; i++)
   fprintf(psFile, "%c", 0);  /* Writes 00000000*/
```

Attempt 6 (Correct)
```
for (i = 0; i < 5; i++)
   fprintf(psFile, "%c", 0x00);  /* Writes 00000000 */
```

Attempt 7 (Correct)
```
for (i = 0; i < 5; i++)
   putc('\0', psFile);  /* Writes 00000000 */
```

Attempt 8 (Correct)
```
for (i = 0; i < 5; i++)
   putc(0, psFile);  /* Writes 00000000 */
```

Attempt 9 (Correct)
```
for (i = 0; i < 5; i++)
   putc(0x00, psFile);  /* Writes 00000000 */
```

Close the File
```
fclose(psFile);
```

Example 2

We wish to write the unsigned long `0x0123456789abcdef` to a file named `data` as it would appear in memory as an eight-byte entity. As humans, we would express the unsigned long `0x0123456789abcdef` in binary like this:

```
00000001 00100011 01000101 01100111 10001001 10101011 11001101 11101111
most sig                                                        least sig
byte                                                            byte
```

But ARMv8 is a little-endian architecture. In the memory of a little-endian computer, the least significant byte of an integer is in the lowest memory location. So the unsigned long `0x0123456789abcdef` appears in memory like this:

```
11101111 11001101 10101011 10001001 01100111 01000101 00100011 00000001
least sig                                                       most sig
byte                                                            byte
```

Or, more precisely, like this:

```
pretend
address
1000     11101111  least sig byte
1001     11001101
1002     10101011
1003     10001001
1004     01100111
1005     01000101
1006     00100011
1007     00000001  most sig byte
```

Open the File

```
FILE *psFile;
psFile = fopen("data", "w");
```

Attempt 1 (Incorrect)

```
fprintf(psFile, "0123456789abcdef");
/* Writes 00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111
          00111000 00111001 01100001 01100010 01100011 01100100 01100101 01100110 */
```

Attempt 2 (Incorrect)

```
fprintf(psFile, "%x", 0x0123456789abcdef);
/* Writes 00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111
          00111000 00111001 01100001 01100010 01100011 01100100 01100101 01100110 */
```

Attempt 3 (Correct)

```
fprintf(psFile, "%c", 0xef);  /* Writes 11101111 */
fprintf(psFile, "%c", 0xcd);  /* Writes 11001101 */
fprintf(psFile, "%c", 0xab);  /* Writes 10101011 */
fprintf(psFile, "%c", 0x89);  /* Writes 10001001 */
fprintf(psFile, "%c", 0x67);  /* Writes 01100111 */
fprintf(psFile, "%c", 0x45);  /* Writes 01000101 */
fprintf(psFile, "%c", 0x23);  /* Writes 00100011 */
fprintf(psFile, "%c", 0x01);  /* Writes 00000001 */
```

Attempt 4 (Correct)

```
putc(0xef, psFile);  /* Writes 11101111 */
putc(0xcd, psFile);  /* Writes 11001101 */
putc(0xab, psFile);  /* Writes 10101011 */
putc(0x89, psFile);  /* Writes 10001001 */
putc(0x67, psFile);  /* Writes 01100111 */
putc(0x45, psFile);  /* Writes 01000101 */
putc(0x23, psFile);  /* Writes 00100011 */
putc(0x01, psFile);  /* Writes 00000001 */
```

Attempt 5 (Correct) **<--- the easiest approach**

```
unsigned long ulData;
...
```

```
ulData = 0x0123456789abcdef;
fwrite(&ulData, sizeof(unsigned long), 1, psFile);
/* Writes 11101111 11001101 10101011 10001001 01100111 01000101 00100011 00000001 */

Close the File
fclose(psFile);
```

Example 3

We wish to write the unsigned int `0x01234567` to a file named `data` as it would appear in memory as a four-byte entity. As humans, we would express the unsigned int `0x01234567` in binary like this:

```
00000001 00100011 01000101 01100111
most sig                   least sig
byte                       byte
```

But ARMv8 is a little-endian architecture. In the memory of a little-endian computer, the least significant byte of an integer is in the lowest memory location. So the unsigned int `0x01234567` appears in memory like this:

```
01100111 01000101 00100011 00000001
least sig                  most sig
byte                       byte
```

Or, more precisely, like this:

```
pretend
address
1000     01100111  least sig byte
1001     01000101
1002     00100011
1003     00000001  most sig byte
```

```
Open the File
FILE *psFile;
psFile = fopen("data", "w");

Attempt 1 (Incorrect)
fprintf(psFile, "01234567");
/* Writes 00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111 */

Attempt 2 (Incorrect)
fprintf(psFile, "%x", 0x01234567);
/* Writes 00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111 */

Attempt 3 (Correct)
fprintf(psFile, "%c", 0x67);  /* Writes 01100111 */
fprintf(psFile, "%c", 0x45);  /* Writes 01000101 */
fprintf(psFile, "%c", 0x23);  /* Writes 00100011 */
fprintf(psFile, "%c", 0x01);  /* Writes 00000001 */

Attempt 4 (Correct)
putc(0x67, psFile);  /* Writes 01100111 */
putc(0x45, psFile);  /* Writes 01000101 */
putc(0x23, psFile);  /* Writes 00100011 */
putc(0x01, psFile);  /* Writes 00000001 */

Attempt 5 (Correct)              <--- the easiest approach
unsigned int uiData;
...
uiData = 0x01234567;
fwrite(&uiData, sizeof(unsigned int), 1, psFile);
/* Writes 01100111 01000101 00100011 00000001 */

Close the File
fclose(psFile);
```