CSE4501 – Vulnerability Research: Lab 2

Eric Pereira

Septemer 17th, 2019

Contents

Use the necessary tools to perform analysis of the compiled source code generated. Since we are working in 32-bit use "-m32" for your gcc compilation options. For compilation I would suggest using compiler option "`-fno-stack-protector`". Please submit write-ups of your analysis. Your points will be based on completeness of analysis, your understanding of the problems, and if you were able to achieve the goals. Each problem is worth 5 pts for a total of 35pts.

Problem 1

Compile the below program; open in disassembler and compare source code with assembled instructions. Examine the disassembly; can you identify the integer overflow? Can you cause the overflow to happen?

```c
#include <stdio.h>

void ParseFileHdr(char* pcFileName){
  FILE* poFile;
  char abHdr[100];
  unsigned long nBytesRead;
  // open the file
  if((poFile = fopen(pcFileName, "rb"))){
    // Move the pointer by rading the data
    fread(abHdr, sizeof(char), 100, poFile);
    fclose(poFile);
    // Get File Position after read
    nBytesRead = ftell(poFile);
    printf("BytesRead: %d 0x%08x\n", nBytesRead, nBytesRead);
    if(nBytesRead > 0){
      printf("Oops Reading Header!\n");
    }
  }
  else{
    printf("Unable to Open File!\n");
  }
}

int main(int argv, char** argc){
  ParseFileHdr(argc[1]);
  return 0;
}
```

Solution:

Without having to review the assembly, based on the C code alone I can spot where the overflow is, it has to do with `fread`. we are expecting something that is of size 100 bytes. So, in this case, we can provide it with something that is a size less than 100 bytes, and it will cause overflow and spit out the thing you sent, and anything information after. In order to let the overflow happen I created an empty .txt file and used the name as the an argument passed into the program.

Once I reran the program with the new empty .txt I ended up getting the error `"Oops reading header"`. This stores a value of -1 in the nBytesRead variable, or `0xffffff`. This is because the variable `nBytesRead` is an `unsigned int` type. it is holding the maximum value of close 65,000, but when signed it actually appears as -1. so it may print out -1 in being read, but if it is read as an unsigned integer it is actually a large number, whereas an unsigned integer should be smaller and include negatives.

Problem 2

Create, compile, and analyze your own integer overflow example. Explain the scenarios that your code is vulnerable to the overflow. Provide an example of how to cause the overflow. What would be the best way to mitigate the vulnerability?

Solution:

A really simple example example is an addition calculator. If we create an addition calculator and you put in numbers that are too big then it will cause an overflow when reading the numbers together, which could cause undesired output. This is a small example, but in much larger examples (i.e maybe how many views/likes a certain post has on social media) it could cause problems. What this would look like is:

```c
#include <stdio.h>

int main(){
  int number1, number2;
  printf("Give 2 numbers as input to add: ");
  scanf("%d %d", number1, number2);
  printf("Result is %d\n", (number1+number2));
  return 0;
}
```

if we add two numbers that add up larger than 2^32 bits, which is how many bits an integer can hold, it will result in a negative number which is an unwanted result. To fix this problem we can restrict, we can read in the numbers in a way where we can verify if it is not larger than 2^32 (more than likely as a character array), and then we can see if adding the two numbers together, results in an impossible result (example: two negative numbers can not be positive, two positive numbers can not be negative).

Problem 3

Compile the below program; open in disassembler and compare source code with assembled instructions. Examine the disassembly; can you identify the buffer overflow? Can you cause the overflow to happen?

```c
#include <stdio.h>
#include <string.h>

unsigned int IsValidUser(char* UserName);

int main(void){
  int tfAdmin = 0;
  char cUserName[8];

  printf("User name: ");
  gets(cUserName);

  if(IsValidUser(cUserName) || tfAdmin){
    printf("You have access!\n");
  }
  else{
    printf("Access Denied.\n");
  }

  return 0;
}

unsigned int IsValidUser(char* Username){
  return 0;
}
```

Solution:

The easiest way to cause the overflow is by putting in more than 8 characters. This happens because of the `gets` function. It stores more than 8 bytes of information if you put in more than 8 characters which then overwrites the `tfAdmin` value of 0 to some nonzero value, making the if statement on line 13 true and gives access to the user when they shouldn't have access.

Problem 4

Create, compile, and analyze your own buffer overflow example. Explain the scenarios that your code is vulnerable to the overflow. Provide an example of how to cause the overflow. What would be the best way to mitigate the vulnerability?

Solution:

Another built in function where the code is susceptable is through the string copy, we can see this in the function `strcmp` in the `string.h` library. I think a simple example, that can show this is:

```c
#include <stdio.h>
#include <string.h>

int main(){
  char guess[10];
  int passVal = 0;

  printf("Guess the password: ");
  scanf(" %s", guess);

  if(strcmp(guess, "password")){
    printf("You got access!\n");
  } else {
    printf("You don't have access.\n");
  }

  if(passVal){
    printf("Here's all the info!\n");
  } else {
    printf("No info for you!\n");
  }
```

```c

    return 0;
}
```

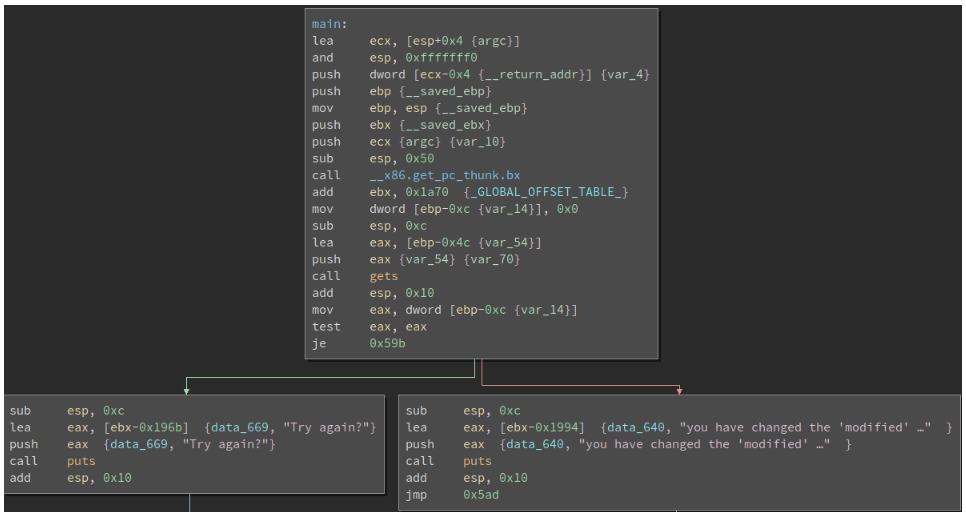
In the example above, if you feed a string with that has a length that is longer than 8 items you will cause an overflow, and some arbitrary character value's data will be stored inside the int `passVal` giving access to some accounts entire information.

Problem 5

Analyze the vulnerable binary named "p5" and solve. Write up your solution

Solution:

From my first impression I believe it is a buffer overflow. Initially when I ran the program I put in nothing as input and it said "`Try again?`". After this I tried to run the program again and put in an estimated 200 characters as input and it then said "`You have changed the 'modified' variable`". This seems to be contingent with what we saw in Problem 3, where `gets`, overwrote a variable. If using a `gets` would cause this issue, but I am going to review the binary to confirm of my suspicions.



```
main:
lea     ecx, [esp+0x4 {argc}]
and     esp, 0xfffffff0
push    dword [ecx-0x4 {__return_addr}] {var_4}
push    ebp {__saved_ebp}
mov     ebp, esp {__saved_ebp}
push    ebx {__saved_ebx}
push    ecx {argc} {var_10}
sub     esp, 0x50
call    __x86.get_pc_thunk.bx
add     ebx, 0x1a70  {_GLOBAL_OFFSET_TABLE_}
mov     dword [ebp-0xc {var_14}], 0x0
sub     esp, 0xc
lea     eax, [ebp-0x4c {var_54}]
push    eax {var_54} {var_70}
call    gets
add     esp, 0x10
mov     eax, dword [ebp-0xc {var_14}]
test    eax, eax
je      0x59b
```

```
sub     esp, 0xc
lea     eax, [ebx-0x196b]  {data_669, "Try again?"}
push    eax  {data_669, "Try again?"}
call    puts
add     esp, 0x10
```

```
sub     esp, 0xc
lea     eax, [ebx-0x1994]  {data_640, "you have changed the 'modified' …"  }
push    eax  {data_640, "you have changed the 'modified' …"  }
call    puts
add     esp, 0x10
jmp     0x5ad
```

My suspicions were confirmed true by looking at the binary, it seems the gets reads in too much data and overwrites some data, although I am not exactly certain how many characters I need it doesn't really matter because we are able to exploit and get the data we needed.

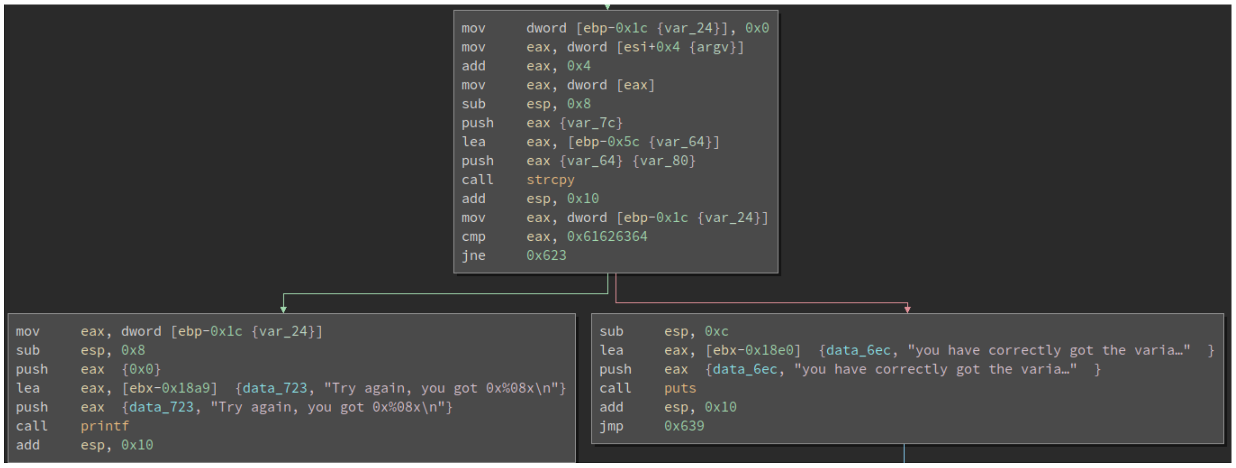
Problem 6

Analyze the vulnerable binary named "p6" and solve. Write up your solution

Solution:

My initial instinct was to think this was an integer overflow. I thought it was an integer overflow because when the program is run it asks "`p6: please specify an argument`". I suspected, similarly to Problem 1 that this requires some file, and reading a larger amount of data than available will cause an integer overflow. Let me try this out with an empty .txt file. When I passed an empty .txt file, running the command: `./p6 stuff.txt`, and it read out "`Try again, you got 0x00000000`". Okay, so maybe I am not supposed get some sort of specific read, let me try to put in an extremely large string as the argument instead of the .txt. I put in about 50 of the letter 'f' and got a different result, "`Try again, you got 0x66666666`". This instead seems to be a buffer overflow issue instead of an integer overflow. If I want to get any further I am going to need to observe the binaries.

From here, I decide to open up Binary Ninja, and look for where the fail text, and hopefully correct text is. I notice This in the binaries:



```
mov     dword [ebp-0x1c {var_24}], 0x0
mov     eax, dword [esi+0x4 {argv}]
add     eax, 0x4
mov     eax, dword [eax]
sub     esp, 0x8
push    eax {var_7c}
lea     eax, [ebp-0x5c {var_64}]
push    eax {var_64} {var_80}
call    strcpy
add     esp, 0x10
mov     eax, dword [ebp-0x1c {var_24}]
cmp     eax, 0x61626364
jne     0x623
```

```
mov     eax, dword [ebp-0x1c {var_24}]
sub     esp, 0x8
push    eax  {0x0}
lea     eax, [ebx-0x18a9]  {data_723, "Try again, you got 0x%08x\n"}
push    eax  {data_723, "Try again, you got 0x%08x\n"}
call    printf
add     esp, 0x10
```

```
sub     esp, 0xc
lea     eax, [ebx-0x18e0]  {data_6ec, "you have correctly got the varia…"  }
push    eax  {data_6ec, "you have correctly got the varia…"  }
call    puts
add     esp, 0x10
jmp     0x639
```

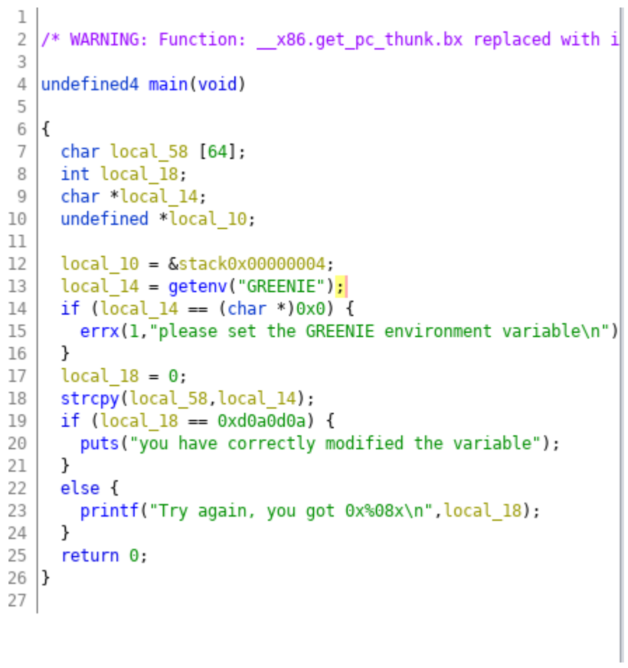
I see from here that the value I am looking for is `0x61626364`. Considering `f` correlates to values `66`, I can assume if we work backwards we will need to place the characters `dcba` at the right place. The problem is now the where, at what point do I place this `dcba`. I review the binaries some more to see exactly at what point I see it. Looking through, it seems that, after counting, it is exactly 64 characters before it starts to overflow into the integer. When we input 64 random chars and put in `dcba` there is a response of: "`You have correctly got the variable to the right value`".

Problem 7

Analyze the vulnerable binary named "p7" and solve. Write up your solution

Solution:

This problem seems similar to the last problem. Essentially, I have to set an environment variable named `GREENIE`. When doing this, and setting it to a random assortment of characters it tells you that your memory fails, and spits out a hex value. Very confusingly, I decided to open up what was going on in Ghidra. I found something like this:



```

/* WARNING: Function: __x86.get_pc_thunk.bx replaced with i

undefined4 main(void)

{
  char local_58 [64];
  int local_18;
  char *local_14;
  undefined *local_10;

  local_10 = &stack0x00000004;
  local_14 = getenv("GREENIE");
  if (local_14 == (char *)0x0) {
    errx(1,"please set the GREENIE environment variable\n")
  }
  local_18 = 0;
  strcpy(local_58,local_14);
  if (local_18 == 0xd0a0d0a) {
    puts("you have correctly modified the variable");
  }
  else {
    printf("Try again, you got 0x%08x\n",local_18);
  }
  return 0;
}

```

What I see here is that there is a clear buffer overflow from `local_58` into `local_18`. What we have to do is put the value of `0xd0a0d0a` inside the environment after an initial placement of 64 characters. This is quite easy to do. We have to set the environemnt to be this character set, and we can get the message that we correctly modified the variable.