CSE4501 – Vulnerability Research: Lab 1

Eric Pereira

Septemer 09th, 2019

Contents

Problem 1

Compile the below program; open in disassembler and compare source code with assembled instructions. Explain why the binary is 8K when it only has an empty function? Is there an oddity with the instructions assembled? Do you see that in the machine code directly?

```c
void main(){
}
```

Solution:

The binary is quite large, at a total of about 7.2kB on my VM, whereas the C code for is about 15 bytes. I decided I would start by not immediately looking at the binaries and just checking out a few things in the file itself. I did this by running "`nm problem1`".

What I got was quite a lot more than I expected, I got a large list of responses, it looks like there is actually quite a bit going on. I had quite a lot of stuff going on when I also ran "`strings problem1`", so I decided to give it a look. It turns out that there is in fact quite a lot going on.

It looks like there is quite a lot happening at the start of the compiled c file, it clears the stack immediately in the start. It appears that, although there is nothing really going on in the program, there is quite a lot of setup in order to create a clean c file in order to execute the commands you want to do. I suppose such a large initial set of instructions for cleanup is a large part of the reason why the compiled is so big.

Problem 2

Compile the below program; open in disassembler and compare source code with assembled instructions. What do you see on the stack frame? How is it different than Problem 1?

```
int main(){
   return 2;
}
```

Solution:

There is one major difference in the main function, that difference being that instead of a `nop` command, the value `2` is pushed into `EAX` and the popped from the stack. Essentially it behaves nearly similarly, however it pushes 2 onto the stack when running, whereas problem has a "`nop`" in its place instead.

One thing I noticed is that problem 2 executable file is exactly the same size as problem 1 executable. So the only difference I noticed is that one line where it pushes 2 onto the stack instead of a "`nop`", and because the file size is identical I assume that that is probably the only difference inside the actual executable, very interesting.

Problem 3

Compile the below program; open in disassembler and compare source code with assembled instructions. Specifically examine how the order of the stack is generated, do you see a difference? You will need to use a debugger for this exercise.

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

#define __cdecl __attribute__((__cdecl__))
#define __stdcall __attribute__((__stdcall__))
#define __fastcall __attribute__((__fastcall__))

int __cdecl print1(char string[]){
   printf("%s", string);
   return 0;
}

int __stdcall print2(char string[]){
   printf("%s", string);
   return 0;
}

int __fastcall print3(char string[]){
   printf("%s", string);
   return 0;
}

int main(int argc, char **argv){
   print1("First Call is cdecl.\n");
   print2("Second Call is stdcall.\n");
   print3("Third Call is fastcall.\n");
   return 0;
}
```

Solution:

Each individual print functions works ever so slightly differently, they mostly look identical though.

In fact, “`__cdecl`” and “`__stdcall`” are almost identical, there is one slight difference. The slight difference has to do with stack cleanup. In “`__stdcall`” the stack is cleaned up by the actual “`__stdcall`” itself, whereas in the “__cdecl” it is not. This is reflected in the binaries:

```
print1:
push    ebp {__saved_ebp}
mov     ebp, esp {__saved_ebp}
push    ebx {__saved_ebx}
sub     esp, 0x4
call    __x86.get_pc_thunk.ax
add     eax, 0x1aaf  {_GLOBAL_OFFSET_TABLE_}
sub     esp, 0x8
push    dword [ebp+0x8 {arg1}] {var_18}
lea     edx, [eax-0x1938]  {data_6a0}
push    edx {var_1c}  {data_6a0}
mov     ebx, eax  {_GLOBAL_OFFSET_TABLE_}
call    printf
add     esp, 0x10
mov     eax, 0x0
mov     ebx, dword [ebp-0x4 {__saved_ebx}]
leave    {__saved_ebp}
retn     {__return_addr}
```

Figure 1: print1 with “__cdecl”

```
print2:
push    ebp {__saved_ebp}
mov     ebp, esp {__saved_ebp}
push    ebx {__saved_ebx}
sub     esp, 0x4
call    __x86.get_pc_thunk.ax
add     eax, 0x1a7d  {_GLOBAL_OFFSET_TABLE_}
sub     esp, 0x8
push    dword [ebp+0x8 {arg1}] {var_18}
lea     edx, [eax-0x1938]  {data_6a0}
push    edx {var_1c}  {data_6a0}
mov     ebx, eax  {_GLOBAL_OFFSET_TABLE_}
call    printf
add     esp, 0x10
mov     eax, 0x0
mov     ebx, dword [ebp-0x4 {__saved_ebx}]
leave    {__saved_ebp}
retn    0x4 {__return_addr}
```

Figure 2: print2 with “__stdcall”

The difference is very clear when observing the last line, very interesting. Now there is also a difference with the print3 using the “__fastcall”. Let’s take a look at that.

```
print3:
push    ebp {__saved_ebp}
mov     ebp, esp {__saved_ebp}
push    ebx {__saved_ebx}
sub     esp, 0x14
call    __x86.get_pc_thunk.ax
add     eax, 0x1a49  {_GLOBAL_OFFSET_TABLE_}
mov     dword [ebp-0xc {var_10}], ecx
sub     esp, 0x8
push    dword [ebp-0xc {var_10}] {var_28}
lea     edx, [eax-0x1938]  {data_6a0}
push    edx {var_2c}  {data_6a0}
mov     ebx, eax  {_GLOBAL_OFFSET_TABLE_}
call    printf
add     esp, 0x10
mov     eax, 0x0
mov     ebx, dword [ebp-0x4 {__saved_ebx}]
leave    {__saved_ebp}
retn     {__return_addr}
```

Figure 3: print3 with “__fastcall”

“__fastcall” seems to use more of the registers initially, as when it moves the value of `ecx` which makes it a bit different. Also, where in the “__cdecl” and "__stdcall" it will `sub esp, 0x4` in the `__fastcall`'s `print3` it is instead `sub esp, 0x14`, very interesting.

Problem 4

Compile the below program; open in disassembler and compare source code with assembled instructions. Is the program using CDECL or STDCALL calling convention? Can you find the string in the binary? Where is the string on the heap? You will need to use a debugger for this exercise.

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

char msg[] = "Chris is the coolest";

int main(){
int len = strlen(msg);

char *msgptr;
msgptr = malloc(len*sizeof(char)+1);
memcpy(msgptr, msg, len*sizeof(char)+1);
printf("Msg: %s\n", msgptr);

return 0;
}
```

Solution:

Problem 5

Analyze the crackme named "crackme" and solve. The crackme can be found in the files share on canvas under Lab1

Solution:

This was quite frankly a very easy binary to review.

Before running the code I did a few things. The first thing I did was run `file crackme`. In a return I got an non-stripped, dynamically-linked ELF-32 bit executable. I then ran `nm crackme`, which then didn't show me anything very interesting, at least not interesting enough for me to pay attention. I lastly ran `strings crackme` and found something incredibly interesting.

What I found was a string "`Enter Password:`", which means that this code will have me inputting some sort of password. Conveniently under that I found the totally not suspicious string "`47ghf6fh37fbgbgj`". Very interesting.

At this point I decided I would run the code. I got the expected "`Enter Password:`". Upon this I entered the string I found, "`47ghf6fh37fbgbgj`", and got the output:

-[Good, You're ready to begin linux reversing]-.

Well, I suppose I got quite lucky, I didn't even have to review the binary, and the problem did'nt even tell me I had to review the binary, it said 'analyze the crackme'. I suppose my work is done then.

....

However, I still did want to analyze the binary so I did that anyways, I started by opening up the crackme in Binary Ninja.

Problem 6

Extra Credit (0 pts, begin accumulating your L33T status)

Analyze the crackme "save_scooby" and create a keygen. The crackme can be found in the files share on canvas under Lab1

Solution:

Extra credit with 0 points attached just seems like extra work, but extra work I shall do.

Primarily, I first tried to see the file type, which was surprisingly an ELF-64 bit. This means that I am unable to use Binary Ninja, because the trial version only allows for taking apart 32 bit binaries. Upsetting.

I then tried to see if I could "Cheat" by simply analyzing the strings like I did in the last problem, and, in a way, I sort of did fudge the "ideal" way of solving this problem.