Buffer-Overflow Attacks on the Stack

Introduction

- A buffer overflow occurs when a program, while writing data to a buffer, overruns the buffer's boundary and overwrites memory in adjacent locations. Programming languages commonly associated with buffer overflows include C and C++, which have no built-in protection against writing anywhere in memory.

- Buffer overflows are the basis of many software vulnerabilities and can be maliciously exploited. They have been used by malicious hackers and virus writers for a long time.

- In this lab we will experiment with harmless buffer overflows that trigger calls to unused functions in our program. A classic exploit would however fill the buffer with its own malicious code.

- If you are interested in digging deeper into buffer-overflow techniques, refer to https://www.symantec.com/avcenter/reference/blended.attacks.pdf

Part1 – Understanding Array Allocation and Vulnerable C Functions

1. Copy bufdemo.c from /mnt/a/mdamian/x86 into your ~/systems/x86 directory.

    ```c
    #include <stdio.h>
    
    void Echo()
    {
        char buf[4];
        gets(buf);
        puts(buf);
    }
    
    int main()
    {
        printf("\n Type in a string: ");
        Echo();
        return 0;
    }
    ```

1. Compile with gcc bufdemo.c -o xbufdemo. Ignore the warnings.
2. Load xbufdemo into the debugger (gdb), then disassemble the code for main:

```bash
gdb xbufdemo
set disassembly-flavor intel
disas main
```

**Disassembled code for main:**

```
0x0804840c <main+10>: push ebp
0x0804840d <main+11>: mov ebp,esp
0x0804840f <main+13>: push ecx
0x08048410 <main+14>: sub esp,0x4
0x08048413 <main+17>: mov DWORD PTR [esp],0x8048504
0x0804841a <main+24>: call 0x8048320 <puts@plt>
0x0804841f <main+29>: call 0x80483e4 <Echo>
0x08048424 <main+34>: mov eax,0x0
0x08048429 <main+39>: add esp,0x4
0x0804842c <main+42>: pop ecx
0x0804842d <main+43>: pop ebp
0x0804842e <main+44>: lea esp,[ecx-0x4]
0x08048431 <main+47>: ret
```

3. Next disassemble the code for Echo:

**Echo:**

```
0x080483e4 <Echo+0>: push ebp
0x080483e5 <Echo+1>: mov ebp,esp
0x080483e7 <Echo+3>: sub esp,0x18
0x080483ea <Echo+6>: lea eax,[ebp-0x4]
    # lea stands for “Load Effective Address”
    # The effect is eax ← ebp-4
0x080483ed <Echo+9>: mov DWORD PTR [esp],eax
    # The value placed on the stack is buf (argument of gets)
0x080483f0 <Echo+12>: call 0x8048300 <gets@plt>
0x080483f5 <Echo+17>: lea eax,[ebp-0x4]
0x080483f8 <Echo+20>: mov DWORD PTR [esp],eax
0x080483fb <Echo+23>: call 0x8048320 <puts@plt>
0x08048400 <Echo+28>: leave
0x08048401 <Echo+29>: ret
```
4. Inspect the stack contents as you step through the program with the debugger:

```
break *(Echo+12) # Just before gets gets called
display /i $eip
run
x /x $ebp-4 # This is the buf address
nexti # Use nexti so that you don’t go into the gets function
# Type in the string 123
x /4bc ($ebp-4) # Display 4 bytes as chars starting at memory address $ebp-4
x /4bx ($ebp-4) # Display 4 bytes in hex starting at memory address $ebp-4
x /1wx ($ebp-4) # Display 1 word (4 bytes) in hex starting at address $ebp-4
# Is this machine little endian or big endian?
x /4wx ($ebp-4) # Display 4 words (16 bytes) in hex starting at address $ebp-4
# You’ll see the user string (4 bytes), the old EBP, and the
# address to return to the main function RA (main)
disas main # Find out the return address, and identify it on the stack
```

5. Rerun `xbufdemo` (in the debugger) with input string 1234. What are the four values printed by

```
x /4wx $ebp-4
```

```
0x34333231 0x??????00 0x08048424 0x???????
```

Marked in red above is the return address for main.  
Marked in blue above is your input string (ASCII codes of the characters you entered).
6. Rerun `xbufdemo` (in the debugger) with input string `1234567`. What are the four values printed by

\[
\text{x /4wx $ebp-4}
\]

\[
0x34333231 \ 0x00373635 \ 0x08048424 \ 0x?????????
\]

Marked in red above is the return address for main.  
Marked in blue above is your input string (ASCII codes of the characters you entered).

7. Rerun `xbufdemo` (in the debugger) with input string `123400005678`. What are the values printed by

\[
\text{x /4wx ($ebp-4)}
\]

\[
0x34333231 \ 0x30303030 \ 0x38373635 \ 0x???????00
\]

\[
\text{x /12bx ($ebp-4)}
\]

\[
0x31 \ 0x32 \ 0x33 \ 0x34 \ 0x30 \ 0x30 \ 0x30 \ 0x30 \ 0x35 \ 0x36 \ 0x37 \ 0x38
\]

8. What happened to the return address for the main function? Write down the address that the execution jumps to when the `ret` instruction in `Echo` gets executed:

\[
0x38373635
\]
Part 2 – Understanding Buffer Overflow Attacks

2. Copy bufattack.c and sendstring (an executable file) from /mnt/a/mdamian/x86 into your systems/x86 directory.

```c
#include <stdio.h>
#include <stdlib.h> /* for exit */

void Echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}

void Fire()
{
    printf("\nYou called Fire?!\n")
    exit(1);
}

int main()
{
    printf("\nType a string: \n")
    Echo();
    return 0;
}
```

3. Compile with gcc bufattack.c –o xbufattack. Ignore the warnings.

4. Load xbufattack into the debugger (gdb), then disassemble the code for main:

   ```
gdb xbufattack
set disassembly-flavor intel
disas main
```

   **Code of interest (main):**
   ```
   0x08048468 <main+24>:  call  0x8048340 <puts@plt>
   0x0804846d <main+29>:  call  0x8048414 <Echo>
   0x08048472 <main+34>:  mov  eax,0x0
   ...
   ```

5. Disassemble the code for Echo:

   ```
   Code of interest (Echo):
   0x08048414 <Echo+0>:  push  ebp
   0x08048415 <Echo+1>:  mov  ebp,esp
   ...
   0x0804841a <Echo+6>:  lea  eax,[ebp-0x4]
   0x0804841d <Echo+9>:  mov  DWORD PTR [esp],eax
   ```
6. Disassemble the code for Fire:

```asm
0x08048432 <Fire+0>:    push    ebp
0x08048433 <Fire+1>:    mov     ebp,esp
0x08048435 <Fire+3>:    sub     esp,0x8
0x08048438 <Fire+6>:    mov     DWORD PTR [esp],0x8048544
0x0804843f <Fire+13>:   call    0x8048340 <puts@plt>
0x08048444 <Fire+18>:   mov     DWORD PTR [esp],0x0
0x0804844b <Fire+25>:   call    0x8048350 <exit@plt>
```

7. Inspect the stack during the program execution by stepping through with the debugger. Here are some debugger commands to get you started:

```
break *(Echo+12)      # Prior to the gets call
display /i $eip      # Display /i $eip
run                  # Run
x /3xw $ebp-4        # Type in 123400005678
nexti                # Type in 123400005678
x /3xw $ebp-4        # Note the changes in the stack contents. Identify your input
                    # Notice the little endian representation
```

![Stack Diagram](image)
8. Write down an exploit string in hexadecimal that causes Fire to execute when gets terminates. Use the address of Fire to overwrite the old EBP as well.

![Diagram of EBP and Address](image)

**Exploit string:**

The first 4 bytes should fill in the buffer.
The next 4 bytes should overwrite the old EBP. You may use any valid address here – so just use the address of the Fire function.
The last 4 bytes should overwrite the return address for main with the address of Fire.

Note that there should be no spaces in the exploit string when passed to the sendstring function (see below).

9. To test out the correctness of your answer in the previous step, we will need to convert your hexadecimal string into a byte string that will serve as input to your program. To do so, first quit the debugger. Write your exploit string into a file, say `in.txt`, then use `sendstring` to produce the corresponding byte string as follows:

```
sendstring < in.txt > in.raw
```

The result is a binary file `in.raw` that contains the bytes you wish to feed to the `xbufattack` program.

10. Reinvoke the debugger on `xbufattack`, then repeat the activities from step 6 above, with the only difference that this time you will provide your exploit string as input when running:

```
run < in.raw
```
11. Step through the code (using nexti or ni). If your exploit string is correct, the execution will jump to the first instruction in Fire after the ret instruction from Echo.

12. Copy bufattack.c into bufattack2.c, then edit bufattack2.c to add two arguments to Fire as shown below.

```c
void Fire(int x, int y)
{
    printf("\nYou called Fire(%x, %x)?!\n", x, y);
    exit(1);
}
```

Compile bufattack2.c into executable xbufattack2, then repeat steps 7 through 10 on the new executable. Your goal is to force Fire(0x11, 0x22) to execute when gets terminates. (Note that the address of Fire may have changed).

Write down the exploit string (in hexadecimal):

```
_____________________________________________________________
The first 4 bytes should fill in the buffer.
The next 4 bytes should overwrite the old EBP. You may use any valid address here - so just use the address of the Fire function.
The next 4 bytes should overwrite the return address for main with the address of Fire.
The next 4 bytes should simulate the return address of the function that called Fire (in this case we forced the call, so place any valid address here).
Then come the function arguments, in little endian format.

Note that there should be no spaces in the exploit string when passed to the sendstring.
```