Uporabniški vnos:

- vedno je problem z njim, uporabnik drugače razume uporabo appa kot programer
- vedno lahko naredimo kakšno napako, da ne shendlamo pravilno nekega inputa

Validacija inputa:

- preverimo, da je uporabnik vpisal pravo stvar - pogosto se zakomplicira
- ne zaupaj userju
- ORM v bazah - opišemo vrstice kot entitete z nekimi lastnostmi - to je potem lažje delati kompleksne constrainte
- validiramo na vseh korakih - frontend in backend
- cross site scripting (XSS), SQL injection
- ne moremo se popolnoma zaščititi, lahko se pa potrudimo
- varnost je overhead:
  - aplikacija bo počasneje delala
  - uporabniki bodo zafrustgrirani, če rabijo več delati
  - lahko pokvarimo stabilnost sistema
- danes je zelo malo softwarea stand alone, vse je web

Problemi pri C:

- prekoračitev medpomnilnika (buffer):
  - Gets, scanf, strcpy še vedno uporabljamo nekje zaradi legacy
- preverjanje omejitev:
  - koliko imamo bufferja, vedno delamo v runtime, torej nas stane časa
- ASLR:
  - skrivamo lokacijo v fizičnem pomnilniku, lahko pokvarimo pomnilniško lokalnost in bo predpomnilnik počasneje delal

    - da CPU ne čaka imamo več CPU ali pa delamo out of order execution (delamo nekaj kar bomo mogoče rabili, medtem ko čakamo na pomnilnik)
- data execution prevention:
    - označimo dele pomnilnika, od koder ne moremo zaganjati kode
- SEHOP:
    - povemo, da nekega dela pomnilnika proces ne sme prepisati - če ga, je nekaj narobe in ustavi program (npr. kanarček)

Primer iz vaj:

-
```c
#include <stdio.h>

void win() {
 printf("You win!\n");
}

int main() {
 char buffer[20];
 gets(buffer);

 printf("%s\n", buffer);
 return 0;
}
```
- če vpišemo predolg niz dobimo seg fault, ker skoči na nek drug segment, ko prepišemo return address
- `objdump -d main` lahko dobimo naslov win funkcije
- rbp, rsp na začetku - rezervira se frame in nastavi se prostor za spremenljivke
-
```
0x401165 <main+25>    lea    rax, [rbp - 0x20]
0x401169 <main+29>    mov    rdi, rax
0x40116c <main+32>    call   puts@plt                  <puts@plt>
0x401171 <main+37>    mov    eax, 0
0x401176 <main+42>    leave
▶ 0x401177 <main+43>    ret                          <0x401136; win>
```
- `leave` - prestavi base pointer na začetek prejšnjega stacka
- `ret` skoči na naslov iz return address in ga da dol iz stacka

Primer:

- ```
  int bytes; char buf[64], in[MAX_SIZE];
  printf("Enter buffer contents:\n");
  read(0, in, MAX_SIZE-1);
  printf("Bytes to copy:\n");
  scanf("%d", &bytes);
  memcpy(buf, in, bytes);
  ```
- `memcpy` samo skopira nekaj nekam
- ne moremo se znebiti `memcpy`
- npr. v Javi ArrayList `clone` - naredi novo instanco objekta z istimi referencami, ker naredi samo shallow copy; če hočemo narediti deep copy, bomo rabili memcpy, ker hočemo dejansko skopirati vsebino iz ArrayLista
- MAX_SIZE-1, ker rabimo še prostor za \0
- ko beremo, če je v `in` nekaj od prej ostalo, ni nujno, da bomo prebrali \0 in bomo imeli ne terminiran string
- po read rabimo dodati še `in[MAX_SIZE-1] = \0;`
- pri "Bytes to copy: " lahko damo poljubno št. bytov za kopirat
- seg fault - tisti memory segment ne obstaja ali pa nimamo dovoljenja za dostop do njega

Primer:
  ```

```c
#include <stdio.h>

void win() {
 printf("You win!\n");
}

int main() {
 setbuf(stdin, NULL);
 setbuf(stdout, NULL);

 char buffer[20];
 char choice = 'n';

 do
 {
  printf("Enter your name: ");
  gets(buffer);

  printf(buffer);
  printf("\n\nIs that correct? [y/n]\n");
  choice = getchar();
  getchar();
 } while (choice != 'y');

 printf("Hello, %s!\n", buffer);

 return 0;
}
```

- problem `gets` - če vpišemo zelo dolg niz, bomo povozili kanarčka in dobimo stack smashing detected
- kanarček = 7 random bytov
- zadnji byte kanarčka je postavljen na 0, da ne moremo izpisati kanarčka, če prepišemo točno do pred koncem kanarčka
- če buffer prepišemo tako, da ne damo na koncu \0, se nam izpiše vse do konca canary in ko vemo canary, ga lahko nazaj zapišemo in stack smashing ne bo detected
- ali je kanarček živ, se preveri preden skočimo ven iz gets funkcije
- problem printf: lahko damo noter format, npr. vpišemo `%p`
- argumenti funkcij grejo na registre, če imamo preveč argumentov, grejo na stack; na stacku imamo tudi canary => če damo v printf prave argumente, bomo izpisali kanarčka, npr. pri `%31$p` se zadeva ravno prav premakne po stacku, da nam izpiše

kanarčka

SMTP primer:

```c
int main()
{
 char **segments = malloc(256 * sizeof(char*));
 char recipient[256];
 printf("Enter recipient: ");
 fgets(recipient, 256, stdin);
 recipient[strlen(recipient) - 1] = 0;

 uint8_t segment = 0;
 segments[segment] = malloc(256 * sizeof(char));
 strcpy(segments[segment], "MAIL FROM: <user@example.com>");
 segment++;
 segments[segment] = malloc(256 * sizeof(char));
 sprintf(segments[segment], "RCPT TO: <%s>", recipient);
 segment++;
 segments[segment] = malloc(256 * sizeof(char));
 strcpy(segments[segment], "DATA");
 segment++;

 printf("Enter message:\n");
 char *line = malloc(256 * sizeof(char));
 while (fgets(line, 256, stdin) != 0 && strlen(line) > 1) {
  segments[segment] = malloc(256 * sizeof(char));
  line[strlen(line) - 1] = 0;
  strcpy(segments[segment], line);
  segment++;
 }

 segments[segment] = malloc(256 * sizeof(char));
 strcpy(segments[segment], "");

 printf("SMTP message:\n");
 for (int i = 0; i <= segment; i++) {
  if (strlen(segments[i]) == 0) {
   break;
  }
  printf("%s\n", segments[i]);
 }
```

- problem da za mail lahko napišemo 255 characterjev, potem ko dodamo prefix in suffix gre lahko čez 256 bytov, ki smo jih rezervirali
- `fgets` vrne število prebranih bytov, če je error vrne negativno število, zato ga castamo v int

- če imamo zelo dolg mail, se dodajo segmenti do 256 segmentov
- `segment` je `uint8_t`, torej ko pride do 256, se nastavi nazaj na 0 => spet pišemo na začetek, torej lahko celoten mail napišemo po svoje, ker lahko prepišemo header (integer overflow)

Primer:

```
114     numSyms = 0;
115     nRefSegs_1 = nRefSegs;
116     refSegs_1 = (int *)refSegs;
117     v28 = nRefSegs;
118     do
119     {
120       Segment = (JBIG2SymbolDict *)JBIG2Stream::findSegment(this, *refSegs_1);
121       if ( !Segment )
122       {
123         v47 = (*(__int64 (__fastcall **)(JBIG2Stream *))(*(_QWORD *)this + 40LL))(this);
124         error(v47, "Invalid segment reference in JBIG2 text region");
125         j__free(*(void **)v106);
126         operator delete(v106);
127         return;
128       }
129       v30 = Segment;
130       if ( Segment->vfptr->getType(Segment) == jbig2SegSymbolDict )
131       {
132         numSyms += v30->size;
133       }
134       else if ( v30->vfptr->getType(v30) == jbig2SegCodeTable )
135       {
136         GList::append(v106, v30);
137       }
138       ++refSegs_1;
139       --v28;
140     }
141     while ( v28 );
142     v89 = v12;
143     v91 = v14;
144     v31 = 0;
145     if...
146     syms = (_QWORD *)gmallocn(numSyms, 8u);
147     i_1 = 0LL;
148     k = 0LL;
149     do
150     {
151       seg = (JBIG2SymbolDict *)JBIG2Stream::findSegment(this, refSegs[i_1]);
152       if ( seg
153         && (symbolDict = seg, seg->vfptr->getType(seg) == jbig2SegSymbolDict)
154         && (size = symbolDict->size, (_DWORD)size) )
155       {
156         bitmaps = symbolDict->bitmaps;
157         do
158         {
159           v40 = (__int64)*bitmaps++;
160           kk = (unsigned int)(k + 1);
161           syms[(unsigned int)k] = v40;          // crash here !!!
162           LODWORD(k) = k + 1;
163           --size;
164         }
165         while ( size );
166       }
167       else
168       {
169         kk = k;
170       }
171       ++i_1;
172       k = kk;
173     }
174     while ( i_1 != nRefSegs_1 );
```

00085228 __ZN11JBIG2Stream17readTextRegionSegEjiijPjj:161 (181D6E228)

- ko je v40 > syms, lahko v sysms pišemo neke svoje ukaze