Ohtlikud koodilõigud

Erkki Moorits
Cybernetica AS, Navigatsioonisüsteemide osakond

Loengu eesmärk

Anda ülevaade programmi võimalikest kohtadest mis võivad sardsüsteemidel ohtlikuks osutuda. Lisaks on toodud mõningad näiteid ohtlike koodilõikude ja koodi suuruse vähendamise kohta. Järgnev loeng on C keele spetsiifiline.

Loengu teemad

- Ohtlikud koodilõigud
- Optimaalne kood
- Staatiline koodi kontroll
- Kodeerimis stiil

Ohtlikud koodilõigud

- Tüübi muutmine (*casting*)
- Tsüklid (do, while, for)
- Kaudsed funktsioonid (*indirect functions* või *indirect calls*)
- *Local* ja *nonlocal* goto
- Tühjad tsüklid
- Sisendandmete kontroll ja selle tegemata jätmine

Tüübi muutmine (Casting) I

- Muutujate tüübi muutmine väiksema maksimumväärtusega tüübilt (ühtlasi ka väiksemalt tüübilt) suuremale ei kujuta reeglina ohtu.
  - Näiteks `uint16_t` tüüpi muutuja (16 bitine) muutmisel `uint32_t` tüüpi muutujaks (32 bitine) ei kujuta piisava mälu juures ohtu.
- Muutuja tüübi muutmisel suurema maksimumväärtusega tüübilt väiksema maksimum väärtusega tüübile peab **alati** tegema eelneva kontrolli.
  - Erandiks on ainult juhus kus ei saa suurema maksimum väärtusega muutuja mitte mingil juhul olla väiksemast muutujast suurem, näiteks kui 32 bitine muutuja on jagatud 2^17'ga ja tulemus muudetud 16 bitiseks.

Tüübi muutmine (Casting) II

- Viida tüüpide (pointerite) suuremaks muutmisel tuleb olla eriti tähelepanelik, näiteks kui muudetakse 16 bitistele andmetele viitav viit 32 bitistele andmetele viitavaks viidaks.
  - C kompilaator ei anna mitte mingit vea indikatsiooni kui tüübi muutmise tagajärjel rikutakse teise funktsiooni mälu ära
  - PC Linuxi all kirjutatud programm mis kasutab vigast viidatüübi teisendust reeglina pannakse kinni, kuid kontrolleritel tekitab selline viga terve süsteemi kokkujooksmist
  - Sellist muutmist annab suhteliselt ohutult teha massiividega, näide järgmisel slaidil

Viidatüübi muutmine massiivis

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

int main (void)
{
    uint8_t arr[] = {0x11, 0x22};
    uint16_t *ptr;

    ptr = (uint16_t*)&arr[0];
    printf ("arr: 0x%02x, 0x%02x\n", arr[0],
                                     arr[1]);
    printf ("ptr: 0x%04x\n", *ptr);

    /* vigane pointeri tüübi muutmine */
    ptr = (uint16_t*)&arr[1];
    printf ("ptr: 0x%04x\n",*ptr);

    return 0;
};

/* väljund */
arr: 0x11, 0x22
ptr: 0x1122
ptr: 0x2200
```

NB! kolmandal väljundreal on kaks viimast numbrit nullid, see tuleneb sellest, et operatsioonisüsteem on kas mälu eest ära nullinud (turvalisuse kaalutlustel) või on juhtunud lihtsalt tühjale alale. Sellist käitumist ei tasu eeldada teistel süsteemidel.

Tüübi muutmine (Casting) III

- Suvalise viidatüübi (näiteks `uint8_t *ptr1`) muutmine tühjaks viidatüübiks (`void *ptr2`) kaotab ära igasuguse tüübi informatsiooni.
  - C puhul on tüübi informatsioon oluline ainult kompilaatorile ja programmeerijale endale, lõplik masinkoodis programm ei erine mitte mingil määral korrektsete tüüpidega programmist

Tsüklid (do, while, for)

- Do, while ja for tsüklid on ühed enimlevinud potentsiaalselt ohtlikud koodilõigud
- Tüüpilised ohtlikud kohad
  - Threadide töötsüklid
  - Pollimine

Threadide töötsüklid ja pollimine

```
for (; 1;)
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;
    /* ... veel mingi kood ... */
};
/*******************************/
while (1)
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;
    /* ... veel mingi kood ... */
};
/*******************************/
do
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;
    /* ... veel mingi kood ... */
}
while (1);
```

- Threadide töötsükli juhul on võimalik jooksutada cooperative schedureriga kernel sellise koodiga kokku, seda juhul kui ei lisata mingit ajalise viitega koodi
- Pollimise juhul on võimalik, et programm ei jõua kunagi siit kaugemale

Pollimine (korrektne näide)

```
/* maksimaalne i väärtus on 2^16 */
uint16_t i;

for (i = 0; i < 0xffff; i++)
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;

    /* ... veel mingi kood ... */
};

if (i == 0xffff)
    puts ("ei olnud aktiivsust");
```

Lisatud on üks 16-bitine muutuja, mille täitudes tsükkel lõppeb ja väljastatakse teade.

Pollimine (vigane näide)

```
/* maksimaalne i väärtus on 2^16 */
uint16_t i;

for (i = 0; i <= 0xffff; i++)
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;

    /* ... veel mingi kood ... */
};

if (i == 0xffff)
    puts ("ei olnud aktiivsust");
```

Kuna muutuja i on 16 bitine (maksimaalväärtus on 0xFFFF), siis võrdlemine 0xFFFF 'ga annab **alati** tõese väärtuse ja tsüklist ei pruugita mitte kunagi väljuda. Kompilaator ei tarvitse seda viga tavalise hoiatuse taseme juures näidata, et näitaks tuleb gcc'l lülitada sisse **-Wall** ja **-Wextra** vigade indikatsioon.

Tsükli kasutamine lõplikus threadis (korrektne näide)

```
/* lõplik thread */

while (1)
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;

    /* mitteaktiivne 10 ajaühikut */
    SLEEP (10);

    /* ... veel mingi kood ... */
};

puts ("PA3 või PB4 oli aktiivne");
```

Igas tsüklis on üks 10 ajaühiku pikkune viide, selline viide annab võimaluse teistel threadidel samaaegselt oma ülesandeid täita, sh võimaluse watchdog'i taimerit nullida.

Tsükli kasutamine lõputus threadis (korrektne näide)

```
/* lõputu thread */

while (1)
{
    /* mitteaktiivne 10 ajaühikut */
    SLEEP (10);

    if (!(PORTA & (1 << PA3)))
        continue;
    else if (!(PORTB & (1 << PB4)))
        continue;

    puts ("PA3 ja PB4 olid aktiivsed");
    /* ... veel mingi kood ... */
};
```

Ajaline viide peab olema alati sellises kohas kus tsükkel käib alati läbi.

Kaudsed funktsioonid

```
#include <stdio.h>

static void fn_1()
{
    puts ("Hello world!");
};

void (*indirect_fn)(void) = fn_1;

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

- Kaudset funktsiooni kutsutakse välja kasutades selleks vastavat viita (pointerit), seejuures see viit on ise sisuliselt muutuja mille tüüp on funktsioon
- Peamiseks kasutuskohaks on kernelis scheduleriga seotud funktsioonid ja draiverid

Kaudsed funktsioonid II

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

typedef struct fp_struct
{
    uint8_t num;
    uint8_t (*func) (uint8_t *p1);
    struct fp_struct *next;
} FP_STRUCT;

uint8_t fn_1 (uint8_t *p1)
{
    printf ("fn_1\t[p1 == %u]\t", *p1);
    return (*p1) + 1;
};

uint8_t fn_2 (uint8_t *p1)
{
    printf ("fn_2\t[p1 == %u]\t", *p1);
    return (*p1) + 10;
};
```

```
int main (void)
{
    uint8_t i, rc;
    FP_STRUCT *fn, *fn_last, *fn_first;

    fn = malloc (sizeof (FP_STRUCT)*2);
    fn_first = fn;
    fn->func = fn_1;
    fn->num = 0;
    fn_last = fn;
    fn++;
    fn_last->next = fn;
    fn->func = fn_2;
    fn->num = 1;
    fn->next = fn_first;

    for (i = 0; i < 5; i++)
    {
        rc = fn->func (&i);
        printf ("fn: %u; rc: %u\n",
                fn->num, rc);
        fn = fn->next;
    };
    free (fn_first);
    return 0;
};
/* väljund */
fn_2    [p1 == 0]       fn = 1; rc = 10
fn_1    [p1 == 1]       fn = 0; rc = 2
fn_2    [p1 == 2]       fn = 1; rc = 12
fn_1    [p1 == 3]       fn = 0; rc = 4
fn_2    [p1 == 4]       fn = 1; rc = 14
```

Probleemid kaudsete funktsioonidega

- Võimalik tööajal funktsiooni aadressi muuta
  - Tüüpiline olukord – üks funktsioon muudab pointeriga valet aadressi, seejuures pole vahet kas pointer on suunatud valele aadressile või on tegemist vale castinguga
- Väga lihtsalt võimalik valesid parameetreid ette anda ja vale tagastusväärtuse saada

Goto, adresseeritud goto ja setjmp/longjmp (non-local goto)

- Kõik goto käsud võimaldavad teha programmis suhteliselt suvalisi hüppeid kuid:
  - Local goto on piiratud ainult ühe funktsiooniga
  - Adresseeritud goto võimaldab tööd jätkata suvaliselt aadressilt
  - „Non-local goto“, ehk setjmp/longjmp võimaldavad hüpata ühest funktsioonist teisse funktsiooni
- Goto kasutamise juures tuleb silmas pidada järgnevaid kitsendusi:
  - Goto käsuga on võimalik üle hüpata muutujate algväärtustamistest, või siis stacki operatsioonidest
  - Goto käsk teeb reeglina koodi raskemini loetavaks

Goto ja adresseeritud goto

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

static void f1 (void);

int main (void)
{
    uint8_t i;
    uint8_t j;
    void *label_2 = &f1;

    for (i = 0, j = 0; i < 7; i++)
    {
        printf ("i = %u, j = %u\n",i,j);

        if (i == 2)
            goto label_1;
        if (i == 5)
            goto *label_2;

        j++;
        continue;
label_1:
        puts ("label_1");
    };

    return 0;
};
```

```
static void f1 (void)
{
    puts ("function f1");
    exit (0);
};

/***********************************/
Programmi väljund

i = 0, j = 0
i = 1, j = 1
i = 2, j = 2
label_1
i = 3, j = 2
i = 4, j = 3
i = 5, j = 4
function f1
```


Goto käskude kasutuskohad

- Goto käskudest õigustab kõige paremini oma kasutamist adresseeritud goto.
  - Kasutatakse ühe programmi lõppedes teise programmi käivitamiseks, näiteks bootloaderi lõppedes põhiprogrammi välja kutsumiseks või ohutuskriitilises rakenduses põhiprogrammi veaga lõppedes ohutust tagava funktsiooni käivitamiseks.
- Enamusjaolt kipuvad goto kasutamised näitama viletsat programmerimis praktikat

Tühjad tsüklid I

- Tsüklid mille eesmärk on ainult niisama teatud aja jooksul protsessori aega kulutada on nn. tühjad tsüklid
  - Kui võimalik, siis on parem neid mitte kasutada, vt. eelnevate tsüklite probleeme
  - Programmi täitmise kiirus tühjade tsüklite juures sõltub väga suurel määral kompilaatori optimeerimistasemest
  - Üpriski tülikas on teha tühja tsüklit mis oleks porditav ja ei kaoks erinevate optimeerimistega ära

Tühjad tsüklid II

```
#include <stdio.h>
#include <stdint.h>
#include <sys/time.h>

#define LOOP_MAX_VAL 0xffffffff
static uint8_t val;
static long int get_tdiff (struct timeval t1,
                           struct timeval t2);

int main (void)
{
    struct timeval start;
    struct timeval end;
    uint32_t i;

    gettimeofday (&start, NULL);
    /* intuitiivne variant */
    for (i = 0; i < LOOP_MAX_VAL; i++);

    gettimeofday (&end, NULL);
    printf ("%ld us\n", get_tdiff (start,end));
    gettimeofday (&start, NULL);

    /* korrektne variant */
    for (i = 0; i < LOOP_MAX_VAL; i++)
        asm volatile ("nop");

    gettimeofday (&end, NULL);
    printf ("%ld us\n", get_tdiff (start,end));
    val = (uint8_t)(0xff & end.tv_sec);

    return 0;
};
```

```
static long int get_tdiff (struct timeval t1,
                           struct timeval t2)
{
    long int v1 = (t1.tv_sec * 1000000 +
                   t1.tv_usec);
    long int v2 = (t2.tv_sec * 1000000 +
                   t2.tv_usec);

    return (v2 - v1);
};
```


Täitmata tsüklid

```
/* Optimeerimistasemega O0 kompileeritud
 * programm */
9440907 us
4414831 us

/* Optimeerimistasemega O3 kompileeritud
 * programm */
0 us
712646 us
```

Optimeeritud programmi töö on ühe suurusjärgu võrra kiirem kui optimeerimata programmil, lisaks on optimeeritud programmist tühi tsükkel välja jäetud

Täitmata tsükli volatile variant

```
#include <stdio.h>
#include <stdint.h>
#include <sys/time.h>

#define LOOP_MAX_VAL 0xffffffff
static uint8_t val;
static long int get_tdiff (struct timeval t1,
                           struct timeval t2);

int main (void)
{
    struct timeval start;
    struct timeval end;
    volatile uint32_t iv;
    uint32_t i;

    gettimeofday (&start, NULL);
    /* intuitiivne variant */
    for (iv = 0; iv < LOOP_MAX_VAL; iv++);

    gettimeofday (&end, NULL);
    printf ("%ld us\n", get_tdiff (start,end));
    gettimeofday (&start, NULL);

    /* korrektne variant */
    for (i = 0; i < LOOP_MAX_VAL; i++)
        asm volatile ("nop");

    gettimeofday (&end, NULL);
    printf ("%ld us\n", get_tdiff (start,end));
    val = (uint8_t)(0xff & end.tv_sec);

    return 0;
};
```

```
/* Optimeerimistasemega O0 kompileeritud
 * programm */
9703093 us
4418082 us

/* Optimeerimistasemega O3 kompileeritud
 * programm */
2361333 us
594329 us
```

Selles näites on tsükli muutuja *i* asendatud muutujaga *iv*, mis on *volatile*. Muutujat mis on märgitud kui *volatile* ei optimeerita, seega võtab sellise muutuja kasutamine rohkem programm mälu ja tõenäoliselt ka RAM'i, kuna muutujat ei hoita registrites.

Alternatiivne tühja tsükli variant

```
/* alternatiiv variant */
uint8_t val = PORTB;

for (i = 0; i < LOOP_MAX_VAL; i++)
{
    if (val != PORTB)
        break;
};
```

Kui lasta programmil võrrelda mingit kindlat registrit mis mitte kunagi ei muutu, siis ei ole võimalik kompilaatoril seda koodilõiku välja optimeerida. Selline võiks olla ka assembleris käsu nop C's kirjutatud asendus.

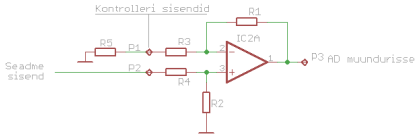
NB! selline tühi tsükkel ei ole porditav

Sisendandmete kontroll ja selle tegemata jätmine

- Kõiki sisendandmeid mille baasil võetakse vastu mingi otsus või on võimalus, et mõjutatakse süsteemi tööd tuleb alati kontrollida, näiteks:
  - Aku laadimisel aku pinge ja laadimisvoolu jälgimine
  - Süsteemidel millele on võimalik palju andmeid saata ja mis kõik pannakse enne töötlust sisendpuhvrisse, nendel tuleb kontrollida et sisendpuhvrit üle ei täidetaks (serverite puhul tüüpiline kontroll)
- Sisendandmeid mis ei ole võimelised mõjutama mitte mingil moel süsteemi tööd mõjutama, näiteks:
  - Kui ADC annab temperatuurianduri mõõtetulemuseks vale väärtuse ja see väärtus on ainult indikatsiooniks

Reaalne näide: digitaalne filter

AD muunduri väljund annab 10 bitise väärtuse, seda nii diferentsiaal sisendite puhul kui ka tavalise sisendi puhul (*single end*), kuid digitaalse filtri jaoks on vaja 9 bitist väärtust. Kui kasutada diferentsiaal sisendiga AD muundurit, ühendades ühe signaali maaga, siis on võimalik saada kohe otse AD muundurist 9 bitine väärtus



Eelmise näite käitumine reaalselt

- Kuna eelmsel skeemil pole näha mitte mingit sisendpingete piiramist, siis võib sisendisse sattuda ka negatiivne pinge, mis diferentsiaal sisendi puhul tekitab negatiivse tulemuse. Kui tegemist on märgita arvudega, siis tegemist väga suure positiivse arvuga.
- Eelmise näite tulemused sõltuvad ka sellest kui suured on takistid R1, R3 ja R5 – piisavalt suurte väärtuste juures võib saada valesid tulemusi

Koodi optimeerimine

- Koodi optimeerimisega tegeleb peamiselt küll kompilaator kuid väga palju annab teha ära ka programmeerija poolt
- Mõned põhilisemad optimeerimise meetodid:
  - Kui võimalik, siis peaks olema tsüklis võrdlustehte nulli võrdlus
  - Korrutamised võiksid võimalusel olla kahe astmetena (2, 4, 8, jne)
  - Kui arvud on esitatud suurte muutujatega, kuid võrreldakse ainult madalamaid bitte, siis tuleks võrreldav enne muutuja *castida* süsteemi enda registri laiuseks muutujaks.
  - Võimalikult vähe kasutada ujuvkoma arve

Mõned märkused 8 bitiste kontrollerite kohta

- Kasutada võimalikult väikeseid muutujatüüpe
  - Muutujatel väärtusega kuni 255 uint8_t, muutujatel väärtusega 256 kuni 65535 uint32_t, jne.
- Kui võimalik, siis kasutada võimalusel märgita tüüpe
- Kui võimalik, siis mitte kasutada *enum* tüüpe, kuna tihtipeale neid muudetakse 16 bitisteks muutujateks
- Kasutada *switch* lausete asemel *if/else* lauseid (juhul kui see koodi loetamatuks ei tee)
- Pidevalt kasutuses olev mälu, näiteks puhvrid, tuleks deklareerida staatilisena

Optimeerimata funktsioon

```
/* mitteoptimaalne */
uint32_t crc32_update (uint32_t crc,
                       const uint8_t data)
{
    uint8_t i;

    crc = crc ^ data;

    for (i = 0; i < 8; i++)
    {
        if (crc & 0x01)
            crc = (crc >> 1) ^ 0xEDB88320;
        else
            crc = crc >> 1;
    };

    return crc;
};
```

Algne funktsioon, kus ei ole mingit optimeerimist tehtud

Optimeeritud funktsioon

```
/* optimaalne */
uint32_t crc32_update (uint32_t crc,
                       const uint8_t data)
{
    uint8_t i = 8;

    crc = crc ^ data;

    for (; i; i--)
    {
        if ((uint8_t)crc & 0x01)
            crc = (crc >> 1) ^ 0xEDB88320;
        else
            crc = crc >> 1;
    };

    return crc;
};
```

Eelnev funktsioon on kirjutatud optimaalsemalt ümber – tsüklis muutuja võrdlus käib nulli suhtes ja CRC võrdlusel kontrollitakse ainult nooremaid bitte

Swithc'i ja if/else

```
#include <stdint.h>

uint8_t test_switch (uint8_t a)
{
    switch (a)
    {
        case 'A':
            return 0;
        case 'B':
            return 1;
        default:
            return 255;
    };
};

/* if/else lausetega sama funktsioon */
uint8_t test_if (uint8_t a)
{
    if (a == 'A')
        return 0;
    else if (a == 'B')
        return 1;
    else
        return 255;
};
```

8 bitistel kontrolleritel tehakse mõnikord switch'i operatsioonid 16 bitise muutujaga, mis omakorda tekitab suurema koodi.

NB! Uuemad GCC kompilaatorid suudavad mõlemad näited niimoodi ära optimeerida, et ei ole suuruse vahet

Koodi staatiline kontroll

- Kompilaatori poolne
  - Wall ja Wextra võtmetega
- Väliste programmidega
  - Splint (lint'i edasiarendus)
  - Frama-C
  - Blast (Berkeley Lazy Abstraction Software Verification Tool)
  - Sparse (Linuxi kerneli koodi staatiline kontroll)
  - Clang (C, C++ kompilaatori front end)

Kompilaatori poolne kontroll

- Kompilaatori poolne programmi kontroll võimaldab avastada enamuse vigu mis programmerimise käigus on tehtud.
  - Missiooni ja ohutuskriitilisi programme ei ole mõtet ilma kompilaatori poolse kontrolli lubamiseta teha (seda lisaks välistele kontrollidele)
- GCC'l on mõtekas sisse lülitada nii `Wextra` kui ka `Wall` kontrollid
  - Need kontrollid küll ei katkesta (enamusjaolt) kompileerimist, kuid lisavad üpriski palju vigadest teavitamise infot. Et nende kontrollidega katkestada kompileerimist tuleks lisada veel ka Werror võti

Koodi kontroll Splintiga

- Toimib sarnaselt kompilaatori eelprotsessorile, on tunduvalt põhjalikum, kuid:
  - Splint teeb ainult staatilist koodi kontrolli ja väljastab teateid võimalike ohtklike kohtade kohta koodis
  - Splint on võimeline leidma ainult koodisiseseid ebakooskõlasid
- Splinti peamisteks puudusteks on:
  - Splint ei ole mõeldud sardsüsteemidele ning võib tekitada sellega suurt segadust
  - Tavaliste testide ajal suhteliselt suur veateadete arv, mis omakorda nõuab iteratiivset splinti kasutamist
  - Splintis on mitmeid omavahel vastuolulisi teste

Kodeerimis stiil/standard

- Ühe projekti raames tuleks jääda kindlalt mingi kodeerimis stiili või standardi juurde
  - Kõige hullem asi mis saab ühes projektis olla on see kui on terve projekti jooksul muudetud mitu korda stiili, selline tegevus raskendab oluliselt programmi lugemist ning võib tekitada ka vigu
- Kui ei ole veel mingit välja kujunenud stiili, siis on mõtekas valida mõni tuntud stiilidest
  - Allman (ANSI)
  - K & R

Kokkuvõte

- Tüübi muutmine juures tuleb jälgida suuremast väiksemasse tüüpi muutused
- Tsüklite kasutamisel peab olema kindel, et tsüklil on lõpu tingimus või siis tsükkel ei võtaks kogu ressurssi endale
- Sisendandmeid tuleb (peaaegu) alati kontrollida
- Ühe projekti raames peab olema kindel kodeerimise stiil