

Securitate Software

III. Vulnerabilități specifice limbajului C

Objective

- Prezentarea principalelor aspecte legate de limbajul C: tipuri de date, reprezentarea numerelor, conversii;
- Prezentarea vulnerabilităților ce apar datorită neînțelegerii limbajului de programare C.

Continut

1 Introducere

2 Limbajul C, reprezentarea datelor

3 Conditii limita

- Depasire pentru intregi fara semn
- Depasire pentru intregi cu semn

4 Conversii de tip

- Definitii
- Vulnerabilitati

5 Concluzii

Sumar

- subiect vechi de cercetare în domeniul securității;
- principala cauză a numeroase probleme raportate;
- problema se datorează spațiului limitat de reprezentare pentru numere;
- de la limbaj la limbaj apar variațiuni ale problemelor.

Referințe CWE

- CWE-682: Incorrect Calculation
- CWE-190: Integer Overflow or Wraparound (locul 24 în Mitre top 25)
- CWE-191: Integer Underflow (Wrap or Wraparound)
- CWE-192: Integer Coercion Error

CWE-682: Incorrect Calculation

- `int *p = x;`
`char * second_char = (char *)(p + 1);`
- `img_t table_ptr;`
`/*struct containing img data, 10kB each*/`
`int num_imgs;`
`...`
`num_imgs = get_num_imgs();`
`table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);`
`...`

CWE-190: Integer Overflow or Wraparound

- ```
nresp = packet_get_int();
if (nresp > 0) {
 response = xmalloc(nresp*sizeof(char*));
 for (i = 0; i < nresp; i++) response[i] = packet_ge
}
```
- ```
short int bytesRec = 0;  
char buf[SOMEBIGNUM];  
  
while(bytesRec < MAXGET) {  
    bytesRec += getFromInput(buf+bytesRec);  
}
```

CWE-191: Integer Underflow

```
#include <stdio.h>
#include <stdbool.h>
main (void)
{
    int i;
    i = -2147483648;
    i = i - 1;
    return 0;
}
```


CWE-192: Integer Coercion Error

```

DataPacket *packet;
int numHeaders;
PacketHeader *headers;

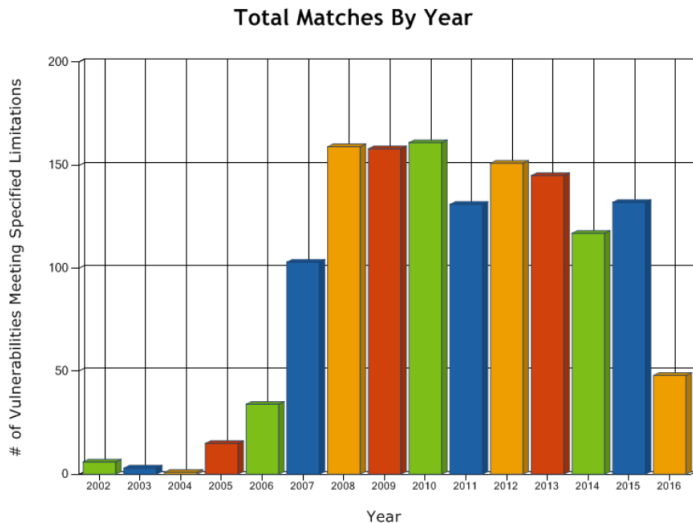
sock=AcceptSocketConnection();
ReadPacket(packet, sock);
numHeaders =packet->headers;

if (numHeaders > 100) {
    ExitError("too_many_headers!");
}
headers = malloc(numHeaders * sizeof(PacketHeader));
ParsePacketHeaders(packet, headers);
```

Limbaje afectate

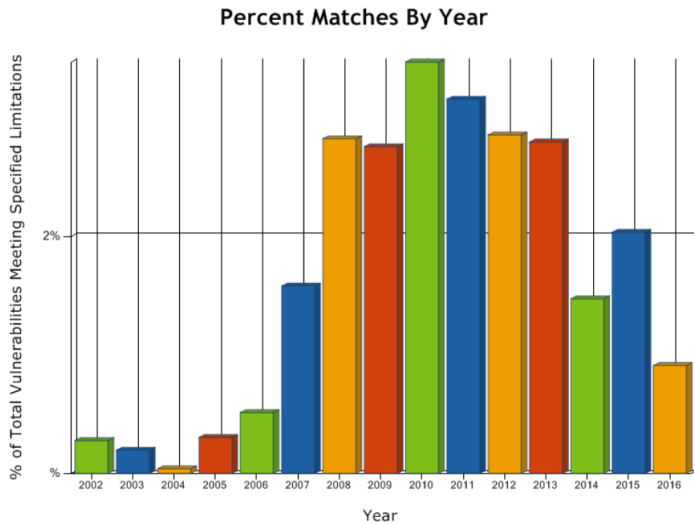
- toate limbajele pot fi afectate
 - ▶ efectul depinde de felul în care limbajul tratează valorile întregi
- C și C++ sunt cele mai vulnerabile
 - ▶ cel mai probabil o eroare *integer overflow* poate fi transformată într-un atac de tipul *buffer overflow*
- toate limbajele sunt susceptibile la atacuri *DoS* și *erori logice*

Statistică vulnerabilități *integer overflow*



sursa: <https://nvd.nist.gov/vuln/search>

Statistică vulnerabilități *integer overflow* (II)



sursa: <https://nvd.nist.gov/vuln/search>

Tipuri de date

- cu / fără semn
 - ▶ precizie
 - ▶ specificator: *signed*
- tipuri de bază
 - ▶ caracter: *char*, *signed char*, *unsigned char*
 - ▶ întreg (cu / fără semn)
 - ★ *short int* / *unsigned short int*
 - ★ *int* / *unsigned int*
 - ★ *long int* / *unsigned long int*
 - ★ *long long int* / *unsigned long long int*
 - ▶ virgulă mobilă: *float*, *double*, *long double*
 - ▶ *bit fields*
- alias
 - ▶ UNIX: *int8_t* / *uint8_t*, *int16_t* / *uint16_t*, *int32_t* / *uint32_t*, *int64_t* / *uint64_t*
 - ▶ WINDOWS: *BYTE* / *CHAR*, *WORD*, *DWORD*, *QWORD*

Dimensiune, valori minime și maxime

Tip	Dim	val. minimă	val. maximă
signed char	8	-128	127
unsigned char	8	0	255
short	16	-32,768	32,767
unsigned short	16	0	65,535
int	32	-2,147,483,648	2,147,483,647
unsigned int	32	0	4,294,967,295
long	32	-2,147,483,648	2,147,483,647
unsigned long	32	0	4,294,967,295
long long	64	-9,223,372,036,845,775,808	9,223,372,036,854,775,807
unsigned long long	64	0	18,446,744,073,709,551,615

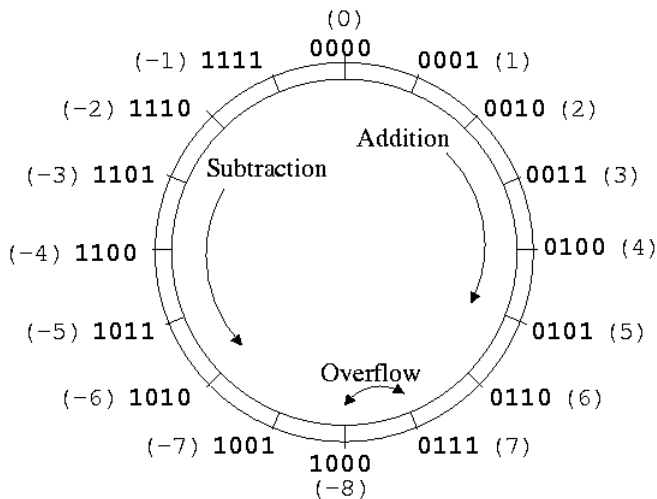
Regula generala: n biti

$$\begin{array}{ll} [0, 2^n - 1] & , \text{fără semn} \\ [-2^{n-1}, 2^{n-1} - 1] & , \text{cu semn} \end{array}$$

Codificarea binară

- biți: 0 și 1
- interpretarea cu semn folosește bitul cel mai semnificativ ca și bit de semn ($0 \implies \geq 0, 1 \implies < 0$)
- reprezentari:
 - ▶ bit de semn (*sign and magnitude*), avantaj - ușor pentru oameni, dezavantaj - dificil de implementat
 - ▶ complement față de unu
 - ★ numere negative: complementez toti biții
 - ★ avantaj: ușor pentru CPU
 - ★ dezavantaj: două valori pentru zero, tratarea eventualului transport / împrumut
 - ▶ complement față de doi
 - ★ numere negative: complementez toți biții și adaug 1
 - ★ operațiile la nivel de bit pentru adunare și scădere se fac la fel pentru numere pozitive și negative
 - ★ o singură valoare pentru zero (0)

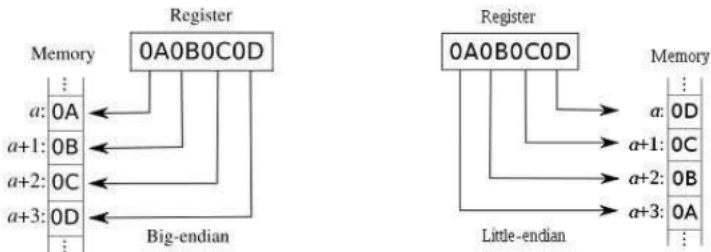
Complement față de doi



Datele în memorie

- **big endian**: cel mai semnificativ octet la adresa mică
- **little endian**: cel mai semnificativ octet la adresa mare

Big Endian vs. Little Endian



Standarde de implementare

- ILP32: *integer, long, pointer* reprezentat pe 32 de biți
- **ILP32LL**
 - ▶ *integer, long, pointer* reprezentați pe 32 de biți, *long long* pe 64 de biți
 - ▶ standardul pentru platformele pe 32 de biți
- **LP64**
 - ▶ *long* și *pointer* reprezentați pe 64 de biți
 - ▶ standardul pentru platformele pe 64 de biți
- ILP64: *integer, long, pointer* reprezentați pe 64 de biți
- LLP64: *long long* și *pointer* reprezentați pe 64 de biți

Condiții limita - Context și definiții

- valorile limită (min și max)
- dependent de reprezentarea binară
- condiția de depășire peste valoarea maximă (numeric / integer overflow)
 - ▶ valoarea maximă ce poate fi reprezentată pe un întreg este depășită
 - ▶ exemplu:

```
unsigned int a;  
a = 0xFFFFFFFF;  
a = a + 1; // -> a = 0
```

- condiția de depășire sub valoarea limită (numeric / integer underflow)
 - ▶ valoarea minimă reprezentată pe un întreg este depășită
 - ▶ exemplu:

```
unsigned int a;  
a = 0;  
a = a - 1; // -> a = 0xFFFFFFFF
```

Riscuri din p.d.v. al securității pentru depășire (overflow/underflow)

- poate modifica în mod greșit valoarea variabilelor
 - ▶ comportament imprevizibil al aplicației
 - ▶ integritatea aplicației este încălcată
- poate duce la o avalanșă de greșeli
- atacatorul are o plajă mare de posibilități pentru a influența comportamentul aplicației
- vulnerabilitățile se datorează operațiilor aritmetice ce folosesc date **controlate** de utilizator (direct sau indirect)
- exemple
 - ▶ calcul greșit de lungime /limită pentru alocare de memorie \implies *buffer overflow*
 - ▶ verificarea greșită a lungimii / limitei \implies *buffer overflow*

Unsigned Integer Overflow

- operațiile sunt supuse următoarelor reguli (*modular arithmetic*):
 - ▶ rezultatul operației este "rezultatul real" modulo (numarul maxim ce poate fi reprezentat + 1)
 - ▶ ex. $R = R \% 2^{32}$
- rezultatul este trunchiat
- operații ce ar putea duce la depășire: adunare, înmulțire, deplasare la stânga (*shift*)
- la nivelul procesorului, flag-ul CF (carry flag) și OF (overflow flag) este setat

```
unsigned int a;  
a = 0xE0000020;  
a = a + 0x20000020;  
// -> a = (0xE0000020 + 0x20000020) % 0x10000000  
// a = 0x40
```

Exemplu

```
u_char *make_table(unsigned int width, unsigned int height,
    unsigned int n;
    int i;
    u_char *buf;

    n = height * width; // !!
    buf = (char*) malloc(n); // !!

    if (!buf)
        return NULL;
    for (i = 0; i < height; i++)
        memcpy(&buf[i * width], init_row, width);
}
```

Exemplu (II)

- n poate să depășească domeniul de reprezentare datorită înmulțirii a două numere controlate de utilizator (*width* și *height*), rezultând un număr relativ mic
 - ▶ exemplu (pe 32 de biți)
 - ★ $0x400 * 0x10000001 - 0x400$ (hexazecimal)
 - ★ $1024 * 268435457 = 1024$
- **dar**, bucla *for* parcurge zona de memorie
 - ▶ în cazul exemplului nostru
 - ★ sunt alocăți 1024 de octeți \implies **alocat un element**
 - ★ dar accesăm mai mult de un element

Vulnerabilitate Unsigned Integer Overflow în OpenSSH 3.1

```
u_int nresp; // valoare controlata de utilizator
nresp = packet_get_int(); // cate raspunsuri se asteapta
if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char*));
    for (i=0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
packet_check_eom();
```


Vulnerabilitate Unsigned Integer Overflow în OpenSSH 3.1 (II)

- variabila *nresp* nu este verificată, valoarea variabilei este setată pe baza datelor primite de la utilizator
- pe platforma *x86* *nresp* este *unsigned int* pe 4 octeți
- $UINT_MAX = 0xFFFFFFFF$
- dimensiunea unui pointer este 4 octeți
- *overflow* când $nresp \geq 0xFFFFFFFF / 4$ ($0x40000000$)
- ex.
 - ▶ $nresp = 0x40000001$,
 $nresp * sizeof(char*) = 0x100000004 = 0x00000004$
 - ▶ *xmalloc* alocă doar 4 octeți
 - ▶ for pe *nresp*

Unsigned Integer Underflow

- cauza: operație a carui rezultat este sub valoarea minima reprezentabilă (0)
- rezultat: numere pozitive mari
- operații care duc la *overflow*: scaderi

Signed Integer overflow and underflow

- depășirea *overflow* poate duce la un număr mare — $>$ număr negativ (datorită complementului față de doi)
- depășirea *underflow* transformă un număr negativ într-un număr pozitiv
- operații ce pot duce la depășire: adunare, înmulțire, deplasare la stânga (shift)
- depinde de cum se modifică bitul de semn

Exemplu

```
char* read_data(int sockfd) {
    char *buf;
    int value;
    int length = network_get_int(sockfd); ///!!

    if (!(buf = (char*) malloc(MAXCHARS)))
        die("malloc");
    if (length < 0 || length + 1 > MAXCHARS) {
        ///!! ambele teste trec pentru length = 0x7FFFFFFF
        free(buf);
        die("bad_length");
    }

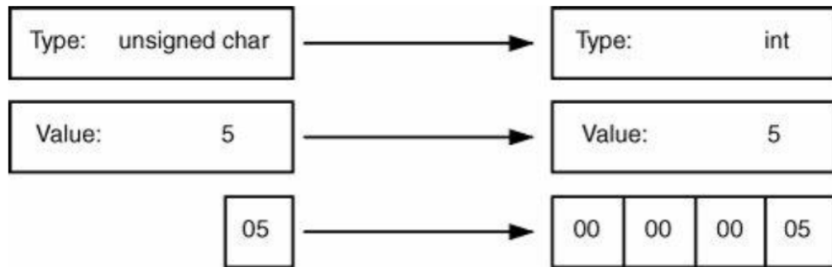
    if (read(sockfd, buf, length) <= 0) {
        free(buf);
        die("read");
    }

    buf[value] = '\0';
    return buf;
}
```

Conversii de tip, definiții și context

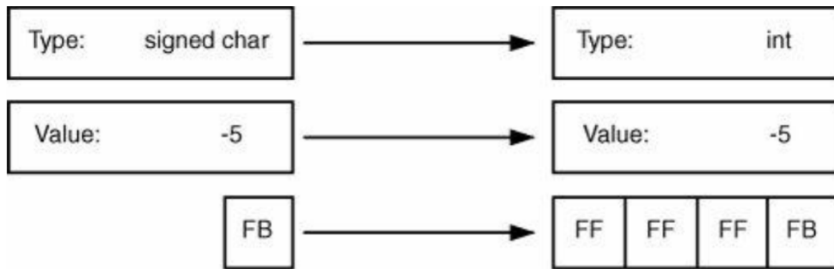
- conversia de la un tip de dată la alt tip de dată
- tipuri de conversii: **explicit**, **implicit**
- pastrarea valorii vs. schimbarea valorii
 - ▶ noul tip poate sau nu poate reprezenta tot intervalul de valori pentru vechiul tip
- cazuri:
 - 1 **lărgire (widening)**
 - ★ **zero-extension**: pentru numere în interpretarea fără semn
 - ★ **sign-extension**: pentru numere în interpretarea cu semn
 - 2 **îngustare** prin trunchiere
 - ★ schimbă valoarea
 - 3 **conversie** între numere cu / fără semn
 - ★ schimbă valoarea

Conversie prin lărgire



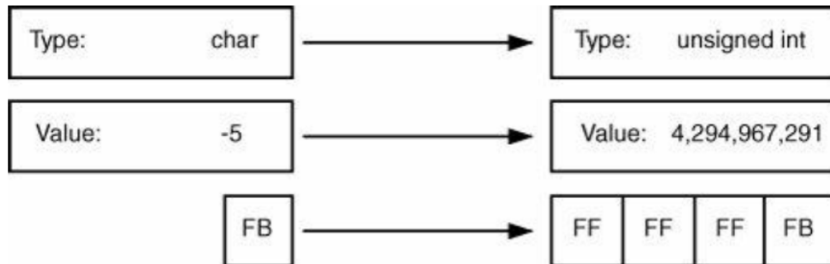
Conversie cu păstrarea valorii, *unsigned char* — > *signed int*

Conversie prin lărgire (II)



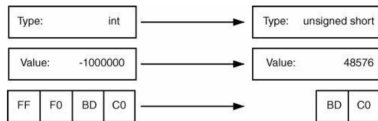
Conversie cu păstrarea valorii, *signed char* \rightarrow *signed int*

Conversie prin lărgire (III)

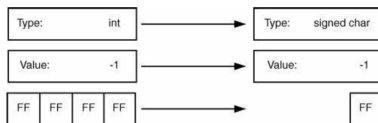


Conversie cu schimbarea valorii, *signed char* \rightarrow *unsigned int*

Conversie prin îngustare

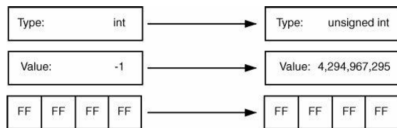


(a) *signed int* – > *unsigned short*

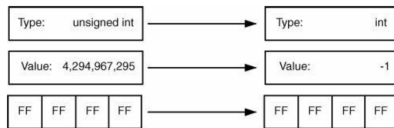


(b) *signed int* – > *signed char*

Conversie între numere cu / fără semn



(a) *signed int* – > *unsigned int*



(b) *unsigned int* – > *signed int*

Reguli de conversie pentru date de tip *intreg*

Conversie prin **lărgire**:

- *signed* – > *unsigned*
 - ▶ extind bitul de semn \implies **schimbă valoarea**
- *signed* – > *signed*
 - ▶ extind bitul de semn \implies valoarea se păstrează
- *unsigned* – > *orice*
 - ▶ extind 0 \implies valoarea se păstrează

Conversie prin **îngustare**:

- *orice* – > *orice*
 - ▶ trunchiere \implies **schimbă valoarea**

Conversie între numere **cu** / **fără** semn:

- *signed* – > *unsigned* (de același tip)
 - ▶ biții se păstrează dar valoarea este interpretată diferit \implies **schimbă valoarea**

Conversii simple

- (type)casts:

```
( unsigned char ) var
```

- asignare

```
short int v1;  
int v2 = -10;  
v1 = v2;
```

- apel funcție, pe baza prototip

```
int dostuff( int x, unsigned char y );
```

```
void func( void ) {  
    char a=42;  
    unsigned short b=43;  
    long long int c;  
    c=dostuff( a, b );  
}
```

Conversii simple (II)

- apel funcție, valoare retur

```
char func(void) {  
    int a=42;  
    return a;  
}
```

Integer Promotions (conversie prin lărgire la int)

- valori întregi de dimensiuni mici — $> int$
 - ▶ când anumite operații necesită un operand întreg
- clasificare / rang:
 - 1 long long int, unsigned long long int
 - 2 long int, unsigned long int
 - 3 signed int, int
 - 4 unsigned short, short
 - 5 char, unsigned char, signed char
- oriunde poate fi folosit un *int* sau *unsigned int*, poate fi folosit un tip întreg de rang mai mic
- când tipul variabilei este mai mare ("lat") decât *int*, conversia nu modifică

Integer Promotions (conversie prin lărgire la int) (II)

- când tipul variabilei este mai "îngust" decât *int*
 - ▶ se încurajează transformarea dacă se păstrează valoarea prin transformare la *int*
 - ▶ altfel se face o conversie la *unsigned int*

Integer Promotions

- operatorul unar $+$ aplică *integer promotion* asupra operandului
- operatorul unar $-$ aplică *integer promotion* asupra operandului apoi neagă
 - ▶ indiferent dacă operandul este cu semn după conversie, se aplică regula complementului față de 2
 - ▶ complementul față de doi al lui $0x80000000$ este tot $0x80000000$
 - ▶ cod vulnerabil

```
int bank1[1000], bank2[1000];
void hashbank (int index, int value) {
    int *bank = bank1;
    if (index < 0) {
        bank = bank2;
        index = -index;
    }
    // scrie la bank2[-648]
    // pentru index = 0x80000000
    bank[index % 1000] = value;
}
```


Integer Promotions (II)

- operatorul unar `~` aplică *integer promotion* asupra operandului apoi complementul față de unu
- operatorul shift la nivel de bit
 - ▶ aplică *integer promotion* asupra celor două argumente
 - ▶ tipul rezultatului este același cu tipul arg din stânga schimbat

```
char a = 1;  
char c = 16;  
int bob;  
bob = a << c;
```

- ▶ *switch* aplică *integer promotion*

Conversii în operații aritmetice

- conversii în evaluarea unor expresii C în care argumentele sunt de tipuri diferite
- argumentele trebuie aduse la un tip de data compatibil

Conversii în operații aritmetice - regula 1

- au prioritate numerele în virgula mobilă
- dacă un argument este reprezentat în virgula mobilă \implies celălalt argument este adus în reprezentarea virgula mobilă
- dacă ambele argumente sunt reprezentate în virgulă mobilă dar unul dintre ele are o precizie mai mică \implies se aduce la tipul celui de precizie mai mare

Conversii în operații aritmetice - regula 2

- dacă nu sunt argumente reprezentate în virgulă mobilă \implies se aplică regula *integer promotion*
 - ▶ dacă e nevoie, toți operanzii sunt convertiți la tipul întreg
 - ▶ exemplu (comparația OK, chiar dacă pare să fie overflow)

```
unsigned char term1 = 255;  
unsigned char term2 = 255;  
if ((term1 + term2) > 300)  
do_something();
```

Conversii în operații aritmetice - regula 2 (II)

- (funcția *do_something()* se va apela)

```
unsigned short a = 1;  
if ((a - 5) < 0)  
    do_something();
```

- (funcția *do_something()* nu se va apela)

```
unsigned short a = 1;  
a = a - 5;  
if (a < 0)  
    do_something();
```

Conversii în operații aritmetice - regula 3

- același tip după *integer promotion*
 - ▶ dacă după conversie ambii operanzi sunt de același tip, nu se mai aplică alte conversii

Conversii în operații aritmetice - regula 4

- același semn, tipuri diferite

- ▶ dacă după conversie ambii operanzi au același semn, dar dimensiune diferită
- ▶ \implies se aplică o conversie prin lărgire
- ▶ exemplu (OK)

```
int t1 = 5;  
long int t2 = 6;  
long long int res;  
res = t1 + t2;
```

Conversii în operații aritmetice - regula 5

- variabila fără semn reprezentată pe mai mulți biți decât variabila cu semn
 - ▶ variabila cu semn este convertită la dimensiunea celei fără semn
 - ▶ exemplu gresit (funcția *do_something()* nu se va apela)

```
int t = -5;  
if (t < sizeof(int)) // i.e. "4294967291 < 4"  
    do_something();
```


Conversii în operații aritmetice - regula 6

- variabila în interpretare cu semn este reprezentată pe mai mulți biți decât variabila în interpretare fără semn, prin conversie se păstrează valoarea
 - ▶ *unsigned* convertit la tipul *signed*
 - ▶ exemplu corect

```
long long int a = 10;  
unsigned int b = 5;  
(a+b);
```

Conversii în operații aritmetice - regula 7

- variabila în interpretare cu semn este reprezentată pe mai mulți biți decât variabila în interpretare fără semn, prin conversie nu se poate păstra valoarea
 - ▶ când variabila *unsigned* de dimensiune mai mică nu poate fi reprezentată de tipul mai mare *signed*, ambele sunt convertite la tipul *unsigned* corespunzător tipului *signed*
 - ▶ exemplu (se presupune ca *int* și *long int* au aceeași dimensiune)

```
unsigned int a = 10;  
long int b = 20;  
(a+b); // rezultatul este de tipul "unsigned long"
```

Conversii în operații aritmetice - aplicare

- adunare
- scădere
- operatori de înmulțire
- operatori relaționali și de egalitate
- operatori la nivel de bit
- operatorul ?

Conversii cu / fără semn

- ex 1 - vulnerabil

```
int copy (char *dst, char *src, unsigned int len) {  
    while (len--)  
        *dst++ = *src++;  
}  
int f = -1;  
copy (d, s, f);
```

Conversii cu / fără semn

- ex 1 - vulnerabil

```
int copy (char *dst, char *src, unsigned int len) {  
    while (len --)  
        *dst++ = *src++;  
}  
int f = -1;  
copy (d, s, f);
```

vulnerabil deoarece:

- **nu se validează** valoarea variabilei *f*
- *signed f* este convertit la *unsigned int* — > *buffer overflow*

Concluzie

- nu lăsați numere negative (*signed int*) să se propage în funcții libc care folosesc *size_t*, *size_t* este de tipul *unsigned int*
- exemple de astfel de funcții: *read*, *sprintf*, *strncpy*, *memcpy*, *strncat*, *malloc*

Conversii cu / fără semn

- ex 2 - vulnerabil

```
int len , sockfd , n;  
char buf[1024];  
len = get_user_len(sockfd);  
if (len < 1024)  
    read (sockfd , buffer , len );
```

Conversii cu / fără semn

- ex 2 - vulnerabil

```
int len , sockfd , n;  
char buf[1024];  
len = get_user_len(sockfd);  
if (len < 1024)  
    read (sockfd , buffer , len ); // len convertit la "unsigned"
```

vulnerabil deoarece:

- *len* este **validat greșit**
- când *len* are o valoare negativă — $>$ *buffer overflow*

Concluzie

- **nu folosiți variabile în interpretare cu semn (signed) pentru dimensiune**
- dacă folosiți variabile cu semn, **verificați dacă valoarea e pozitivă**, pe lângă verificarea limitei intervalului

Extensia semnului

- în unele cazuri extensia semnului poate avea consecințe neprevăzute
 - ▶ când se face conversia de la un număr în interpretare cu semn pe o dimensiune mai mică la un număr în interpretare fără semn pe o dimensiune mai mare
- exemplu de cod vulnerabil (atât varianta inițială cât și varianta nouă)

```
char len;  
len = get_len();  
// snprintf(dst, len, "%s", src);  
// initial: probleme pentru len negativ  
snprintf(dst, (unsigned int)len, "%s", src);  
// solutie: probleme datorita extensiei de semn
```


Extensia semnului (II)

- nu uitați: *char* și *short* sunt tipuri de date în interpretare cu semn
- exemplu cod vulnerabil, de ce?

```
char *indx;
int count;
char nameStr[MAX_LEN];
...
memset(nameStr, 0, sizeof(nameStr));
...
indx = (char*) (pkt + tt_offset);
count = (char) *indx;
while (count) {
    (char*)indx++;
    strncat(nameStr, (char*)indx, count);
    indx += count;
    count = (char) *indx;
    strncat (nameStr, ".", sizeof(nameStr) - strlen(nameStr));
}
nameStr[strlen(nameStr)-1] = 0;
```

Extensia semnului (II)

- nu se verifică limita superioară pentru *count*

```
char *indx;
int count;
char nameStr[MAX_LEN]; // 256
...
memset(nameStr, 0, sizeof(nameStr));
...
indx = (char*) (pkt + tt_offset);
count = (char) *indx;
while (count) {
    (char*)indx++;
    strncat(nameStr, (char*)indx, count);
    indx += count;
    count = (char) *indx;
    strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr))
}
nameStr[strlen(nameStr)-1] = 0;
```

Extensia semnului (III)

- exemplu cod vulnerabil, de ce?

```
...
while (count) {
    if ((unsigned int)strlen(nameStr) + (unsigned int) count <
        (MAX_LEN - 1)) {
        ...
        strncat(nameStr, (char*)indx, count);
        ...
    }
}
nameStr[strlen(nameStr)-1] = 0;
```

Extensia semnului (III)

- exemplu cod vulnerabil
- conversii inutile

```
...
while (count) {
    if ((unsigned int)strlen(nameStr) + (unsigned int) count <
        (MAX_LEN - 1)) { // trece pentru 5 + (-1),
                        // datorita depasirii
        ...
        strncat(nameStr, (char*)indx, count);
    }
}
nameStr[strlen(nameStr)-1] = 0;
```

Extensia semnului (IV)

- exemplu cod vulnerabil datorită conversiei explicite (*char typecast*)

```
unsigned char *indx;
unsigned int count;
unsigned char nameStr[MAX_LEN];
...
indx = (char*) (pkt + tt_offset);
count = (char) *indx; // inca vulnerabil pentru numere negative
while (count) {
    if (strlen(nameStr) + count < (MAX_LEN - 1)) { // nu trece
        //cand strlen() este 0
        indx++;
        strncat(nameStr, indx, count);
        indx += count;
        count = *indx;
        strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
    } else { die("error"); }
}
nameStr[strlen(nameStr)-1] = 0; // scrie la nameStr[-1]
```

Extensia semnului (V)

- extensia semnului

- ▶ cod C

```
// caz 1
unsigned int no;
char c=5;
no = c;
```

- ▶ cod asamblare

```
mov [ebp+var_5], 5
movsx eax, [ebp+var_5]

mov [ebp+var_4], eax
```

Extensia semnului (V) - caz 2

- extensia semnului

- ▶ cod C

```
// caz 2
unsigned int no;
unsigned char c=5;
no = c;
```

- ▶ cod asamblare

```
mov [ebp+var_5], 5
xor eax, eax
mov al, [ebp+var_5]
mov [ebp+var_4], eax
```

- **hint audit:** căutați instrucțiuni *movsx*

Trunchiere

- un tip de dimensiune mare convertit la un tip de dimensiune mai mică în urma unei semnări
- exemplu

```
int g = 0x12345678;  
short int h;  
h = g; // h = 0x5678;
```


Trunchiere II

- exemplu, *size_t* trunchiat la *short int*

```
unsigned short int f;  
char mybuf[1024];  
char *userstr = getuserstr();  
f = strlen(userstr); // f = 464 pentru strlen 66,000  
if (f < sizeof(mybuf) - 5) // trece pentru strlen 66,000  
    die ("string_too_long");  
strcpy(mybuf, userstr);
```

- depășirile duc la un comportament neprevăzut al aplicației
- adesea duc la vulnerabilități de tipul *buffer overflow*
- limbajele C/C++ cele mai afectate

Recomandări

- verificați toate datele controlate de utilizator înainte de folosirea lor
- verificați aritmetica ce implică date de la utilizator
- nu folosiți *signed integer* ca și parametrii *unsigned*
- scrieți cod clar, nu folosiți "trucuri"
- comentați codul cu eventualele conversii ce se fac implicit în cazul unor operații
- activați opțiunile compilatorului ce vă ajută la identificarea acestor vulnerabilități
 - ▶ Visual Studio: -W4
 - ▶ gcc: -Wall, -Wsign-compare, -ftrapv

Recomandări pentru code audit

- monitorizați toate datele de intrare
- verificați codul ce scrie în zone de memorie
- uitați-vă după conversii explicite
- verificați aritmetica ce implică date de la utilizator
- folosiți unelte de analiză

Bibliografie

- "The art of Software Security Assessments", chapter 6, "C Language Issues", pp. 203-296
- "The 24 Deadly Sins of Software Security", Sin 7. Integer Overflows, pp. 119 – 142