Université Grenoble Alpes **Grenoble INP**
UFR IM2AG **ENSIMAG**

Master 2 CyberSecurity
Software Security and Secure Programming

Exercices on Access Control and Information Flow

Exercise 1

Let consider the following code, where security classes are ordered S > C > U (constant values being in class U):

```
x : integer class S;
y,z : integer class C;
t : integer class U;

y := 2; z:= 3;
x := y+z ;
if ( y<5 ) then
      t := 4;
else
      t := 3;
```

We require that a user of given security class should not get access to information belonging to a higher class.

Q1. Is this program correct for a user of class C ?

Q2. And for a user of class U ?

Answers

Q1. We want to check that there is no information-flow from S values to C or U data.

```
In this code, variable x (of class S) is never used, so it never flows to a
variable of lower class.
```

Q1. We want to check that there is no information-flow from S or C values to U data.

```
In this code, variable y (of class C) is used in the condition of the if
statement. Hence its value implicitely flows to variable t (of class U)
conditionally assigned. Confidentiality of C values is therefore not guaranteed
with respect to U users.
```

Exercise 2

Assuming parameters n and k are "high" (confidential), is this function potentially leaking information ? And if yes, where and how ?

```
int crypto_secretbox_open
   (unsigned char *m,  const unsigned char *c,
           unsigned long long clen,
    const unsigned char *n,  const unsigned char *k)
{
  int i;
  unsigned char subkey [32];

  if (clen < 32) return -1;

  subkey = crypto_stream_salsa20(32,n,k);

  if (crypto_auth_hmacsha512_verify(c,c+32,clen -32, subkey)!=0)
        return -1;
  crypto_stream_salsa20_xor(m,c,clen ,n,k);

  for (i = 0;i < 32;++i)
        m[i] = 0;

  return 0;
}
```

Answers

Assuming we want to keep confidential the **values** *n and *k

```
int crypto_secretbox_open
   (unsigned char *m,  const unsigned char *c,
           unsigned long long clen,
    const unsigned char *n,  const unsigned char *k)
{
  int i;
  unsigned char subkey [32];

  if (clen < 32) return -1;   // clen is low, NO PROBLEM ...

  subkey = crypto_stream_salsa20(32,n,k); // subkey may become HIGH ...

  if (crypto_auth_hmacsha512_verify(c,c+32,clen -32, subkey)!=0)
        return -1;   // PB ! (gives info about subkey)

  crypto_stream_salsa20_xor(m,c,clen ,n,k); // *m may become HIGH

  for (i = 0;i < 32;++i)
        m[i] = 0;   // PB ! (out-of-bound access -> size of m)

  return 0;
}
```

Exercise 3

We consider the following function:

```
void buildfname ( char *gecos , char *login , char * buf)
{
   char *p;
   char *bp = buf ;

   for (p = gecos ; *p != '\0 ' && *p != ',' && *p != ';' && *p != '%'; p ++){
   if (*p == '&') {
                 strcpy (bp , login );
                 *bp = toupper (* bp );
                 while (* bp != '\0 ')
                       bp ++;
          } else {
                 bp ++;
                 *bp = *p;
          }
   }
   *bp = '\0 ';
}
```

The objective is to identify vulnerable statement able to write *untrusted* (i.e. user controlled) values into memory. We use the following notation:

- a value is said **tainted** (T) if it depends on a user input;
- it is said **untainted** (U) otherwise.

Q0. Explain why/how this *taint analysis* problem is related to *non-interference* ?

Q1. Which instructions perform **memory write** operations (i.e, are potentially vulnerable) ?

Q2. Assuming both parameters `gecos` and `login` are tainted, how does this taint propagate to potentially vulnerable instructions ?

Q3. Same question if only `gecos` is tainted

Q4. Same question if only `login` is tainted

Answer

Q0.

Taint analysis aims to track if input (attacker-controlled) values may flow to vulnerable statements . In non-interference we want to check whether low and high data are used consistently with respect to confidentiality or integrity properties.

Both analyis are based on tracking data and control-flow dependencies, but :

- in non-interference, variables labels (low/high) are fixed
- in taint analysis, taint labels are propagated through assigments :

Both analysis can be performed using similar (static or dynamic) techniques.

Q1. lines 9, 10, 15, 18 corespond to memory writes.

Q2. function buildfname uses 3 buffers : gecos, login and buf. Only buffer **buf** is concerned by write accesses, through pointer **bp**. We want to check when such a **write access** may become **vulnerable** (i.e, potentially leading to an **invalid** memory write) in a way which is controlled by the user (i.e., through a *tainted* data). This situation may occur either if bp becomes too large or negative, or if login is too long. In the codes below taint propagation is shown in blue.

`case 1 : both gecos and login are tainted.`

```
1 void buildfname ( char *gecos , char *login , char * buf)
2 {
3    char *p;
4    char *bp = buf ;
5
6    for (p = gecos ; *p != '\0 ' && *p != ',' && *p != ';' && *p != '%'; p ++){
7    if (*p == '&') {
9                 strcpy (bp , login );  // BAD: potential buffer overflow
10                *bp = toupper (*bp );  // BAD: potential buffer overflow
11                while (*bp != '\0 ')
12                      bp++;
13          } else {
14                bp++;
15                *bp = *p;  // BAD: potential buffer overflow
16          }
17   }
18   *bp = '\0 ';  // BAD: potential buffer overflow
19 }
```

Q3. case 2 : only gecos is tainted

```
1 void buildfname ( char *gecos , char *login , char * buf)
2 {
3    char *p;
4    char *bp = buf ;
5
6    for (p = gecos ; *p != '\0 ' && *p != ',' && *p != ';' && *p != '%'; p ++){
7    if (*p == '&') {
9                 strcpy (bp , login );  // BAD
10                *bp = toupper (*bp );
11                while (*bp != '\0 ')
12                      bp++;
13          } else {
14                bp++;
15                *bp = *p;  // BAD
16          }
17   }
18   *bp = '\0 ';  // BAD
19 }
```

Q4. case 3: only login is tainted

```
void buildfname ( char *gecos , char *login , char * buf)
{
   char *p;
   char *bp = buf ;

   for (p = gecos ; *p != '\0 ' && *p != ',' && *p != ';' && *p != '%'; p ++){
   if (*p == '&') {
               strcpy (bp , login );//BAD: potential BoF if login is too long
               *bp = toupper (*bp );
               while (*bp != '\0 ')
                     bp++;
         } else {
               bp++;
               *bp = *p;
         }
   }
   *bp = '\0 ';
}
```

Exercise 4

We consider the following piece of code, assuming that variable x0 is a **tainted** data and f() is a “dangerous” function which should not be called with a tainted argument.

```
while (i < 10) {
        x3 = x2 ;
        x2 = x1 ;
        x1 = x0 ;
        i = i+1 ;
} ;
f (x3)
```

Discuss for which initial values of i this code is dangerous or not …

Answers

```
while (i < 10) {
        x3 = x2 ;
        x2 = x1 ;
        x1 = x0 ;
        i = i+1 ;
} ;
f (x3)
```

on 1 iteration, x1 becomes tainted by x0
on 2 iterations, x2 becomes tainted by x1
on 3 iterations, x3 becomes tainted by x2, hence calling f() become dangerous.
Consequently this function is insecure when the initial value of i is less or equal than 7 ...