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.`

```
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 buffer overflow
                *bp = toupper (*bp );  // BAD: potential buffer overflow
                while (*bp != '\0 ')
                      bp++;
          } else {
                bp++;
                *bp = *p;  // BAD: potential buffer overflow
          }
   }
   *bp = '\0 ';  // BAD: potential buffer overflow
}
```

Q3. case 2 : only gecos 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
                *bp = toupper (*bp );
                while (*bp != '\0 ')
                      bp++;
          } else {
                bp++;
                *bp = *p;  // BAD
          }
   }
   *bp = '\0 ';  // BAD
}
```

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 5

In some languages like Java the compiler checks if (local) variables are initialized before being used (objects and global variables are initialized by the compiler).

For instance compiling the following programs will fail:

P1 : { x := 3; y:= (x+3); z := (y+z); }
P2 : { x := 3; if (x > 10) then y:=1 ; else z:= 2 ; end ; x:= (y+3); }

Q1. With respect to variable initialization, several solutions can be adpoted depending on the programming language semantics:

1) nothing is done (no verification)
2) uses of uninitialized variable are detected at runtime
3) variables are initialized by the compilers
4) uses of uninitialized variable are detected at compile time

Discuss these different options with respect to:
1) cost
2) consequences from a safety and/or security point of view

Q2. Propose an algorithm allowing to compute at compile time the set of non-initialized variable for a small language (assignment, conditional statement, iteration).

Answers

option 1 (nothing is done) : like in C or C++
- no cost overhead
- safety and security risk when an uninitialized variable is used :
  unpredictable result, or re-use of « old » values stored in memory (heap or stack)

option 2 (verification at runtime) : like in Python
- runtime cost overhead (needs to store and chek initialization information at each assignment)
- exception may be raised at runtime when a non initialized variable is used (« denial of service »)

option 3 (initalization performed at compile time) : like in Java for objects and globals
- still a small runtime cost overhead (extra assigments to initialize variables)
- potential safety risk (e.g., access to a NULL object), but behavior is always the same (no « random » execution)

option 4 (verification at compile time) : like in Java for local variables
- no cost overhead
- the code produced is safe/secure wrt variable initialization, but the compiler may reject « correct » programs (verification is undecidable => conservative approach)

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 ...

Exercise 5

We consider a Java Class C1 with a public method m1() allowing to perform some computations on a **secret** resource *key* and returning some integer value. Clearly, this method should **not** be called by any **untrusted** caller. To ensure that, the caller should provide as a parameter to m1() some credential as a string s.
A check is performed within m1() to verify that the caller is legitimate. When it is the case, permission P, allowing to read key is granted. Later on this permission is disabled (when no longer required). The corresponding code (in pseudo Java) is given below.

```
import java.util.* ;

class C1 {

int key[N] ;  // secret resource of size N

public int m1 (String s, int length) {
 // s is used to authenticate the caller
 int i, sum, result ;
 b = checkAcess(s) ;
 if (b) enablePermission(P) ; // give read acces to buffer key
 try {
   if (b) {
         i=0 ;
         sum= 0 ;
         while (i<length) {
               sum = key[i] + sum ;
               i = i+1 ;
```

```
        } ;
          disablePermission(P) ; // disable acccess to buffer key
          if (sum>0)
                result = Hash(sum); // returns a positive hash value
          else
                result = -1 ;
          return result ;
     }
    } catch (IndexOutofBoudsException e) {
          // in case key is accessed out of bounds
         System.out.println("Error !") ;
   }
  }
  }
```

Q1. Why is it necessary/useful to explicitly enable permissions to read key inside m1()(since the caller credential is already explicitly checked beforehand) ? Indicate in which conditions enabling this permission is required or not required ...

Q2. The way permission P is enabled/disabled inside m1() is clearly **insecure**. Indicate why, and how to correct it.

Q3. If this code was written in C or C++, it would **not** be possible to enable/disable permission P like in Figure 2. Explain (in a few lines) which other solutions could be used in terms of access control (indicating their advantages and drawbacks).

Q4. If a trusted caller executes method m1(), which information could it get about secret buffer key ?
Assuming that function call Hash(sum) returns no confidential information about sum, does m1() leak some confidential information about ke ? If yes, which information, if not, why not ?

Answers

Q1. Enabling permission within m1 is required if one of its calling method (in the call
    stack) do not have already P permission. It is useless otherwise.

Q2. The pb occurs is the IndexOutofBounds exception is raised during a call
     to m1. In this case permission P remains enabled ...

Q3. For a C code, access to method m1 could ne controlled
     - either at the OS level, but it works only to protect against external
       caller (and the OS itself should be secure ...), not against caller
      from the same application
     - or by using specific hardware (assuming this HW is available on the
       platform ...)

Q4. The call to m1 could leak:
1. the value of Hash(sum), where sum depends on the secret key
2. the fact that sum is <=0
3. some indications about the size of the key, if an exception occurs,
   or by estimating the execution time of this function ...