Questões de Segurança em Engenharia de Software (QSES), 2016/17
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Exame de 1ª época — 18 de Janeiro de 2017
Duração: 2:00

1. Considere os seguintes fragmentos de código **(i)** a **(iv)**.

```c
// (i) linguagem: C
void processUserFile(const char* user) {
  char filename[PATH_MAX+1];
  sprintf(filename, "/home/%s/someFile.txt", user);
  if (access(filename, R_OK) == 0) { // Check that file is readable
    FILE* fp = fopen(filename, "r"); // If so, open and process it
    processFile(fp);
    fclose(fp);
  }
}
```

```java
// (ii) linguagem: Java
MyData readData(...) ... {
  ObjectInputStream in = new ObjectInputStream(...);
  return (MyData) in.readObject();
}
```

```java
// (iii) linguagem: Java
void deleteUser(String id) ... {
  Connection c = ...; // Database connection
  PreparedStatement s =
    c.prepareStatement("DELETE FROM Users WHERE Id = '" + id +"'");
  s.execute();
}
```

```python
# (iv) linguagem: Python
def getInputAndLaunchProgram():
  param = readFromNetwork()
  # Execute some program using the system shell
  os.system("/opt/bin/someProgram " + param)
  return
```

(a) Para cada fragmento de código acima, identifique a(s) vulnerabilidade(s) de segurança em causa e explique sucintamente como estas podem ser exploradas por um adversário.

(b) Para *uma* das vulnerabilidades identificadas em **(a)**, explique sucintamente quais são os tipos de ameaças que se podem materializar segundo a taxonomia STRIDE[1].

(c) Para *duas* das vulnerabilidades identificadas em **(a)**, explique como estas podem podem ser mitigadas ajustando o código.

[1] S: Spoofing; T: Tampering; R: Repudiation; I: Information disclosure; D: Denial of service; E: Escalation of previlege

2. Considere o seguinte procedimento para comparar 2 arrays de tipo **char** em C:

```
int equalContents(char* a, char* b, int len) {
  for (int i = 0; i < len; i++)
    if ( a[i] != b[i] )
      return 0;
  return 1;
}
```

(a) Considere os mutantes M_1 a M_5 definidos por cada uma das seguintes mutações:

M_1	i=0	$\rightarrow$	i=1
M_2	i < len	$\rightarrow$	i <= len
M_3	a[i] != b[i]	$\rightarrow$	a[i] == b[i]
M_4	**return** 0;	$\rightarrow$	**return** i;
M_5	**return** 1;	$\rightarrow$	**return** 0;

Para cada um dos mutantes caracterize se possível dois testes (em termos de valores de "input" e "output" esperado): um teste que mate o mutante e outro teste que atinja o ponto de mutação mas não mata o mutante. Justifique as sua escolhas, ou porque não consegue encontrar mutantes nas condições pretendidas.

(b) Considere a execução simbólica de equalContents usando o Klee da seguinte forma:

```
char a[2];
char b[2];
klee_make_symbolic(a, sizeof(a), "a");
klee_make_symbolic(b, sizeof(b), "b");
int len = klee_range(0, 2, "len");
equalContents(a, b, len);
```

Apresente a correspondente árvore de execução simbólica.

(c) Suponha que equalContents é usado para uma computação que deverá ser segura, ex. num ambiente de um servidor que a é obtido de uma base de dados local e que b é recebido de um cliente na rede. Explique porque o código é sensível a "timing attacks", e de que forma pode ser ajustado por forma a preveni-los.

3. Dê respostas claras e sucintas às seguintes questões.

(a) Os chamados "zero-day exploits" tiram partido de vulnerabilidades num sistema que ainda não são conhecidas no domínio público, ou sequer por parte do responsável pelo código em causa. Porque podem ser particularmente gravosos ataques baseados em "zero-day exploits" ? Discuta a relação com o conceito de janela de vulnerabilidade ("vulnerability window").

(b) O que distingue uma validação de software baseada em testes face a outra baseada em técnicas de verificação formal? Refira uma vantagem e uma desvantagem de cada uma das aproximações.

(c) Em programas concorrentes (por ex. "multi-threaded") surgem os chamados "heisenbugs", cujas condições são difíceis de reproduzir usando testes ou "debugging". Porque é que isto acontece? Refira uma técnica que surta algum efeito na detecção de "heisenbugs", bem como possíveis limitações da mesma.

4. Considere anotações Java para "taint analysis", @trusted e @untrusted, que indicam se a execução de um método é respectivamente confiável ou não em termos de segurança. Em associação, assuma a seguinte política de segurança:

- A invocação de um método @trusted é segura se e só se os seus argumentos (e por inerência também o resultado obtido) dependerem apenas de valores constantes ou obtidos por um outro método @trusted. Caso contrário, considera-se que existe violação da segurança da computação por "input" não confiável.
- A invocação de um método @untrusted é segura se só se os seus argumentos (e portanto o resultado obtido) dependerem apenas de valores que sejam constantes ou obtidos por um outro método @untrusted. Caso contrário, considera-se que existe uma "leak" de informação confiável.

Agora, suponha que temos o seguinte fragmento com declaração de métodos (onde por uma questão de simplicidade todos os métodos lidam com valores **int**) e uma sequência de código onde esses métodos são invocados:

```
1  @trusted int t1(int v);
2  @trusted int t2(int v1, int v2);
3  @untrusted int u1(int v);
4  @untrusted int u2(int v1, int v2);
5  ...
6  int a, b, c, d;
7  a = t1(0) - t1(t1(123));
8  if (u1(0) > 0)
9    b = u1(123);
10 else
11   b = u1(a + u1(123));
12 if (b > 0)
13   c = u2(1, b + 22);
14 else
15   c = u2(1, u1(22) + t1(a + 22));
16 d = t2(a * a, t1(a) - t1(0));
17 d = t2(a * d, t1(d) - t1(c + t1(a)));
18 d = t2(a * d, t1(d) - t1(a));
```

Indique, justificando, quais das várias chamadas aos métodos (t1, t2, u1, e u2; 21 no total) podem ser consideradas seguras.