Secure Programming Lecture 8: Race Conditions

David Aspinall, Informatics @ Edinburgh

9th October 2019

Recap

We have looked at:

- examples of vulnerabilities and exploits
- particular programming failure patterns
- software based mitigations

In this lecture we consider a new vulnerability category and also a new defence strategy

- **language-based security** principles

for (ensuring) secure programs.

We introduce security vulnerabilities that can arise in concurrent systems, due to multi-processes or multi-threading.

Race conditions with check before use

```
res = access("/tmp/userfile", R_OK);
if (res!=0)
    die("access");

/* ok, we can read from /tmp/userfile */
fd = open("/tmp/userfile", O_RDONLY);
```

API docs (GNU C library)

`int access(const char *filename, int how)`

The access function checks to see whether the file named by filename can be accessed in the way specified by the how argument. The how argument either can be the bitwise OR of the flags R_OK, W_OK, X_OK, or the existence test F_OK.

This function uses the real user and group IDs of the calling process, rather than the effective IDs, to check for access permission. As a result, if you use the function from a setuid or setgid program (see How Change Persona), it gives information relative to the user who actually ran the program.

The return value is 0 if the access is permitted, and -1 otherwise. (In other words, treated as a predicate function, access returns true if the requested access is denied.)

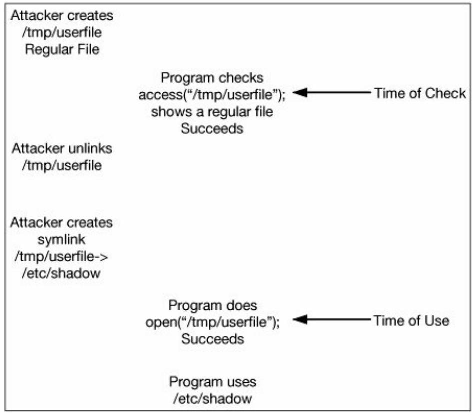
Race conditions with check before use

```
res = access("/tmp/userfile", R_OK);
if (res!=0)
    die("access");

/* ok, we can read from /tmp/userfile */
fd = open("/tmp/userfile", O_RDONLY);
```

- `access()` is designed for setuid programs
- privilege check on real user id (user running prog)
- `open()` returns a **file descriptor**
- f.d. is data type that refers to specific file

Time of Check to Time of Use (TOCTOU)



How can this be exploited?

- Unix runs multiple processes at once
  - Attacker runs a process alongside suid program
  - Must attack at exactly right moment
- Processes are scheduled by the OS
  - maybe on multiple CPUs
- Attacker may be able to influence scheduling
  - slow down system, send job control signals
- Attacker may be able to *automatically* schedule attack
  - e.g. Linux **inotify** API for monitoring file system

General problem: repeatedly looking up pathnames

Kernel resolves pathnames to *inodes* using file system.

Looking up file status twice repeats this:

```
stat("/tmp/bob", &sb);
...
stat("/tmp/bob", &sb);
```

If `/tmp/bob` (or `/tmp/`) change between the two calls, different files are examined by the two calls!

Fix: using file descriptors instead

File descriptors contain the resolved inode.

```
fd=open("/tmp/bob", O_RDWR);
fstat(fd, &sb);
...
fstat(fd, &sb);
```

This always examines the same (actual) file on disk twice, whatever `/tmp/bob` points to by the second call.

Even if the file has been deleted from the filesystem the inode is not deallocated until the reference count becomes zero.

Risky patterns: using same filename twice

1. A status check like
   - `stat()`
   - `lstat()`
   - `access()`
2. An access to the file like
   - `open()`, `fopen()`,
   - `chmod()`, `chgrp()`, `chown()`,
   - `unlink()`, `rename()`,
   - `link()`, `symlink()`

Better to use the file descriptor based calls instead:

- `fstat()`, `fchmod()`, and `fchown()`

Windows APIs a bit better here (but still tricky areas like the following).

Permission Races

```
FILE *fp;
int fd;

if (!(fp=fopen(myfile, "w+")))
    die("fopen");

/* we'll use fchmod() to prevent a race condition */
fd=fileno(fp);
/* let's modify the permissions */
if (fchmod(fd, 0600)==-1)
    die("fchmod");
```

- `fopen()` creates a file with default perms 0666

Exercise. (Recall labs): review the codes for file permissions and masks on Linux.

Ownership races

```
drop_privs();

if ((fd=open(myfile, O_RDWR | O_CREAT | O_EXCL, 0600))<0)
    die("open");

regain_privs();

/* take ownership of the file */
if (fchown(fd, geteuid(), getegid())==-1)
    die("fchown");
```

Directory position race

GNU file utils had a race vulnerability in recursive deletion. Example strace for `rm -fr /tmp/a` removing `/tmp/a/b/c` tree:

```
chdir("/tmp/a")
chdir("b")
chdir("c")
chdir("..")
rmdir("c")
chdir("..")
rmdir("b")
fchdir(3)
rmdir("/tmp/a")
```

Question. Can you see an attack here?

- let `rm` work until it gets into `/tmp/a/b/c`
- move c directory to `/tmp/c`
- then two `chdir("..")`s navigate to `/`

Races with temporary files

```c
char temp[1024];
int fd;
strcpy(temp, "/tmp/tmpXXXX");
if (!mktemp(temp))
    die("mktemp");
fd=open(temp, O_CREAT | O_RDWR, 0700);
if (fd<0)
{
    perror("open");
    exit(1);
}
```

Question. Can you see two security issues here?

- `mktemp()` replaces XXXX with random data
- unique so *not* completely unpredictable
- moreover, has race condition
- (although better than old `foobar.PID` scheme)

Recommended replacement: `fd = mkstemp(temp)`.

Risky Banking

```java
public class BankAccount {

  private int balance;

  public BankAccount(int initialBalance) {
      if (initialBalance < 0)
        throw new
          IllegalArgumentException("initial balance must be >= 0");

      balance = initialBalance;
  }
}
```

Risky Banking

```java
public class BankAccount {

    public void adjustBalance(int adjustment) {
       balance = balance + adjustment;
    }
}
```

Q: What's wrong with this code?

Risky Banking

```java
public class BankAccount {

    public void adjustBalance(int adjustment) {
       balance = balance + adjustment;
    }
}
```

A: it goes wrong in a multi-threaded context.

Under the bonnet: Java bytecode

```
[dice]da: javac BankAccount.java
[dice]da: javap -c BankAccount
Compiled from "BankAccount.java"
public BankAccount1(int);
 Code:
   0: aload_0                 // push address of this object
   1: invokespecial #1        // Method java/lang/Object."<init>":()V
   4: iload_1                 // push first argument integer
   5: ifge          18
   8: new           #2        // class java/lang/IllegalArgumentException
  11: dup
  12: ldc           #3        // String initial balance must be >= 0
  14: invokespecial #4        // Method java/lang/IllegalArgumentExceptio
  17: athrow
  18: aload_0                 // push address of this object
  19: iload_1                 // push first argument integer
  20: putfield      #5        // store in field balance
  23: return
```

```
public void adjustBalance(int);
  Code:
     0: aload_0           // push address of this object
     1: aload_0           //  and again
     2: getfield      #5  // fetch field balance
     5: iload_1           // first argument: adjustment
     6: iadd              // top of stack = this.balance + adjustment
     7: putfield      #5  // store in field balance
    10: return
```

Observe that:

```
    balance = balance + adjustment
```

is implemented in these steps:

```
    temp = balance
    temp = temp + adjustment
    balance = temp
```

where `temp` is a location in the (thread local) stack.

Racy interleaving: missed update 1

```
        Thread 1                  Thread 2
        ========                  ========

   temp1 = balance
                              temp2 = balance

   temp1 = temp1+adj1
                              temp2 = temp2+adj2

   balance = temp1
                              balance = temp2
```

- Final balance loses the adjustment `adj1`.

Racy interleaving: missed update 2

```
        Thread 1                  Thread 2
        ========                  ========

   temp1 = balance
                              temp2 = balance

   temp1 = temp1+adj1
                              temp2 = temp2+adj2

                              balance = temp2
   balance = temp1
```

- Final balance loses the adjustment `adj2`.

Data races defined

A data race is a race condition at the level of atomic memory accesses. It is the root cause of many subtle programming errors involving multi-threaded programs.

Data Race

A *data race* occurs when two or more threads access a shared variable:

1. (potentially) at the same time, and
2. at least one of the accesses is a write

Bugs from data races

Data races are usually accidental bugs.

- Lead to non-determinism
- Buggy behaviour may be very rare
- Hence difficult to reproduce: a "heisenbug"

Occasionally data races are *intentional* and safe:

- E.g., write-write races which write the same value
- Used knowingly e.g., in *lock-free* algorithms

This kind of thing is usually just for expert library code or O/S kernel developers.

Normal application developers should aim to write **data race free** programs.

Why can data races lead to security flaws?

Just as with race conditions:

- attacker may be able to influence thread scheduling
- or execute many, many times
- ... to cause an erroneous calculation/inconsistent value

Additionally, racy programs may have a strange issue:

- circular *causality* loops: undefined behaviour
- which allows registers to have any values..
- prevented by making **no out-of-thin-air** requirement

Java Memory Model: No Out-of-Thin-Air

Requirement: *A program should not be able to read values that couldn't be written by that program.*

```
Thread 1           Thread 2
--------------------------
r1 := x            r2 := y
y := r1            x := r2
print r1           print r2
```

- x, y are shared memory locations, initially both 0
- r1 and r2 are thread-local memory locations

The only possible result should be printing two zeros because no other value appears in or can be created by the program.

However, certain compiler/CPU optimisations would allow *any* value to be output here! (**Q.** Why is that bad?)

Write speculation breaks no out-of-thin-air

```
Thread 1           Thread 2
--------------------------
r1 := x            r2 := y
y := r1            x := r2
print r1           print r2
```

using **write speculation** this can be executed as

```
Thread 1           Thread 2
--------------------------
y := 42
r1 := x            r2 := y
if (r1 != 42)      x := r2
    y := r1
print r1           print r2
```

Now the example program could output 42!

Exercise. Give an interleaved execution showing this.

Hardware security

2018: *Meltdown* and *Spectre* announced.

CPU architecture bugs affecting most current CPUs.

- Combine a *race condition* with *side-channel attack*
  - result: process A steals data from process B
  - attacks are generally undetectable
- Complex CPUs use *microcode* to implement ISAs
  - bugs/vulns also possible in microcode
  - but workarounds/repairs possible

Emerging areas: hardware security cost-risk trade-off assessments for security mitigations.

The coursework asks you to study these vulnerabilities in more detail.

Ensuring atomicity

In general, race conditions are prevented by ensuring that compound operations occur *atomically*.

- Examples previously with APIs for file systems
- If we are getting a value (file, variable, etc):
  - broken: **test**, then **get** (TOCTOU)
  - fix: combined API function **test-and-get**

Question. How can we write API functions that ensure atomicity?

- usually: enforce *mutual exclusion*
- or: use a *transaction* mechanism (has rollback)

Databases and file systems allow high throughput concurrency with transactions. *Transactional memory* is still a research topic.

Using locks

For multi-threaded application programs, e.g., in Java

- **locks** to ensure mutual exclusion for shared resources

Sometimes programmers are *forgetful* about doing this

- path through code possible without locking
- or use complicated, implicit conventions
- e.g., lock objects stored/removed in memory

It's better to be carefully explicit about locking conventions.

Safer online banking

Returning to the banking example:

```java
protected final Object lock = new Object();

@GuardedBy("lock")
private int balance;
```

- Whenever we access `balance`, `lock` should be held
- `GuardedBy` annotation is a hint from the developer
  - readable by other developers
  - but also by a tool, so it can be checked
- Several fields might be protected by the same lock

We can split the API into internal and external methods:

```java
protected int readBalance() {
   return balance;
}

protected void adjustBalance(int adjustment) {
   balance = balance + adjustment;
}

public void credit(int amount) {
   if (amount < 0)
     throw new IllegalArgumentException("credit amount must be >= 0");

   synchronized (lock) {
     adjustBalance(amount);
   }
}
```

But we need to be careful that the locking strategy is followed in all subclasses.

For more, see Contemplate's technical briefing

Dynamic analysis

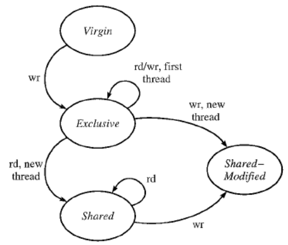
Dynamic analysis is in principle very expensive: monitor every access to every memory location, and see whether the access *might have raced* with a previous access from a different thread.

The **Lockset algorithm** simplifies this using the heuristic/expectation that every shared variable is protected by at least one lock.

- For each location `x`, initialise `C(x)` be all locks
- For each thread `t`, let `locks(t)` be locks held by `t`
- On each access to `x` from thread `t`
  - refine `C(x)` by removing locks not in `locks(t)`
  - if `C(x)={}` then give a warning

The *Eraser* tool operates a tuned version of this algorithm that distinguishes the kinds of access.

Eraser state model for shared locations



- Calculate locksets for *Shared* and *Shared-Modified*
- Only report errors in the *Shared-Modified* state

Eraser implemented this using binary modification to instrument a program dynamically.

Static analysis for race detection

Can use a static version of the Lockset algorithm. Advantages:

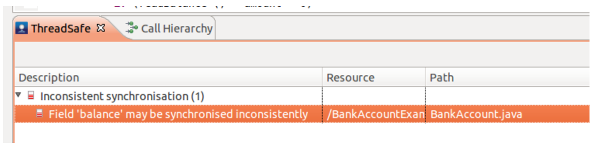
- Spot data races that are missed by dynamic tool
  - dynamic: may not explore paths "near enough"
- Doesn't impact code execution speed
  - dynamic: instrumentation gives significant slow-down

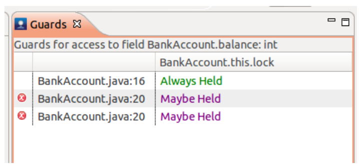
Disadvantages:

- Difficult to track locks held in data structures, etc.

The analysis can be made precise if programmers use `GuardedBy` annotations to describe the locking policy. Otherwise a tool has to guess the relevant locks and use heuristics to report discrepancies.

Contemplate's ThreadSafe tool





Review Questions

Race Conditions

- Using an example based on Unix file handling, describe what a *race condition* is, and explain how an attacker can exploit it.

Data races

- Describe the two necessary conditions for a program to contain a data race.
- Discuss whether it is possible for a racy program to compute a completely arbitrary value.

Program securely

- Describe two programming techniques that can be used to avoid security issues with race conditions.

References and credits

This lecture included examples from:

- M. Dowd, J. McDonald and J. Schuh. *The Art of Software Security Assessment*, Addison-Wesley 2007. The Unix file samples and TOCTOU picture are from Chapter 9.
- Contemplate Ltd's technical briefing on its *ThreadSafe* tool.
- Savage et al. *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, ACM TOCS, **15**(4), 1997.