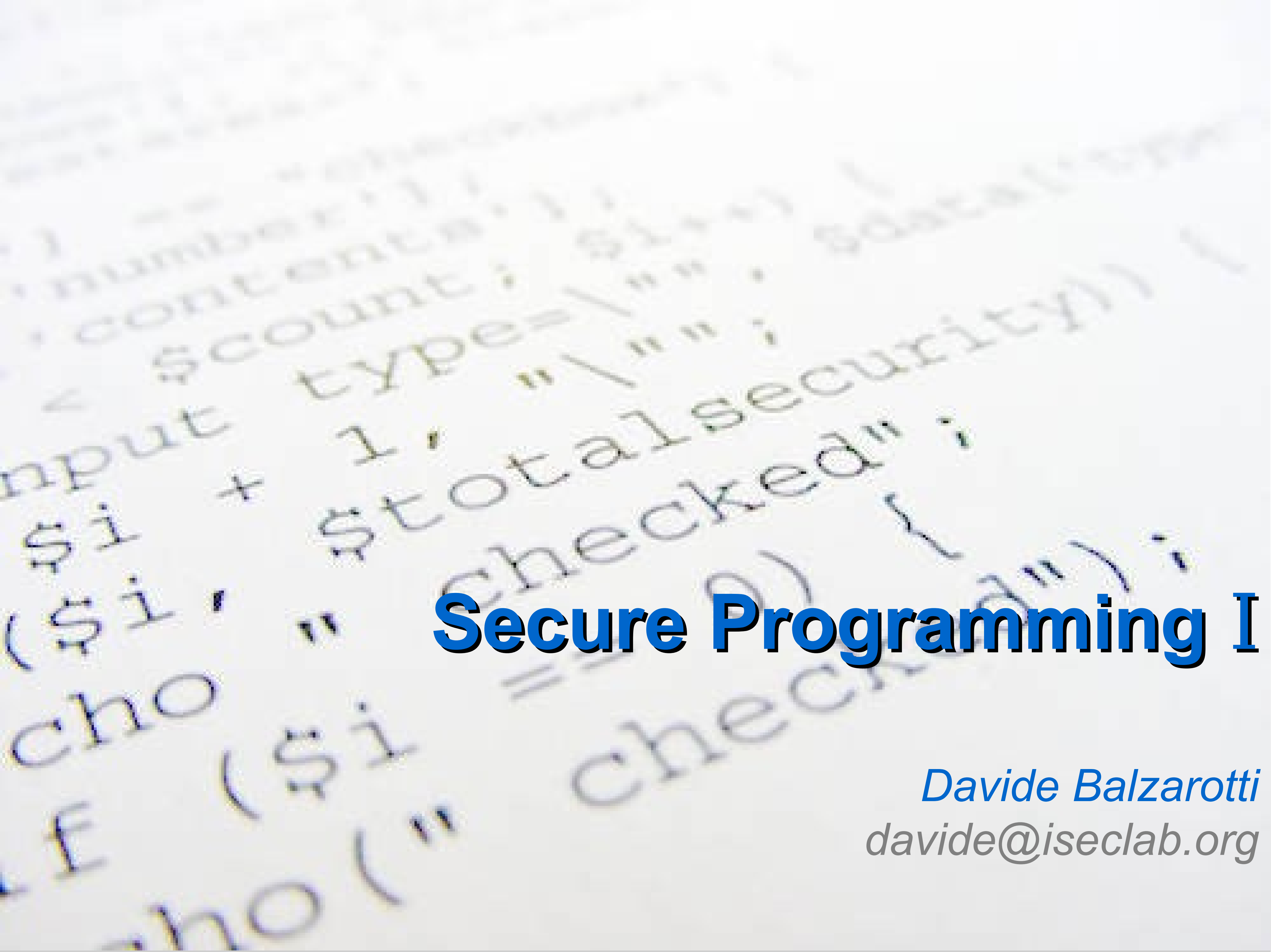


Secure Programming I

Davide Balzarotti
davide@iseclab.org

Overview

- Parallel execution of tasks
  - multi-process or multi-threaded environment
  - tasks can interact with each other

- Interaction
  - shared memory (or address space)
  - file system
  - signals

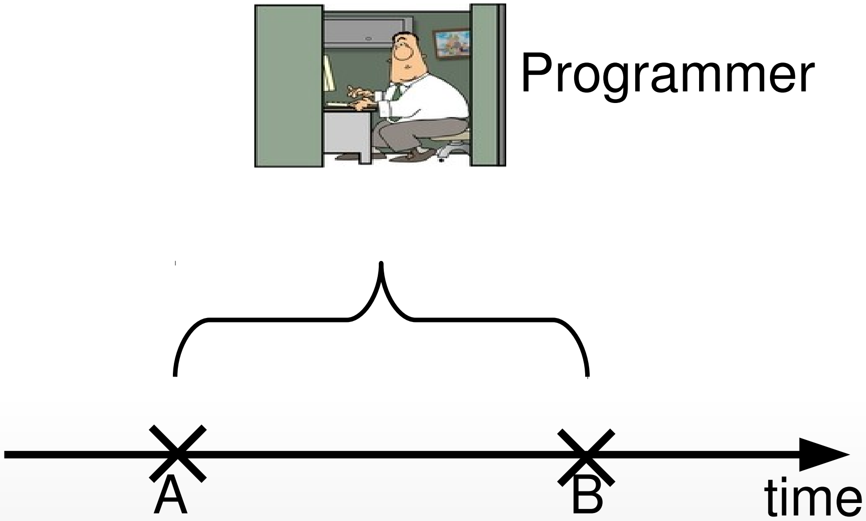
- Results of tasks depends on relative timing of events
  - Indeterministic behavior

Race Conditions

- Race conditions
  - alternative term for indeterministic behavior
  - often a robustness issue
  - but also many important security implications
- Assumption needs to hold for some time for correct behavior,
  - but assumption can be violated
- Time window when assumption can be violated
  - window of vulnerability

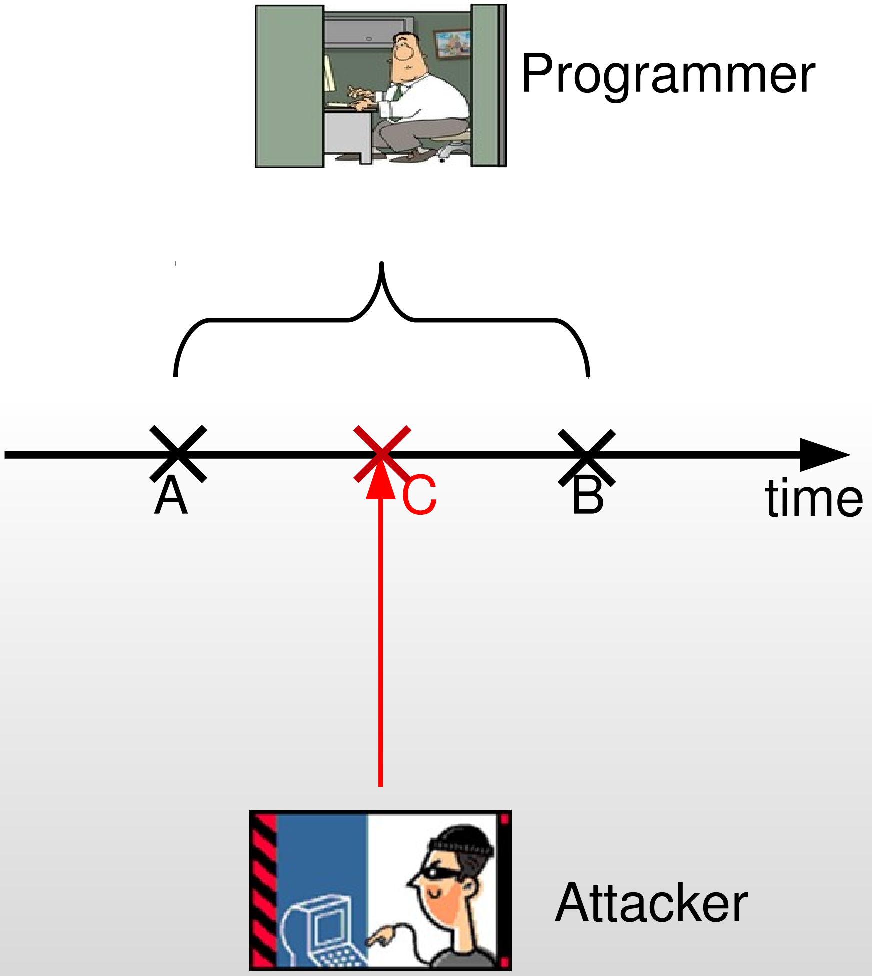
Race Conditions

- Programmer views a set of operations as atomic
- In reality, atomicity is not enforced
- Attacker can take advantage of this discrepancy



Race Conditions

- Programmer views a set of operations as atomic
- In reality, atomicity is not enforced
- Attacker can take advantage of this discrepancy



Shared Memory

- Sharing of memory between tasks can lead to races
  - Threads share the entire memory space
  - Processes may share memory mapped regions
- Use synchronization primitives:
  - locking, semaphores
  - Java:
    - `synchronized` classes and methods (Monitor model)
    - Atomic types (`java.util.concurrent.atomic.AtomicInteger`, etc)
- Avoid shared memory:
  - use message-passing model
  - still need to get the synchronization right!

Shared Memory

(trivial) example:

```
public class Counter extends HttpServlet {
    int count = 0;
    public void doGet(HttpServletRequest in,
                      HttpServletResponse out)
    {
        out.setContentType("text/plain");
        Printwriter p = out.getWriter();
        count++;
        p.println(count + " hits so far!");
    }
}
```

Shared Memory

(trivial) example:

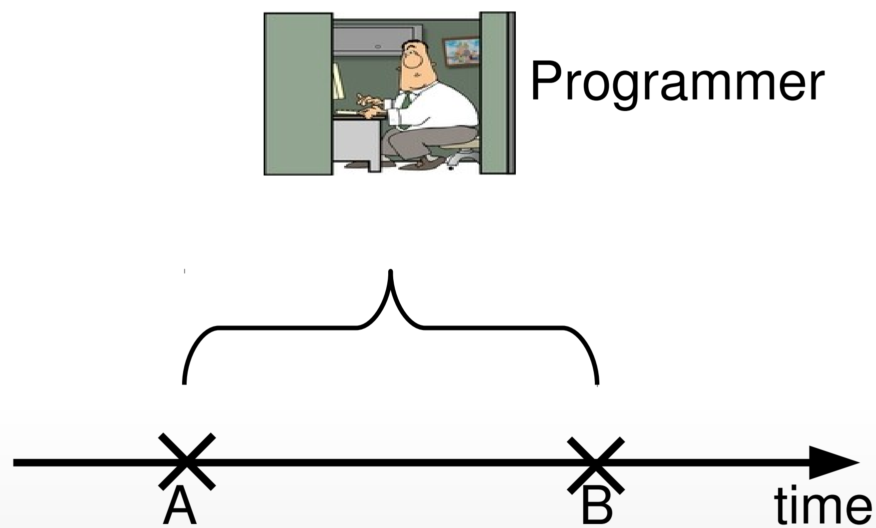
```
public class Counter extends HttpServlet {
    int count = 0;
    public void doGet(HttpServletRequest in,
                      HttpServletResponse out)
    {
        out.setContentType("text/plain");
        Printwriter p = out.getWriter();
        count++;
        p.println(count + " hits sc
    }
}
```

- Looks atomic (1 line of code!)
    - It's not!
- Simple race:
    - 2 threads read count
    - both write count+1
    - missed 1 increment

Race Conditions

Sequence of operations (A,B):

- Is not atomic
- can be interrupted at any time for arbitrary amounts of time



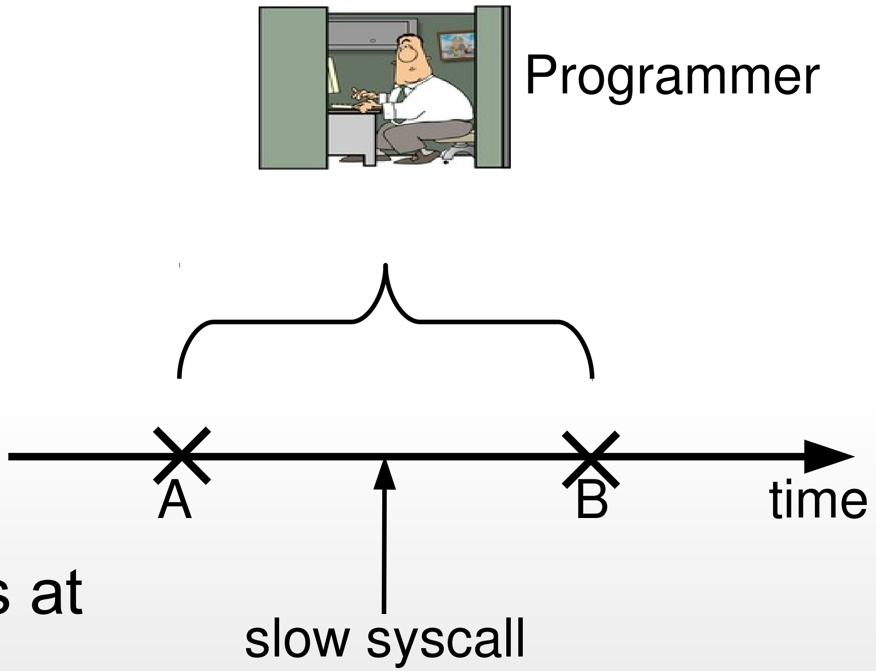
Scheduler can interrupt a process at any time

- Can happen between A and B

Race Conditions

Sequence of operations (A,B):

- Is not atomic
- can be interrupted at any time for arbitrary amounts of time



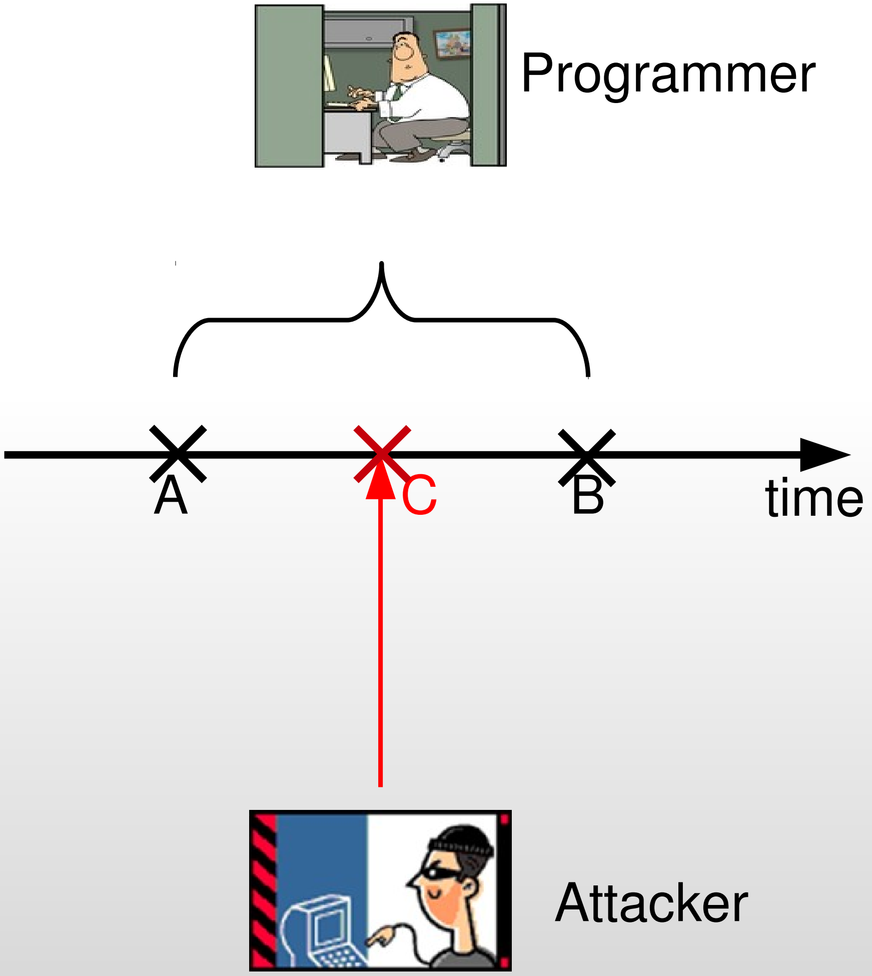
Scheduler can interrupt a process at any time

- Can happen between A and B
- Much more likely if there is a blocking system call in between

Window of Vulnerability

- Things go wrong if C happens between t_A and t_B.

- (t_A, t_B) is the window of vulnerability



Race Conditions

- Window of vulnerability can be very short
  - race condition problems are difficult to find with testing
  - difficult to reproduce and debug
- Myths about race conditions
  - "*races are hard to exploit*"
  - "*races cannot be exploited reliably*"
  - "*only 1 chance in 10000 that the attack will work!*"
- Attackers can often find ways to beat the odds!

Beating the Odds

- Can the attacker try the exploit 1 million times?
  - if yes, and the odds are 1 to 10000, then he has a reliable exploit

- Attacker can try to slow down the victim machine/process to improve the odds
  - high load
  - computational complexity attacks

- Attacker can run the attack many times in parallel to increase the probability that the attacking process will be scheduled by the processor at the right moment

Time-of-Check, Time-of-Use (TOCTOU)

- Time-of-Check, Time-of-Use (TOCTOU)
  - common race condition problem
- Problem:
  - Time-Of-Check (t_A): validity of assumption X on entity E is checked
  - Time-Of-Use (t_B): assuming X is still valid, E is used
  - Time-Of-Attack (t_C): assumption X is invalidated
  - $t_A < t_C < t_B$
- Program has to execute with elevated privilege
  - otherwise, attacker races for his own privileges

Time-of-Check, Time-of-Use

- Steps to access a resource
  - obtain reference to resource
  - query resource to obtain characteristics
  - analyze query results
  - if resource is fit, access it

- Often occurs in Unix file system accesses
  - check permissions for a certain file name (e.g., using access(2))
  - open the file, using the file name (e.g., using fopen(3))
  - four levels of indirection (symbolic link - hard link - inode - file descriptor)

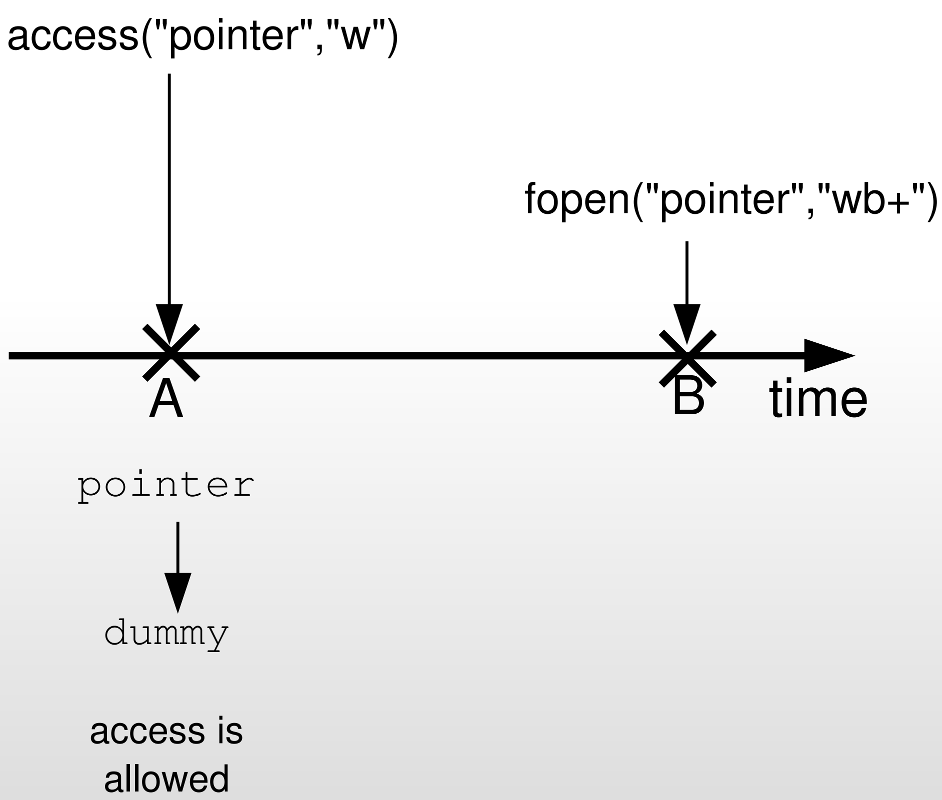
access/open Race

- Case study: setuid program
- `man 2 access`:
  "The check is done using the calling process's real UID and GID, rather than the effective IDs as is done when actually attempting an operation (e.g., open(2)) on the file. This allows set-user-ID programs to easily determine the invoking user's authority."

```
/* access returns 0 on success */
if(!access(file, W_OK)) {
   f = fopen(file, "wb+");
   write_to_file(f);
} else {
   fprintf(stderr, "Permission denied,
                      cannot open %s.\n", file);
}
```

access/open Race

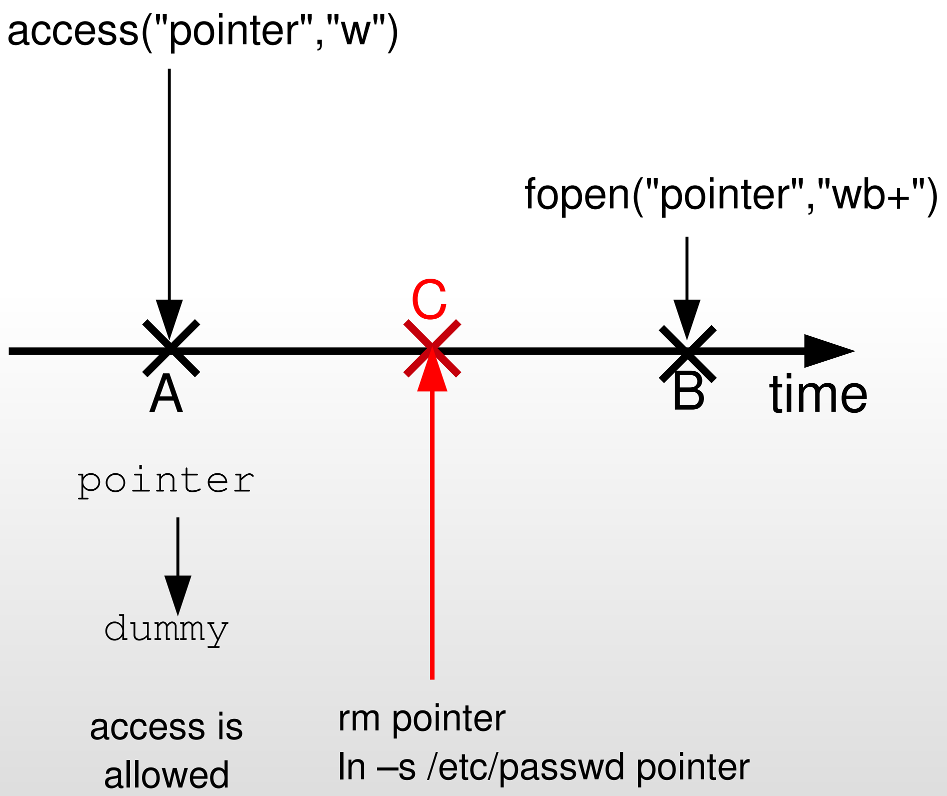
```
/* access returns 0 on success */
if(!access(file, W_OK)) {
   f = fopen(file, "wb+");
   write_to_file(f);
} else {
   ...
}
```



$ touch dummy; ln –s dummy pointer

access/open Race

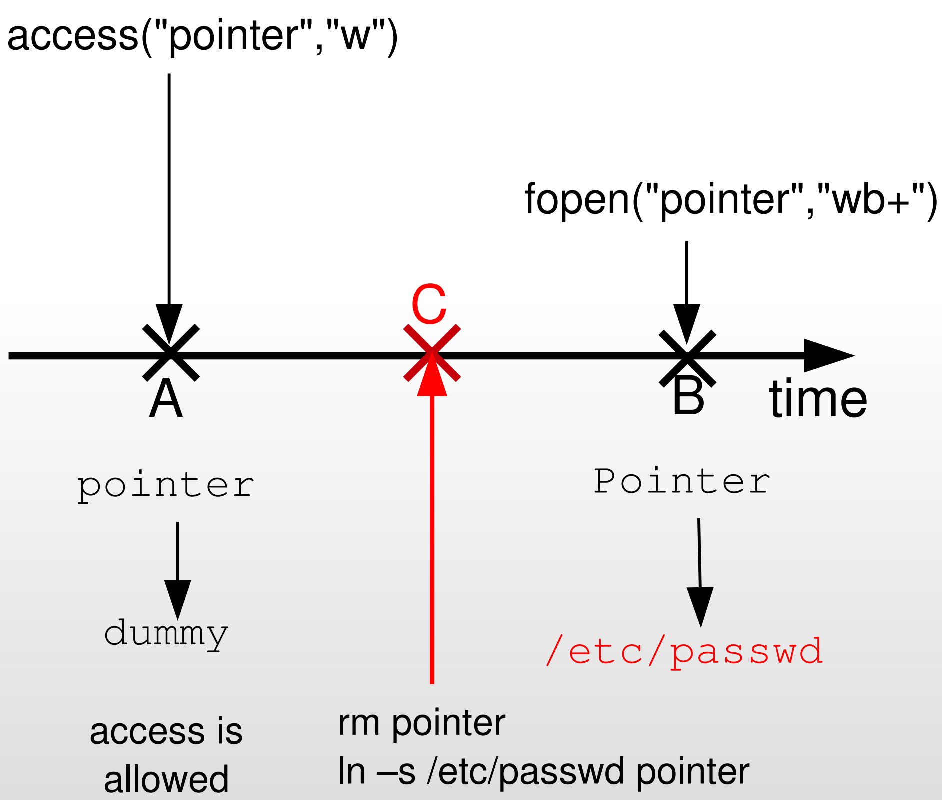
```
/* access returns 0 on success */
if(!access(file, W_OK)) {
   f = fopen(file, "wb+");
   write_to_file(f);
} else {
   ...
}
```



$ touch dummy; ln –s dummy pointer
$ rm pointer; ln –s /etc/passwd pointer

access/open Race

```
/* access returns 0 on success */
if(!access(file, W_OK)) {
   f = fopen(file, "wb+");
   write_to_file(f);
} else {
   ...
}
```



$ touch dummy; ln –s dummy pointer
$ rm pointer; ln –s /etc/passwd pointer

Script execve Race

- Filename redirection
  - soft links again

- Setuid Scripts
  - `execve()` system call invokes `seteuid()` call prior to executing program
  - A: program is a script, so command interpreter is loaded first
  - B: program interpreter (with root privileges) is invoked on script name
  - attacker can replace script content between step A and B

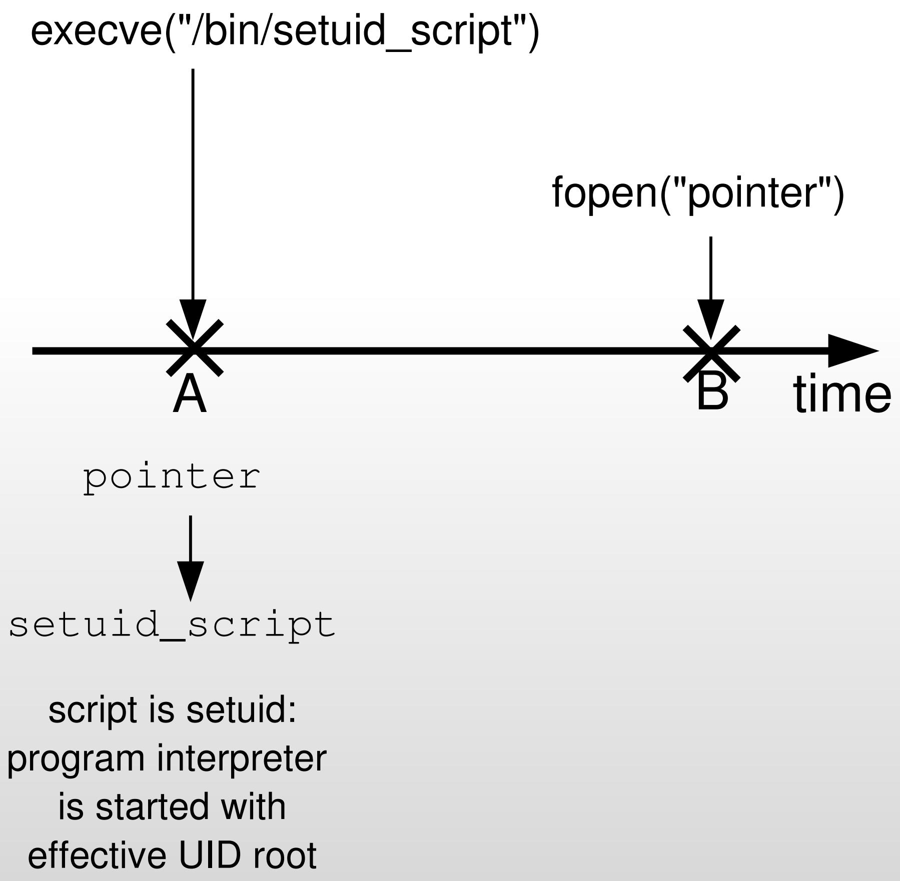
- Setuid not allowed on scripts on most platforms!
  - although there are some work-arounds

script execve Race

- A: program interpreter is started (with root privilege)
  - e.g: /bin/sh, /usr/bin/python,
- B: program interpreter opens script pointed to by "pointer"
- Interpreter runs the script

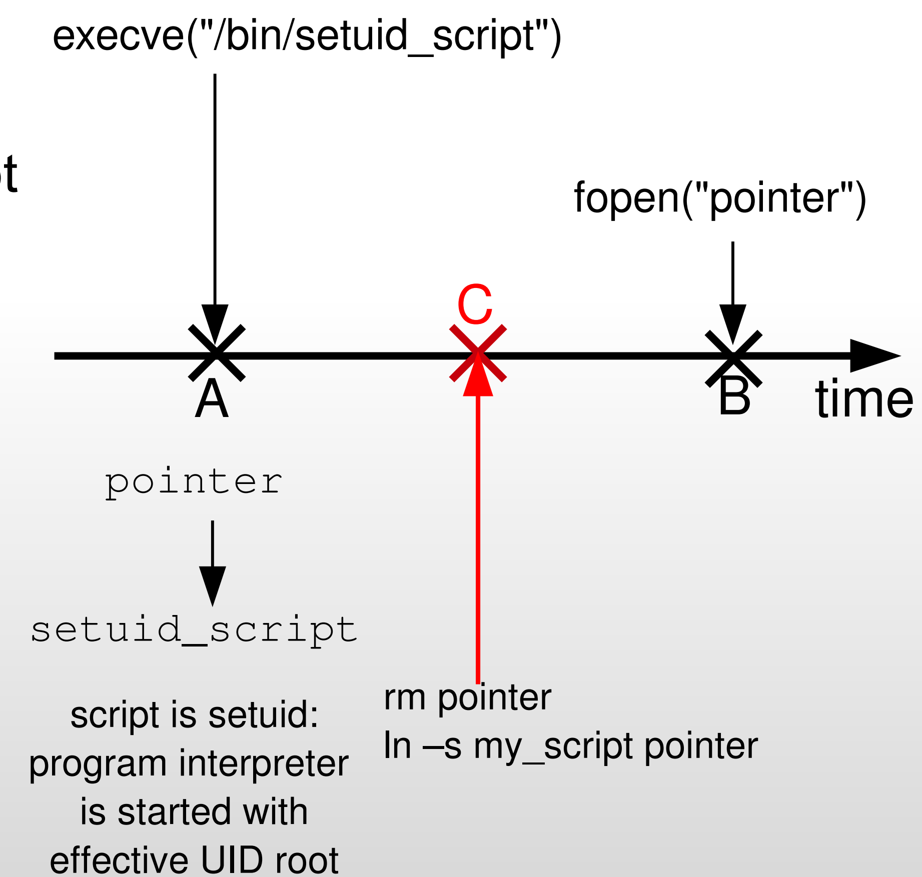
- Attack:

$ ln –s /bin/setuid_script pointer



script execve Race

- A: program interpreter is started (with root privilege)
  - e.g: /bin/sh, /usr/bin/python,
- B: program interpreter opens script pointed to by "pointer"
- Interpreter runs the script



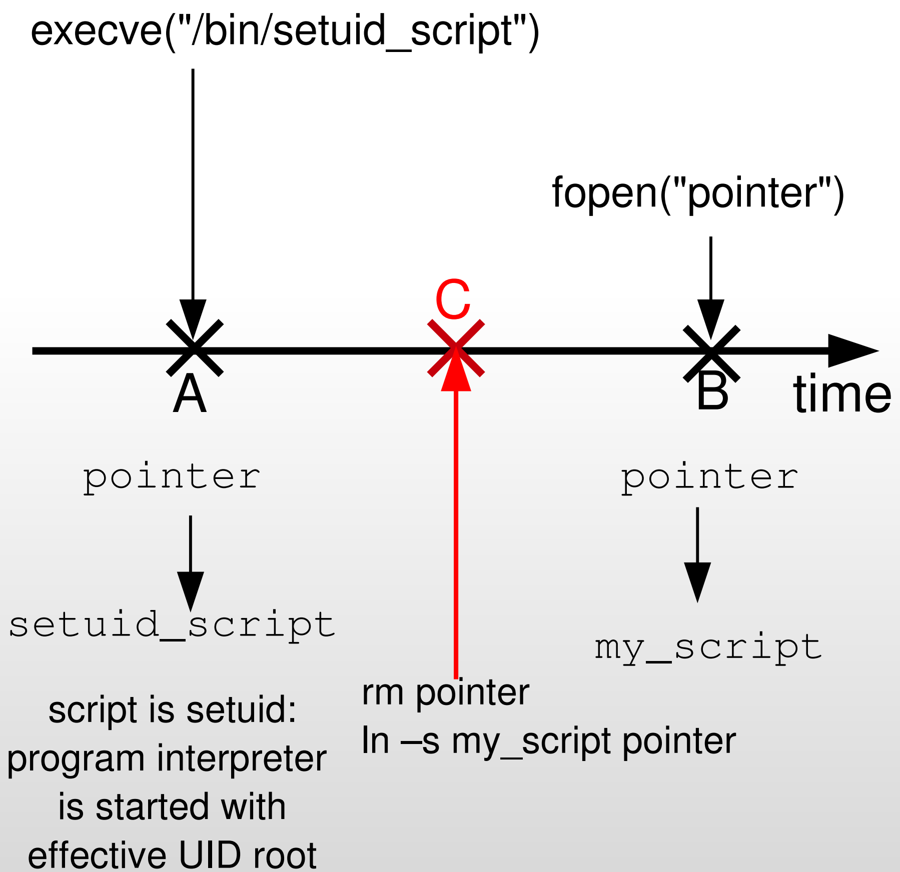
- Attack:

$ ln –s /bin/setuid_script pointer

$ rm pointer; ln –s my_script pointer

script execve Race

- A: program interpreter is started (with root privilege)
  - e.g: /bin/sh, /usr/bin/python,
- B: program interpreter opens script pointed to by "pointer"
- Interpreter runs the script



- Attack:

$ ln –s /bin/setuid_script pointer

$ rm pointer; ln –s my_script pointer

Directory operations

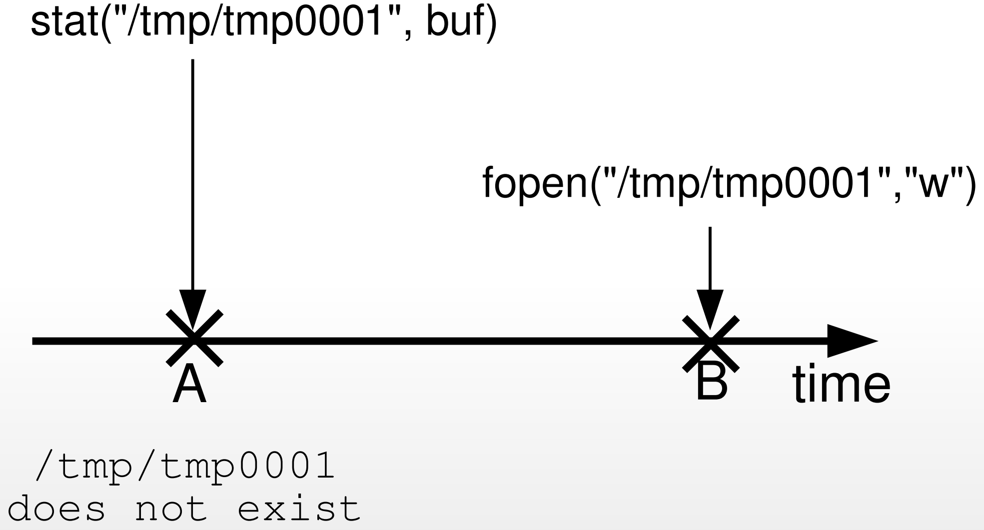
- "`rm -r`" race
  - rm can remove directory trees, traverses directories depth-first
  - issues chdir("..") to go one level up after removing a directory branch
  - by relocating subdirectory to another directory (while `rm -r` is running!), arbitrary files can be deleted

Races on temporary files

- Similar issues as with regular files
  - commonly opened in /tmp or /var/tmp
  - creating files in /tmp requires no special permissions
  - often guessable file name
- (One) Attack:
  - guess the tmp file name: "/tmp/tmp0001"
  - ln -s /etc/target /tmp/tmp0001
  - victim program will create file /etc/target for you, when it tries to create the temporary file!
  - if first guess doesn't work, try 1 million times

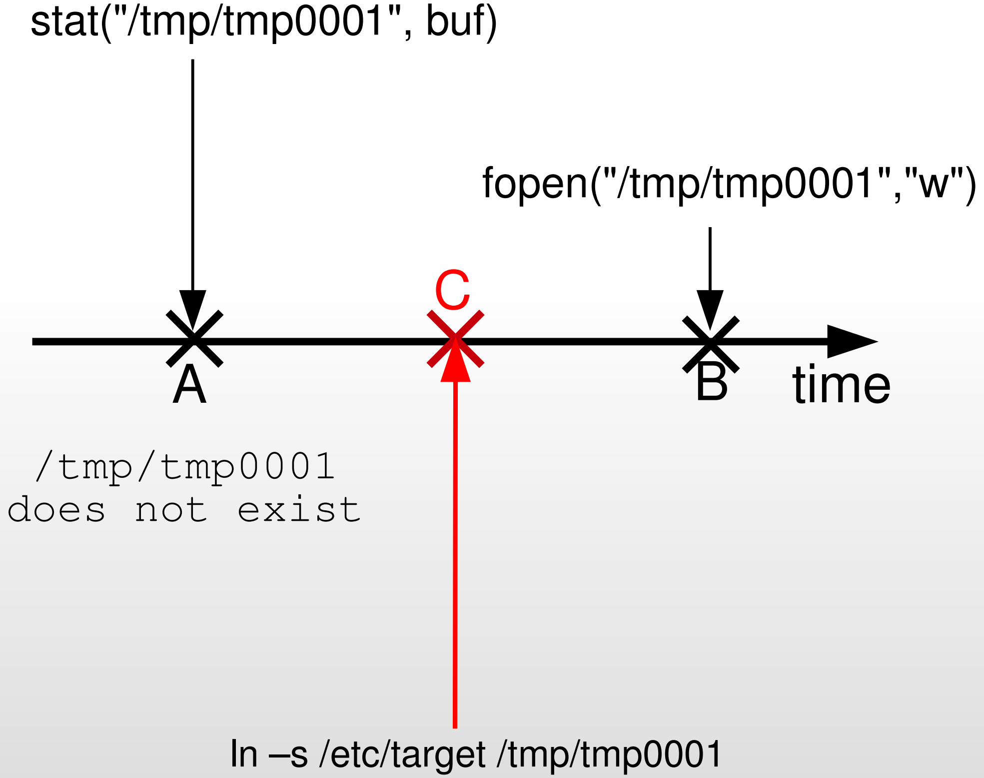
Races on temporary files

- A: program checks if file "/tmp/tmp0001" already exists
- B: program creates file "/tmp/tmp0001"



Races on temporary files

- A: program checks if file "/tmp/tmp0001" already exists
- B: program creates file "/tmp/tmp0001"

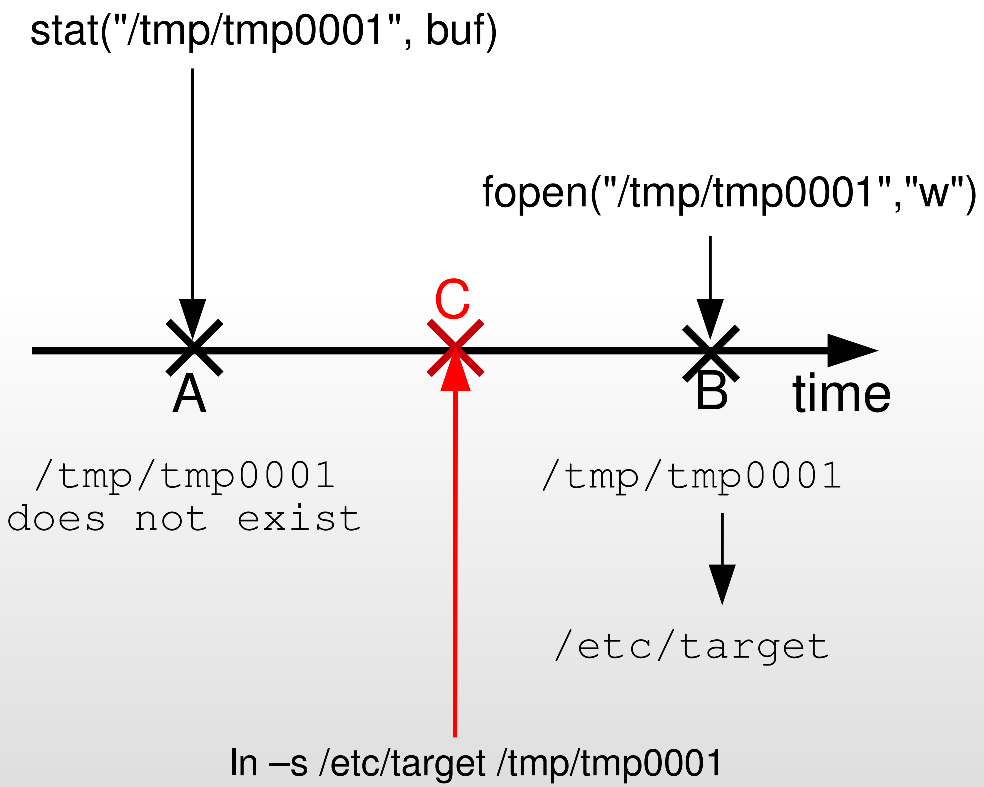


- Attack:

$ ln -s /etc/target /tmp/tmp0001

Races on temporary files

- A: program checks if file "/tmp/tmp0001" already exists
- B: program creates file "/tmp/tmp0001"
  - /etc/target is created!



- Attack:

$ ln -s /etc/target /tmp/tmp0001

Races on temporary files

- Temp Cleaners
  - programs that clean “old” temporary files from temp directories
  - first `lstat(2)` file, then use `unlink(2)` to remove files
- attack: arbitrary file deletion
  - race condition when attacker replaces file (softlink) between `lstat(2)` and `unlink(2)`
- attack: delete temporary file too early
  - delay program long enough until temp cleaner removes active temporary file

Races on temporary files

- “Secure” procedure
  - pick hard to guess filename (randomize part of name)
  - set umask appropriately (0066 is usually good)
  - atomically test for existance AND create the file
  - use open(2) O_CREAT|O_EXCL to create the file, opening it in the proper mode
  - if file exists, fopen will fail: try again with another file name (in a loop)
  - delete the file immediately using unlink(2)
    - until the file descriptor is open is still usable by the program
  - perform reads, writes, and seeks on the file as necessary
  - finally, close the file: it is automatically deleted

Races on temporary files

- umask issues
  - if all users have read access, can lead to leak of private data
  - if all users have write access, can lead to data tampering
    - programs treat their temporary files as trusted
    - they may not validate input from them
    - maybe I can find a vulnerability in the program if I can tamper with its temporary files
- Use library functions to create temporary files
  - don't roll your own implementation!
- Some library functions are insecure
  - `mktemp(3)` is not secure, use `mkstemp(3)` instead
  - old versions of `mkstemp(3)` did not set umask correctly

More examples

- File meta-information
  - `chown(2)` and `chmod(2)` are unsafe because they operate on file names
  - use `fchown(2)` and `fchmod(2)` that use file descriptors
- Logging/Crash reporting
  - example: Joe Editor vulnerability
  - when joe crashes (e.g., segmentation fault, xterm crashes)

    unconditionally append open buffers to local DEADJOE file
  - DEADJOE could be symbolic link to security-relevant file

More examples

- SQL select before insert
  - use select to check if a certain element already exists
  - when select returns no results, insert a (unique) element
- Race condition:
  - 2 processes may do this at the same time, leading to 2 insertions
- Countermeasures
  - locking
  - primary keys: use a single atomic insert. It will fail if key already exists

Computational Complexity Attacks

Beating the odds

- Window of vulnerability can be short
- Attacker can try to make the program run more slowly
- Computational complexity attacks
  - many algorithms are fast on average
  - ...but are slow in some corner cases?
- Example: filename lookups
  - deeply nested directory structure
  - chain of symbolic links
  - looking up the file in the FS will take longer!

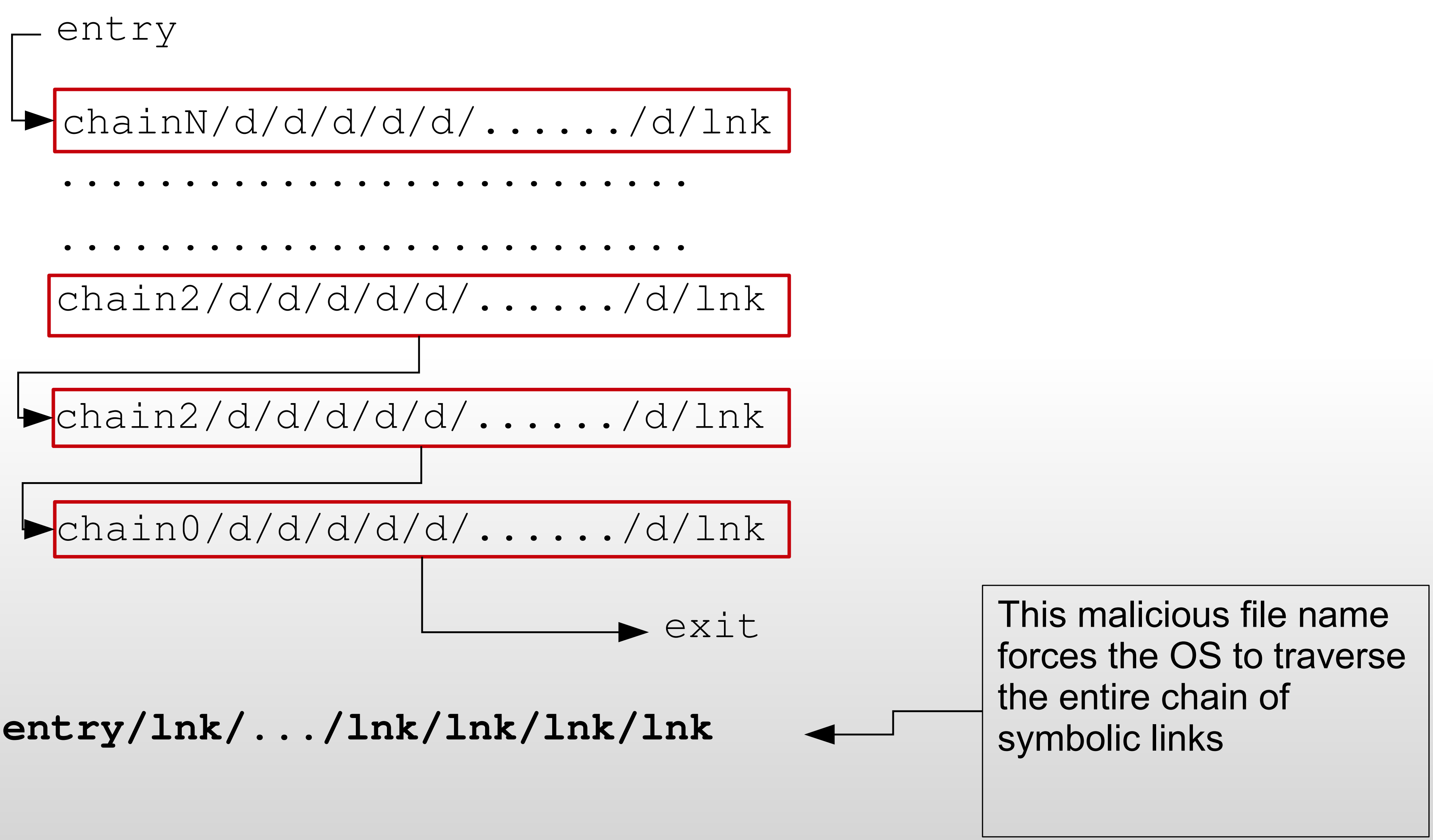
Slow file lookups

- Deeply nested directory structure:
  - `d/d/d/d/d/d/d/....../d/file.txt`
- To resolve this file name, the OS must:
  - look for directory named d in current working directory
  - look for directory named d in that directory
  - ...
  - look for file named file.text in final directory
- Limit to length of a file name:
  - MAXPATHLENGTH (4096 on my linux)
  - Max depth of ~2000

File System Maze

- Combine deeply nested directory structure with chain of symbolic links
  - MAXPATHLENGTH limits length of file parameter to a single system call (e.g, `open`,`access`)
  - But parts of a file name can themselves be links
  - Length of link chain limited by kernel parameter
    - 40 on my linux box
- Total file system lookups:
  - follow 40 chains...
  - ...each with 2000 nested directories
  - 80000 lookups!

File System Maze



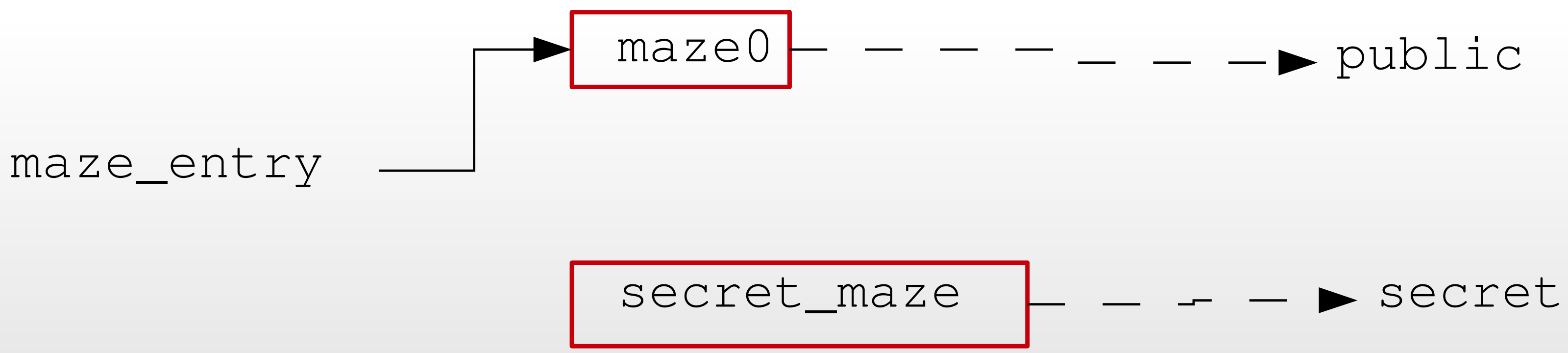
Mini-Demo

```
pmilani@rorschach:~$time cat
entry/lnk/lnk/lnk/lnk/lnk/lnk
/lnk/lnk/lnk/lnk/lnk/lnk/lnk/
lnk/lnk/lnk/lnk/lnk/lnk/lnk/l
nk/lnk/lnk/lnk/lnk/lnk/lnk/ln
k/lnk/lnk/lnk/lnk/lnk/lnk/lnk

CONTENT OF FILE


real    7m22.498s

user    0m0.000s

sys     0m4.964s
```

- In my tests, traversing the maze sometimes took over 7 minutes!
  - while keeping the disk busy with other operations
  - with the directories in the maze not already cached by the OS
- Reliably over half a second
- Plenty of time to exploit a race condition

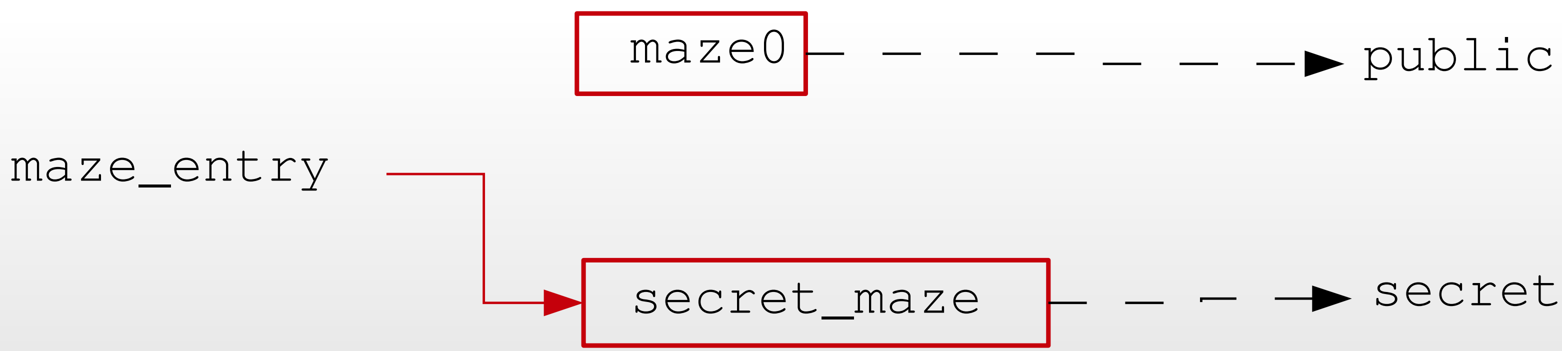
access/open Race Demo

- suid_cat:
  - vulnerable program: setuid version of cat utility
  - uses access to check if it can open a file
- multiple chains of symbolic links:



access/open Race Demo

- suid_cat:
  - vulnerable program: setuid version of cat utility
  - uses access to check if it can open a file
- multiple chains of symbolic links:



- Change the maze_entry while suid_cat is running!

Computational Complexity Attacks

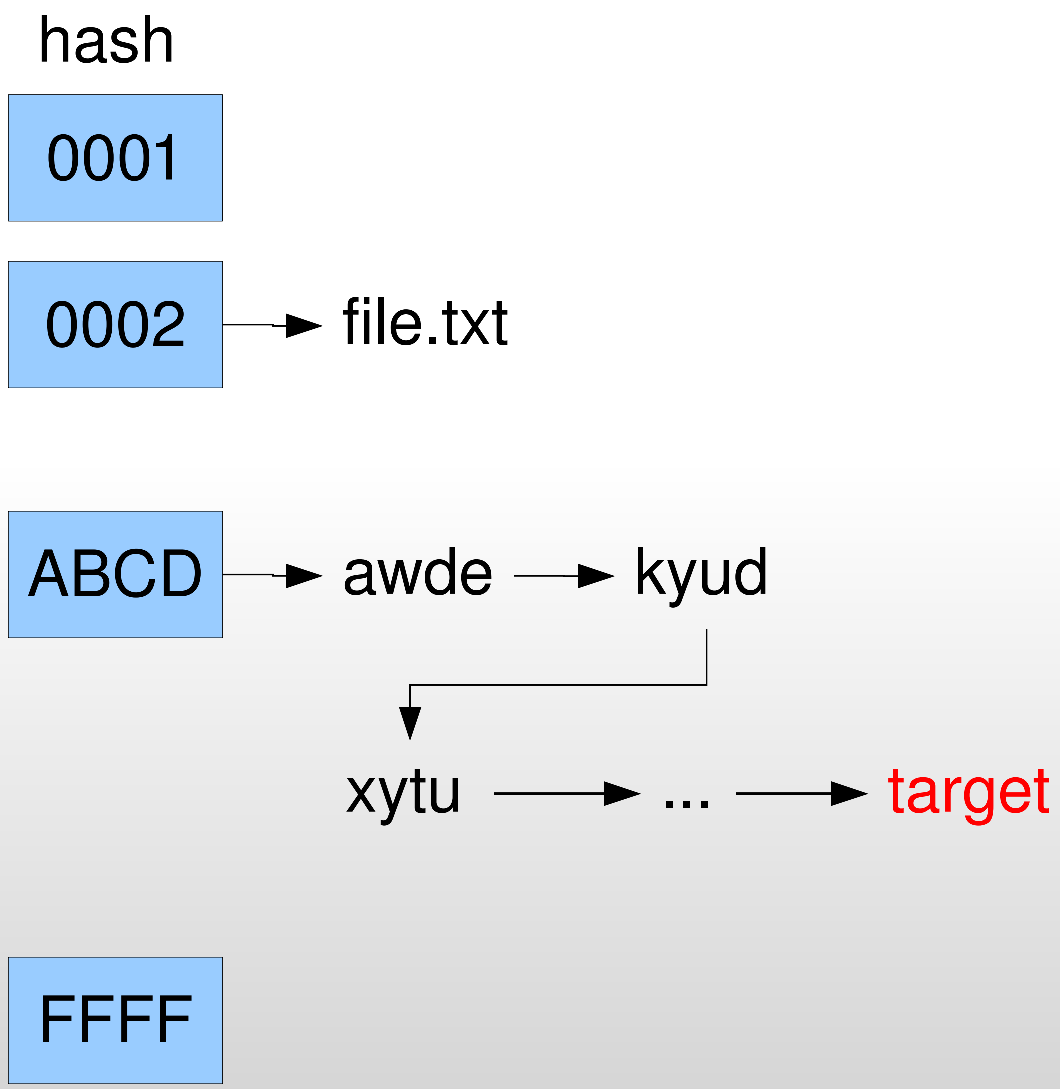
- Exploit worst-case performance of an algorithm
- Example: file lookup (again)
- How does OS store mapping between file names and inodes in a directory?
  - linked list or array?
    - what if there are 100000 files in directory?
    - too slow in practice
  - hash table!
    - good average performance
    - bad worst-case performance
    - can an attacker exploit this?

Hash Table

- Store objects in a number of buckets
- Each bucket is a linked list
- Choose bucket for an object X based on hash function h(X)
- Accessing an item in a hash table of N elements is O(1) most of the time
  - because most buckets hold 1 or 0 elements
- Worst case complexity is O(N)!
  - if all objects have the same hash
- Worst case does not occur accidentally (very unlikely)
- Attacker can make worst case happen

Computational Complexity Attacks

- File entries in a directory typically stored in a hash table
- Hash tables are slow when there are many entries in the same bucket
- Create 10000 files with same hash!



Detection and Prevention

Prevention

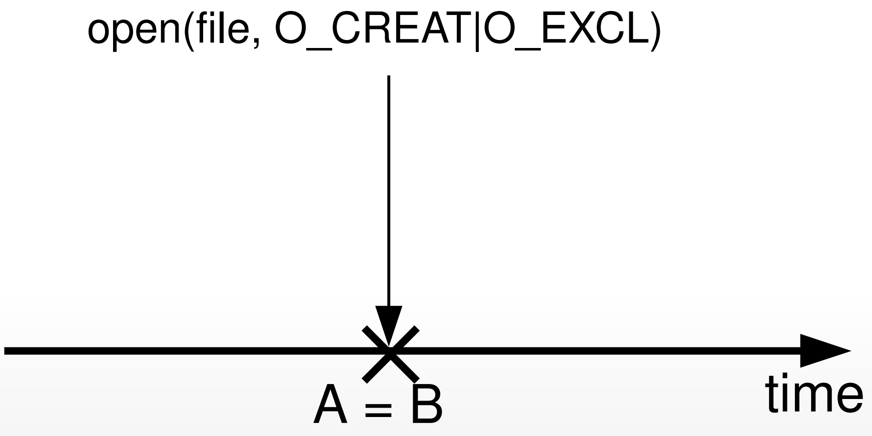
- Do not assume you are safe from race conditions just because:
  - Window of opportunity is short
    - Attacker may well be able to make it bigger!
  - Success is unlikely
    - Attacker may be able to try 1 million times!
    - Now that most of the computers have multiple cores, attacks are even easier

Preventing Race Conditions

- operate on file descriptors
  - not on file names (as much as possible)
- do not check access by yourself
  - (i.e., no use of `access(2)`)
- drop privileges instead and let the file system do the job
- Use the O_CREAT | O_EXCL flags to create a new file with open(2)
  - and be prepared to have the open call fail

Avoiding the access/open Race

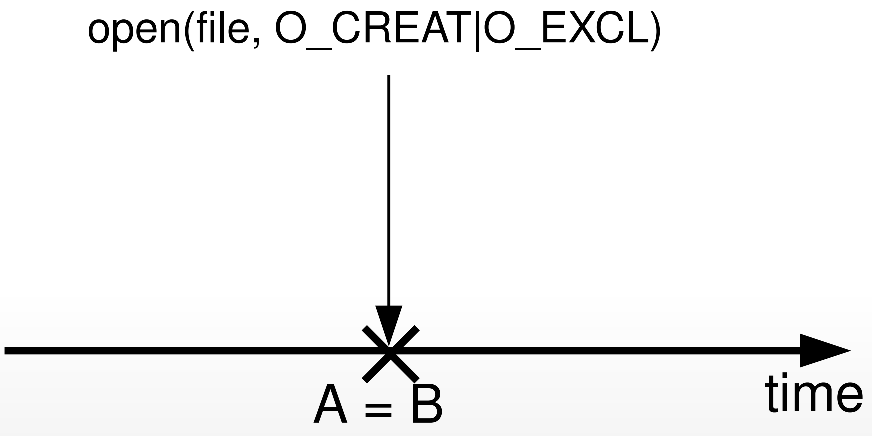
```
ruid=getuid();
euid=geteuid();
/* drop privileges */
seteuid(ruid);
int fd = open(file, O_CREAT|O_EXCL);
if(fd!=-1) {
   write_to_file(fd);
} else {
   fprintf(stderr, "Permission denied);
}
seteuid(euid);
```



- when acting on behalf of the user, assume his identity
  - let the operating system check permissions!
- this code snippet still has 2 potential vulnerabilities

Avoiding the access/open Race

```
ruid=getuid();
euid=geteuid();
/* drop privileges */
seteuid(ruid);
int fd = open(file, O_CREAT|O_EXCL);
if(fd!=-1) {
   write_to_file(fd);
} else {
   fprintf(stderr, "Permission denied);
}
seteuid(euid);
```



- check seteuid for errors
  - if setuid fails, you are effective UID is unchanged (you are still root!)
- also drop group privileges with setegid()

Prevention

- Some calls require file names

  `link(), mkdir(), mknod(), rmdir(), symlink(), unlink()`

- especially `unlink(2)` is troublesome

- Secure File Access
  - create “secure” directory
  - directory only write and executable by UID of process
  - check that no parent directory can be modified by attacker
  - walk up directory tree

    checking for permissions and links at each step

Locking

- Ensures exclusive access to a certain resource
  - Used to avoid accidental race conditions
  - advisory locking (processes need to cooperate)
  - not mandatory, therefore not secure
- Often, files are used for locking
  - portable (files can be created nearly everywhere)
  - “stuck” locks can be easily removed

Non FS Race Conditions

- Linux / BSD kernel ptrace(2) / execve(2) race condition

- ptrace(2)
  - debugging facility
  - used to access other process' registers and memory address space
  - allows to tamper with internal state and execution of a process
  - can only attach to processes of same UID, except when run by root

- execve(2)
  - execute program image
  - setuid functionality (modifying the process EUID)
  - not invoked when process is marked as being traced

execve/ptrace Race

- Problem with execve(2)
  1. first checks whether process is being traced
  2. open image (may block)
  3. allocate memory (may block)
  4. set process EUID according to setuid flags

- Window of vulnerability between step 1 and step 4
  - attacker can attach via ptrace
  - blocking kernel operations allow other user processes to run

- Kernel-side defense against this attack (locking)

Non-FS Race Conditions

- Windows DCOM / RPC vulnerability
- RPCSS service
  - multiple threads process single packet
  - one thread frees memory while other process still works on it
  - can result in memory corruption
  - and thus denial of service

Detection

- Static code analysis
  - specify potentially unsafe patterns
    and perform pattern matching on source code
    - trivial form:
      - grep access *.c

- source code analysis and model checking
  - MOPS (MOdel-checking Programs for Security properties)

Detection

- Static code analysis
  - Source code analysis and annotations / rules
    - RacerX (found problems in Linux and commercial software)
    - rccjava (found problems in java.io and java.util)
- Dynamic analysis
  - inferring data races during runtime
    - “Eraser: A Dynamic Data Race Detector for Multithreaded Programs”

Conclusion

- We looked at Race Conditions today
  - Overview
  - TOCTOU
  - Examples
  - Complexity attacks
  - Prevention and Detection