

# Internet Security 2

(aka Advanced InetSec)

## Race Conditions

Christian Platzer	<a href="mailto:cplatzer@seclab.tuwien.ac.at">cplatzer@seclab.tuwien.ac.at</a>
Paolo Milani Comparetti	<a href="mailto:pmilani@seclab.tuwien.ac.at">pmilani@seclab.tuwien.ac.at</a>
Clemens Kolbitsch	<a href="mailto:ck@seclab.tuwien.ac.at">ck@seclab.tuwien.ac.at</a>
Thorsten Holz	<a href="mailto:tho@seclab.tuwien.ac.at">tho@seclab.tuwien.ac.at</a>

# News from the Lab

*Int. Secure Systems Lab  
Technical University Vienna*

- Challenge 5 (forensics) was solved by 18 people
  - congratulations!
- ...and the winner is BERNHARD "Perfect Prime" MILLER (< 6 hours)
- Challenge 6 (fuzzing) starts today, and runs until next monday
  - coding challenge: no copying (we will use plagiarism detection systems)
- Forward project blog:
  - <http://blogs.ict-forward.eu/forward/>
  - extra chance to get 1 challenge awarded for good posts and comments

# Race Conditions

# Overview

*Int. Secure Systems Lab  
Technical University Vienna*

- 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**

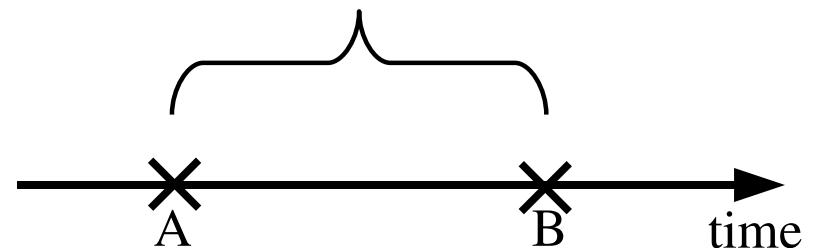
# Race Conditions

*Int. Secure Systems Lab  
Technical University Vienna*

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



Programmer



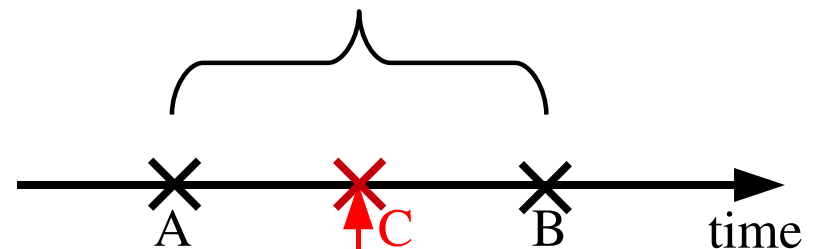
# Race Conditions

Int. Secure Systems Lab  
Technical University Vienna

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



Programmer



Attacker

# Shared Memory

*Int. Secure Systems Lab  
Technical University Vienna*

- 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)
- 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 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 so far!");
    }
}
```

- 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):

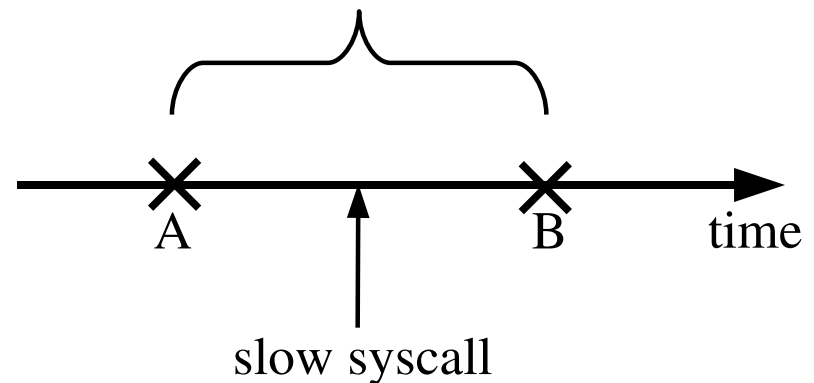
- Is not atomic
- can be interrupted at any time for arbitrary amounts of time



Programmer

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

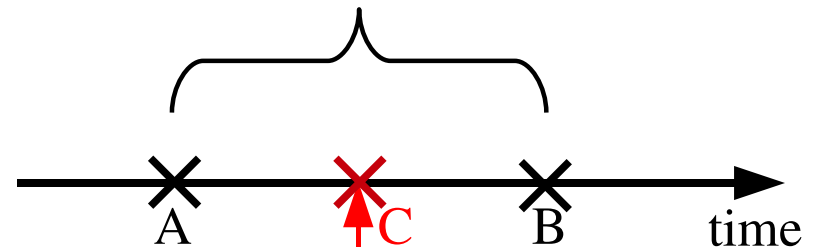
Int. Secure Systems Lab  
Technical University Vienna

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



Programmer

- $(t_A, t_B)$  is the window of vulnerability



Attacker

# Race Conditions

*Int. Secure Systems Lab  
Technical University Vienna*

- 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": won't stop a determined attacker
  - "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

*Int. Secure Systems Lab  
Technical University Vienna*

- 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

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

*Int. Secure Systems Lab  
Technical University Vienna*

- 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

*Int. Secure Systems Lab  
Technical University Vienna*

- 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);
}
```

- Attack

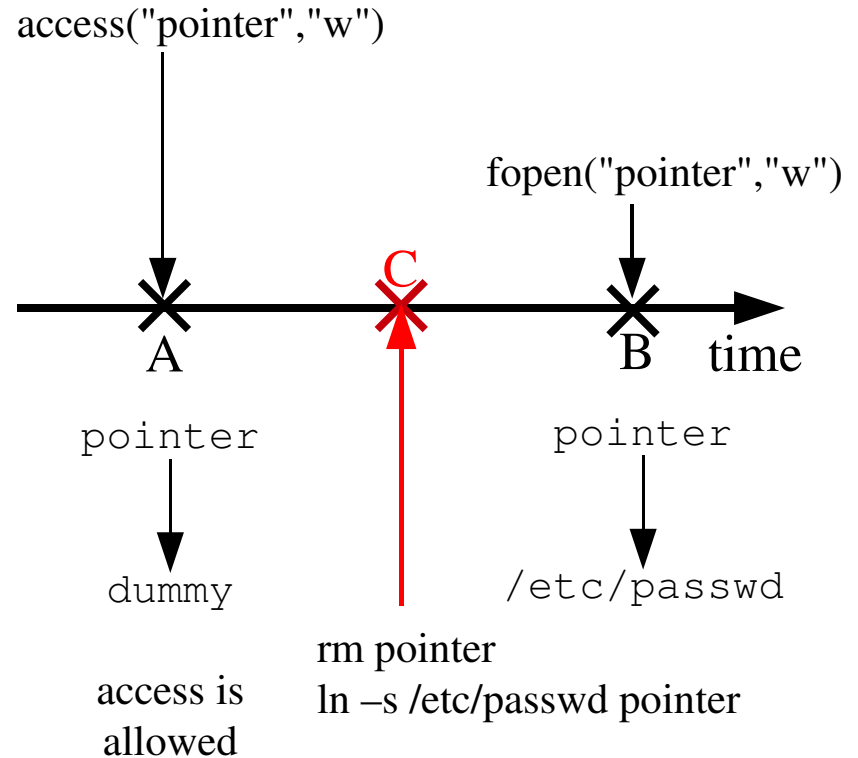
```
$ touch dummy; ln -s dummy pointer
```

```
$ 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
```



# TOCTOU Examples

# script execve Race

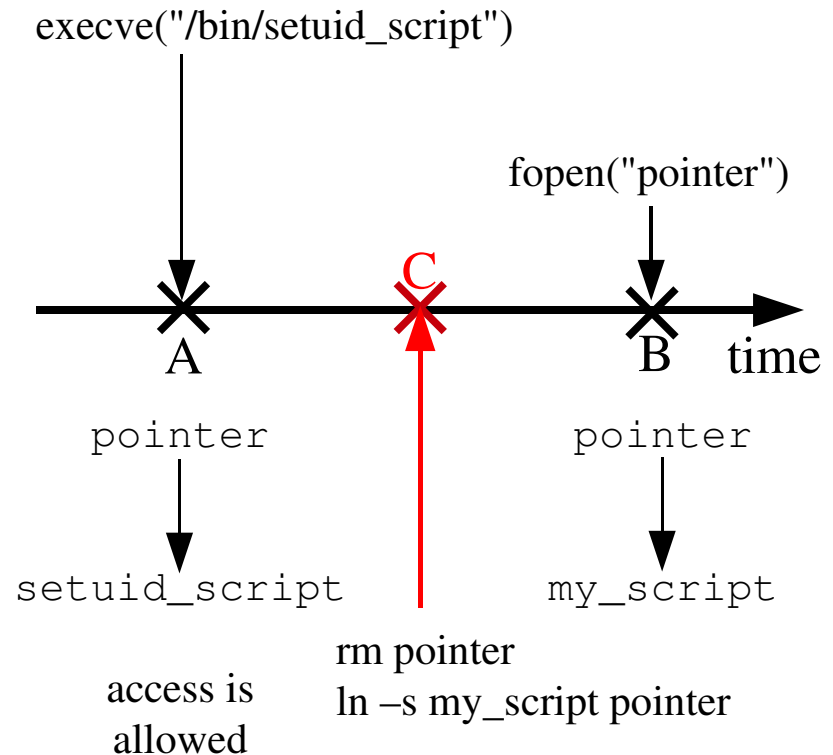
*Int. Secure Systems Lab  
Technical University Vienna*

- 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 normally not allowed on scripts!
  - although there are some work-arounds

# script execve Race

Int. Secure Systems Lab  
Technical University Vienna

- A: program interpreter is started (with root privilege)
- 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

*Int. Secure Systems Lab  
Technical University Vienna*

- `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

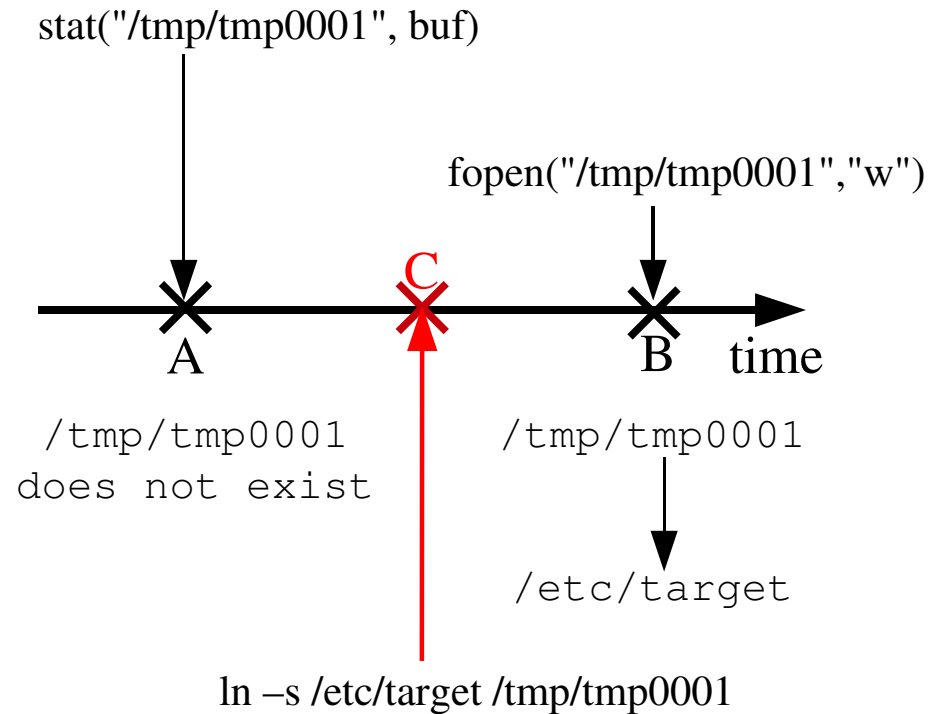
# Races on temporary files

*Int. Secure Systems Lab  
Technical University Vienna*

- 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"
  - **In -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"
  - /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

*Int. Secure Systems Lab  
Technical University Vienna*

- “Secure” procedure
  - pick hard to guess filename (randomize part of name)
  - set umask appropriately (0066 is usually good)
  - **atomically** test for existence 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)`
  - 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

*Int. Secure Systems Lab  
Technical University Vienna*

- File meta-information
  - chown(2) and chmod(2) are unsafe
  - 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

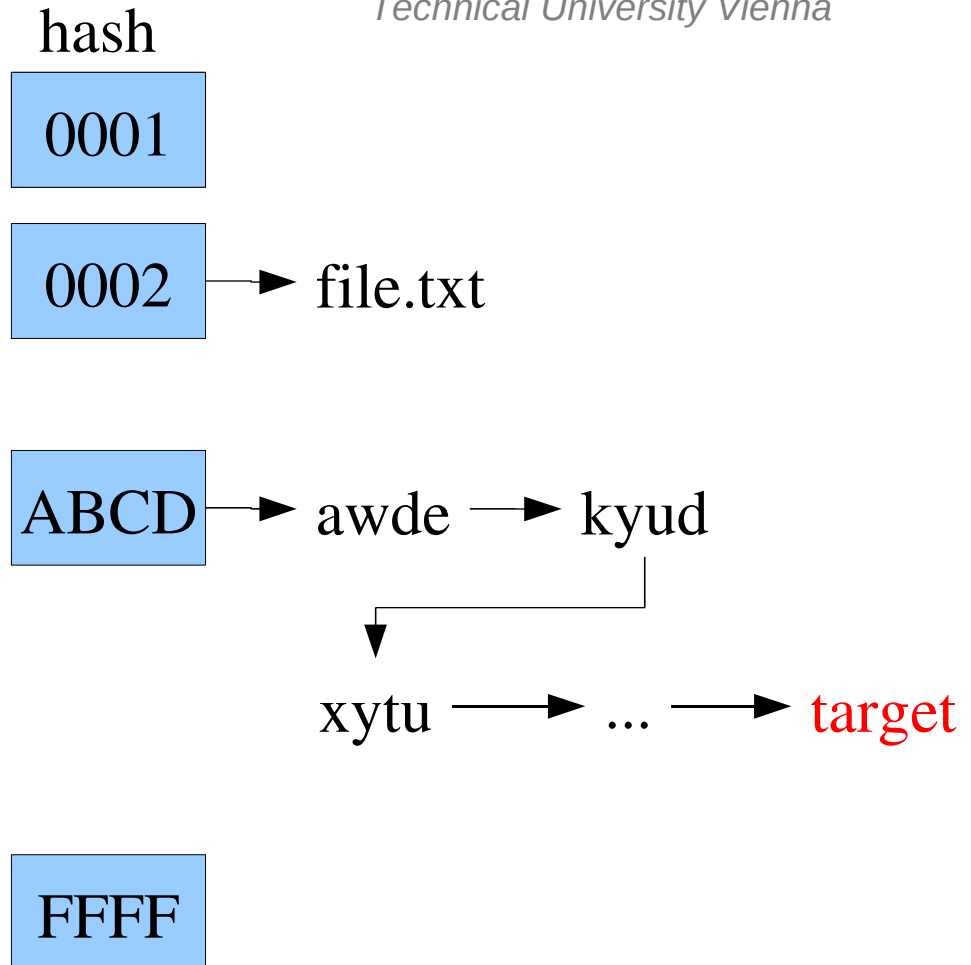
# Beating the odds

*Int. Secure Systems Lab  
Technical University Vienna*

- Window of vulnerability can be short
- Attacker can try to make the program run more slowly
- Example: filename lookups
  - very deep path names
  - many files in a directory
  - looking up the file in the FS will take longer!
- Computational complexity attacks
  - many algorithms are fast on average
  - ...but are slow in some corner cases

# 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!



# Computational Complexity Attacks

*Int. Secure Systems Lab  
Technical University Vienna*

- Accessing an item in a hash table of  $N$  elements is  $O(1)$  most of the time
- Worst case complexity is  $O(N)$ !
- Worst case does not occur accidentally (very unlikely)
- Attacker can make worst case happen

# Detection and Prevention

# Prevention

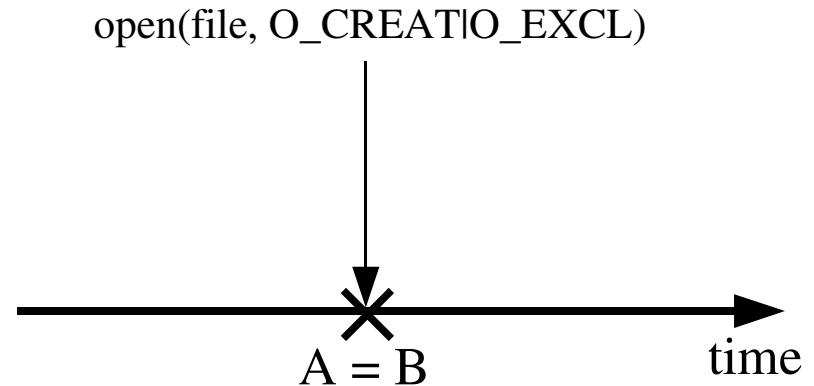
*Int. Secure Systems Lab  
Technical University Vienna*

- “Handbook of Information Security Management” suggests
  1. increase number of checks
  2. move checks closer to point of use
  3. immutable bindings
- Only number 3 is acceptable!
- Immutable bindings
  - operate on file descriptors
  - 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

*Int. Secure Systems Lab  
Technical University Vienna*

```
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 a user, assume his identity
  - let the operating system check permissions!
- Also need to drop group privileges (setegid)
- Check seteuid for errors!

# 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
- Simple method
  - open file using the `O_EXCL` flag

# Locking

*Int. Secure Systems Lab  
Technical University Vienna*

- Problem
  - NFS up to version 2 does not support O\_EXCL
  - multiple processes can capture the lock
- Solution (man page for open(2))
  - create unique file on file system (e.g., using host name)
  - use link(2) to make a link to lock file
  - when link(2) succeeds, then the locking operation was successful
- POSIX record locks
  - using fcntl(2) calls
  - can lock portions of files, and are automatically removed on process exit

# Non FS Race Conditions

*Int. Secure Systems Lab  
Technical University Vienna*

- Linux / BSD kernel ptrace(2) / execve(2) race condition
- ptrace(2)
  - debugging facility
  - used to access other process' registers and memory address space
  - can only attach to processes of same UID, except being 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

*Int. Secure Systems Lab  
Technical University Vienna*

- 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)

# Signal Handler Race Conditions

*Int. Secure Systems Lab  
Technical University Vienna*

- Signals
  - used for asynchronous communication between processes
  - signal handler can be called in response to multiple signals
  - signal handler must be written re-entrant or block other signals
- Example
  - sendmail up to 8.11.3 and 8.12.0.Beta7
  - syslog(3) is called inside the signal handler
  - race condition can cause heap corruption because of double free vulnerability

# Non-FS Race Conditions

*Int. Secure Systems Lab  
Technical University Vienna*

- 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

*Int. Secure Systems Lab  
Technical University Vienna*

- Static code analysis
  - specify potentially unsafe patterns and perform pattern matching on source code
- source code analysis and model checking
  - MOPS (MOdel-checking Programs for Security properties)

# Detection

*Int. Secure Systems Lab  
Technical University Vienna*

- 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

*Int. Secure Systems Lab  
Technical University Vienna*

- We looked at Race Conditions today
  - Overview
  - TOCTOU
  - Examples
  - Complexity attacks
  - Prevention and Detection
- Good luck with Challenge 6
- See you next week!