

Race Conditions

- Parallel execution of tasks
 - multi-process or multi-threaded environment
 - multi-user
 - tasks can interact with each other
- Three properties are necessary for a race condition to exist:
 - **Concurrency:** There must be at least two control flows executing concurrently.
 - **Shared Object:** A shared race object must be accessed by both of the concurrent flows.
 - **State Change:** At least one of the control flows must alter the state of the object of a race
- Results of tasks depend on the relative timing of events
 - Non-deterministic behavior

Race Conditions: Basics

- Programmer views a set of operations as atomic
 - In reality, atomicity is not enforced
 - Scheduler can interrupt a process at any time
 - Even more likely if there is a blocking system call
- Attacker can take advantage of this discrepancy
- Race condition vulnerabilities typically arise when:
 - checking for a given privilege, and
 - exercising that privilege
- Race conditions are eliminated by making conflicting operations mutually exclusive

TOC(T)TOU: Time-Of-Check- (To)-Time-Of-Use

- **Check** – Establish some precondition (invariant), e.g., access permission
- **Use** – Operate on the object assuming that the invariant is still valid
- Can occur in any concurrent system:
 - shared memory (or address space)
 - file system
 - signals

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 Race: 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);  
    }  
}
```

Looks atomic (1 line of code!)

- It's not!

Simple race:

- 2 threads read count
- both write count+1
- missed 1 increment

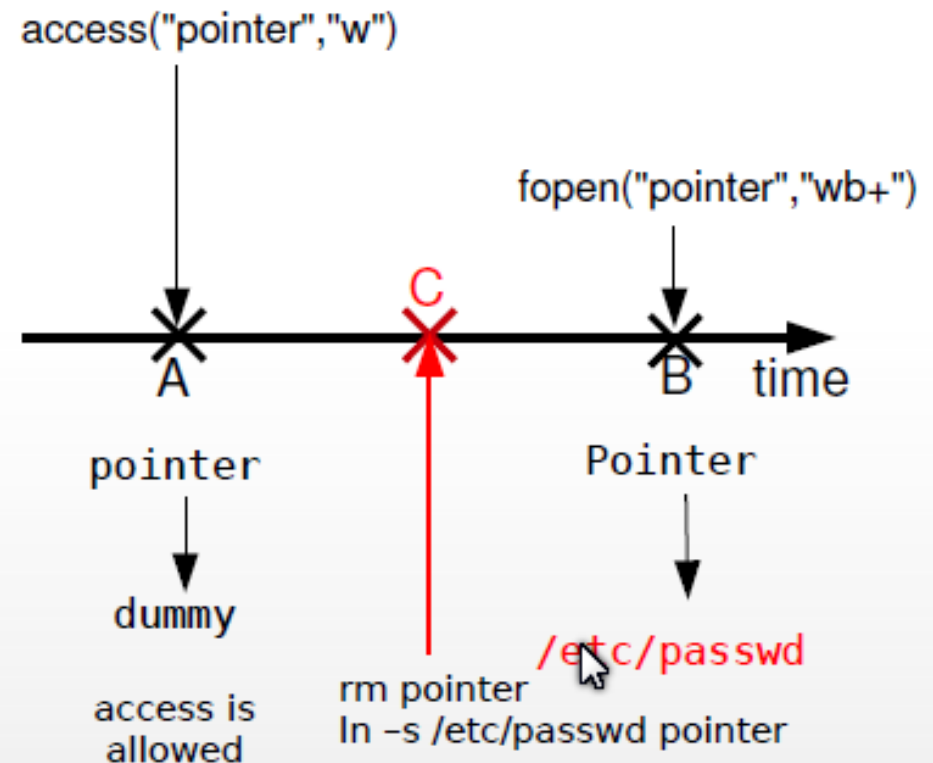
UNIX File System Security

- Access control: user should only be able to access a file if he has the permission to do so
- But what if user is running as setuid-root?
 - E.g., a printing program is usually setuid-root in order to access the printer device
 - Runs “as if” the user had root privileges
 - But a root user can access any file!
 - How does the printing program know that the user has the right to read (and print) any given file?
- UNIX has a special `access()` system call

Unix File System: Access/Open Race

```
/* 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);
}
```

```
$ touch dummy; ln -s dummy pointer
$ rm pointer; ln -s /etc/passwd pointer
```

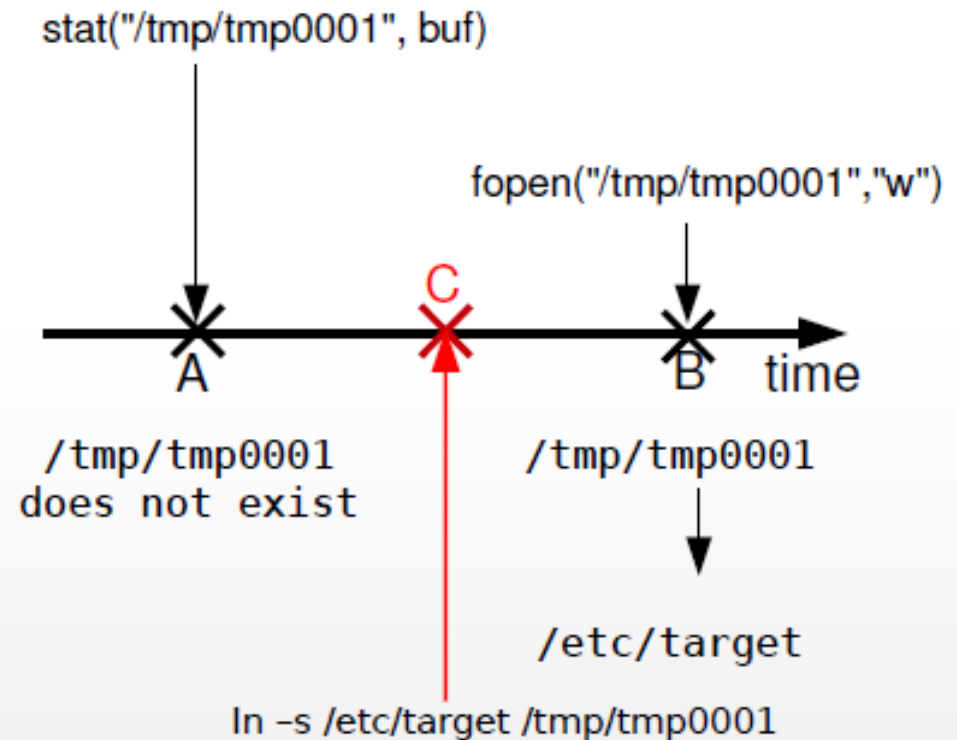


Races on temporary files

- 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
- A possible 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`
 - `/etc/target` is created!



- Attack:

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

Unix File System: Script Execve Race

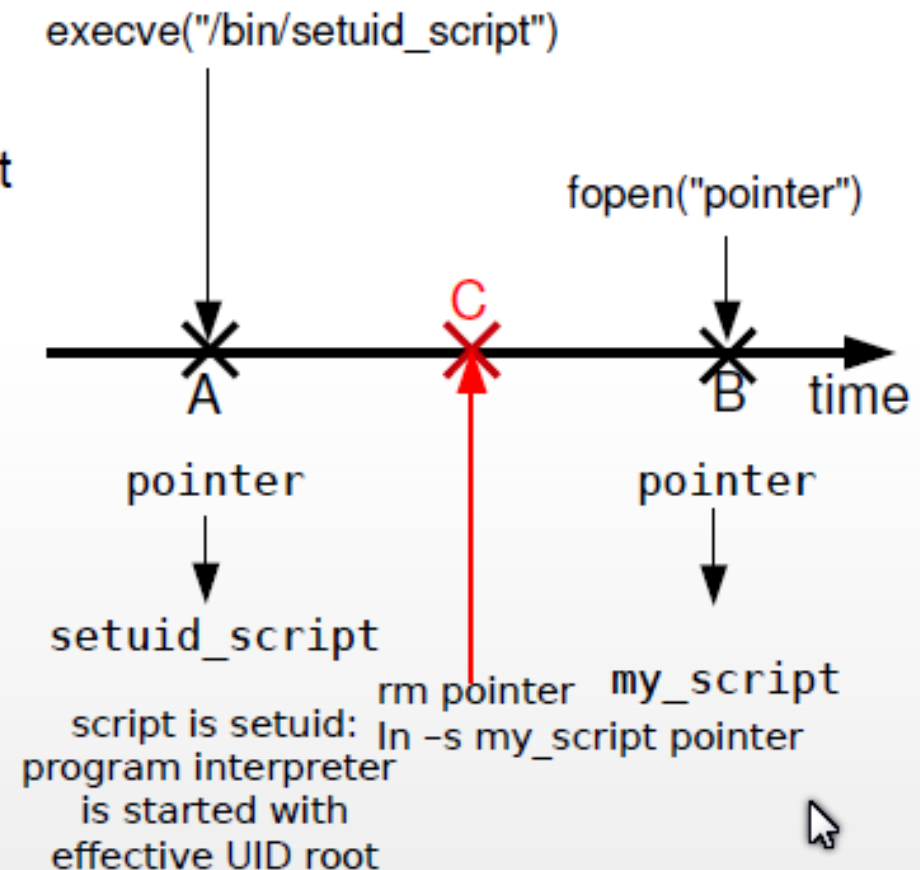
- Filename redirection
 - soft links again
- Setuid Scripts
 - `execve()` system call invokes `seteuid()` call prior to executing program
 - A: program is a script, so command interpreter is loaded first
 - B: program interpreter (with root privileges) is invoked on script name
- attacker can replace script content between step A and B
- Setuid not allowed on scripts on most platforms!
 - Some work-arounds

Unix File System: 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
```



Threaded programs: Use-after-free

- Thread #3 gives access to protected resources

- Thread 1

```
extern int * a;  
a = malloc(10);  
// Launch Thread 2
```

```
if(some_error)  
    free(a);
```

- Thread 2

```
extern int *a;
```

```
/* is password checked  
?*/
```

```
if(a[0])
```

```
    /* do passwd  
    protected stuff */
```

- Thread 3

```
/* same memory  
   block allocated  
   */
```

```
X=malloc(10);
```

```
X[0]=1;
```

Window of Vulnerability

- 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!
 - Repeated attempts
 - Attacker can try to slow down the victim machine/process to improve the odds (high load, computational complexity)
 - 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

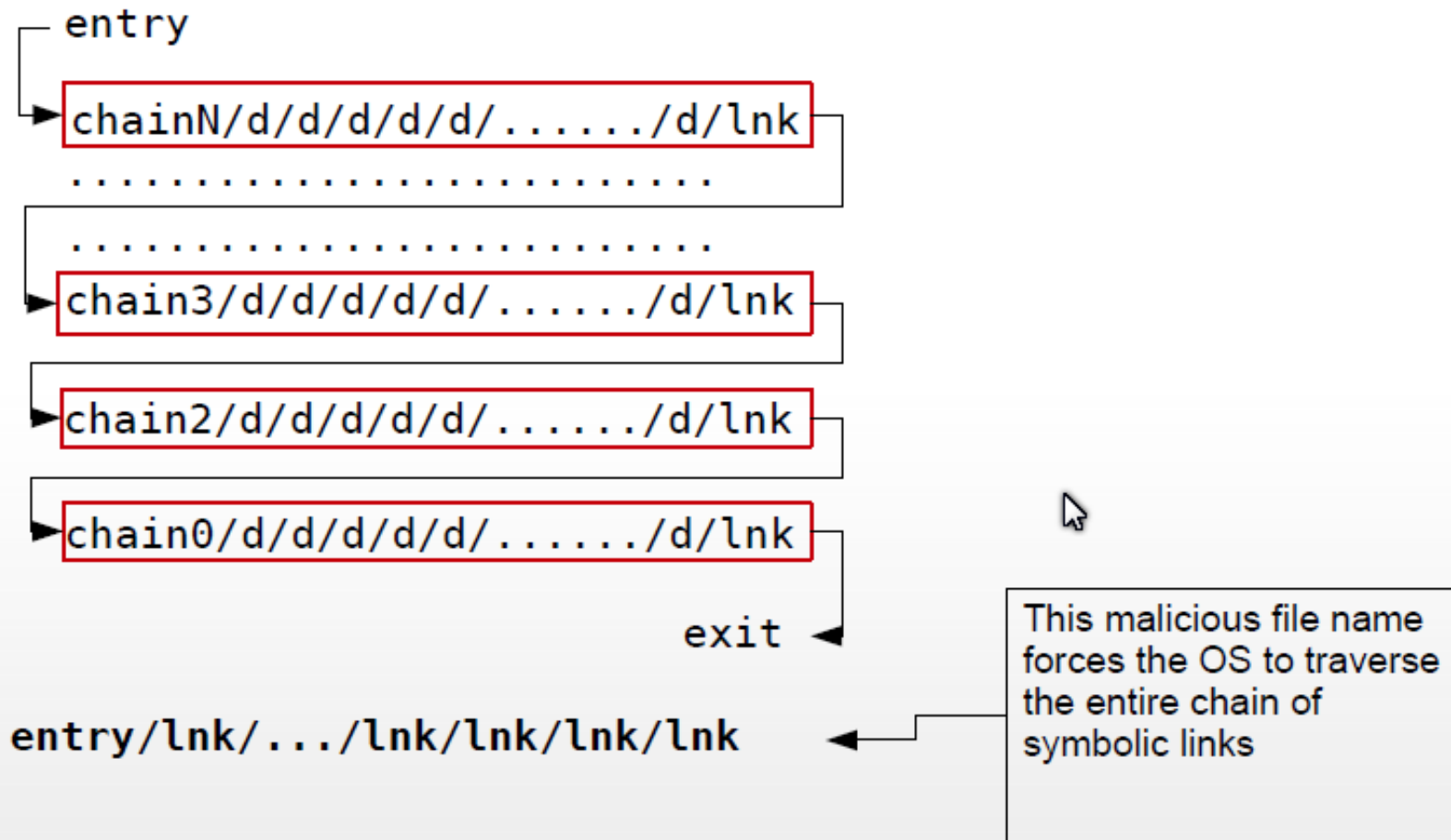
Slow file lookups

- 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.txt in final directory
- Limit to length of a file name:
 - MAXPATHLENGTH (4096 on Linux)
 - Max depth of ~2000

Making It Slower: 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 Linux box
- Total file system lookups:
 - follow 40 chains...
 - ...each with 2000 nested directories
 - 80000 lookups!

File System Maze



Prevention and Detection

- Prevention: many solutions depending on actual race
 - OS specific solutions: ID or filename related
 - Forking: delegate operations to separate process with EUID (effective UID)
 - Locking: suppress race, but slows down process
 - Hardness amplification: Reduce success probability of attacker (k-races, pseudo-atomic transactions)
- Detection:
 - Static analysis with pattern matching
 - Static analysis with model checking (MOPS, RacerX, rccjava)
 - Dynamic Analysis (Eraser)

Software exploitation: the bigger perspective

- Software Fault Injection
 - Software built for one purpose, but attacker misuses the software for another purpose
 - Notably through specifically crafted inputs
 - Any Turing machine can be exploited
- Hardware Fault Injection
 - Don't forget that software runs within hardware
 - Perturbating the execution environment during code execution (laser, power supply glitch, clock glitch)
 - Cosmic/Gamma rays lead to random errors (bit flips)
 - Particular memory access patterns lead to bit errors in DRAM