Dynamic Detection and Prevention of Race Conditions in File Accesses

Eugene Tsyrklevich

eugene@securityarchitects.com

Outline

- What are race conditions?
- How can we prevent them?
- Implementation description
- Demonstration

What are Race Conditions?

- File race conditions occur when file operations are not carried out atomically

- An operation/transaction is carried out atomically when it executes without being interrupted or does not execute at all

Race Condition Example #1

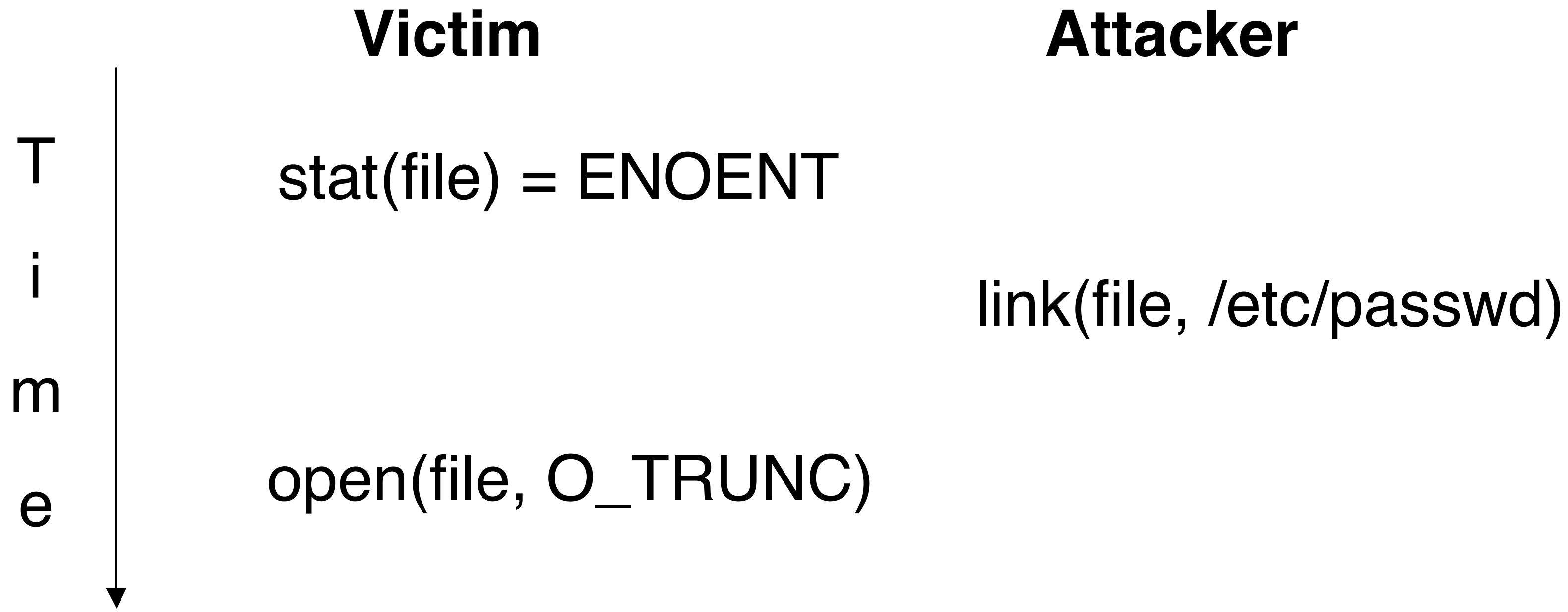
Time

Victim	Attacker
access(file, W_OK)	

	unlink(file)
	link(file, /etc/passwd)

open(file, O_TRUNC)	

Race Condition Example #2



Victim

Attacker

T i m e

stat(file) = ENOENT

link(file, /etc/passwd)

open(file, O_TRUNC)

Other Race Conditions

- Other types of file race conditions:
  - Directory operations (GNU fileutils)
  - Setuid shell scripts (Early Unices)
  - Temporary files (all Unix programs that use temporary files? :-)

Why are RC dangerous?

- File race conditions are
  - Still constantly being discovered
  - Hard to find

- Race conditions can be used for
  - Privilege elevation
  - Denial of service

Related Work

- Various static analysis tools
- RaceGuard (Crispin Cowan, et al)
  - Addresses /tmp stat races only
- Openwall Project (Solar Designer)
  - Limits users from following untrusted symbolic links created in certain directories
  - Limits users from creating hard links to files they don't have read and write access to

The Problem

Programmers assume that sequences of file operations execute in isolation

Transactions

- Model filesystem activity in terms of transactions
  - access() + open() operation is a pseudo-transaction
- Race conditions violate transaction ACID (Atomicity, Consistency, Isolation, and Durability) properties

Transactions (2)

- Race conditions in file accesses primarily violate the isolation property
- Enforcing isolation in pseudo-transactions requires
  - detection
  - prevention of race conditions

Detecting Race Conditions

- Mediate all file operations

- Look for explicit attacks
  (Default allow policy)

Or

- Look for normal file activity
  (Default deny policy)

Default Allow Policy

- Look for explicit attack patterns

```
REMOVE=UNLINK | RMDIR | RENAME
DENY(ACCESS, REMOVE)
DENY(CHDIR, REMOVE)
DENY(EXEC, REMOVE)
```

Default Deny Policy

- Look for normal file activity

```
OPEN_RW = OPEN_READ | OPEN_WRITE
RENAME = RENAME_TO | RENAME_FROM

PERMIT(OPEN_RW, OPEN_RW | ACCESS | UTIMES | CHDIR | EXEC |
UNLINK | READLINK | CHMOD | CHOWN | RENAME)
PERMIT(OPEN_CREAT, OPEN_RW | ACCESS | UTIMES | CHDIR | EXEC |
RENAME_FROM)
PERMIT(ACCESS, OPEN_RW | ACCESS | UTIMES | CHDIR |EXEC)
PERMIT(EXEC, OPEN_READ | EXEC)
PERMIT(CHDIR, OPEN_READ | CHDIR | ACCESS | READLINK)
PERMIT(RENAME_FROM, OPEN_RW | ACCESS | UNLINK | RENAME_FROM)
PERMIT(RENAME_TO, OPEN_RW)
PERMIT(CHMOD | CHOWN, OPEN_RW | ACCESS | CHMOD | CHOWN)
PERMIT(UTIMES, OPEN_RW | ACCESS | CHMOD | CHOWN)
PERMIT(READLINK, READLINK)
```

Preventing Race Conditions

- Transaction rollback
- User confirmation
- Locking out processes
- Killing processes
- Suspending processes

Transaction Rollback

- Pros
  - Leaves system in a consistent state
- Cons
  - Requires transaction support which few operating systems provide

User prompting

- Pros
  - Less intrusive
- Cons
  - Difficult usability problem
  - Not suitable for servers

Locking out processes

- Pros
  - Guarantees race condition free environment
- Cons
  - Possible deadlocks
  - Poor performance

Killing processes

- Pros
  - Prevents any possible abuse
- Cons
  - Subject to denial-of-service attacks

Suspending processes

- Pros
  - The worst possible outcome (in case of a false positive) is a process delay
- Cons
  - Difficult to decide when to wake up a sleeping process

Suspending Processes (2)

T
i
m
e

Victim	**Attacker**
access(file) (starts new pseudo transaction X)	

	unlink(file)
	link(file, /etc/passwd)
	(interferes with transaction X)
	SUSPEND

open(file, O_TRUNC) (ends transaction X, starts transaction Y)	
	(wake up and execute unlink())

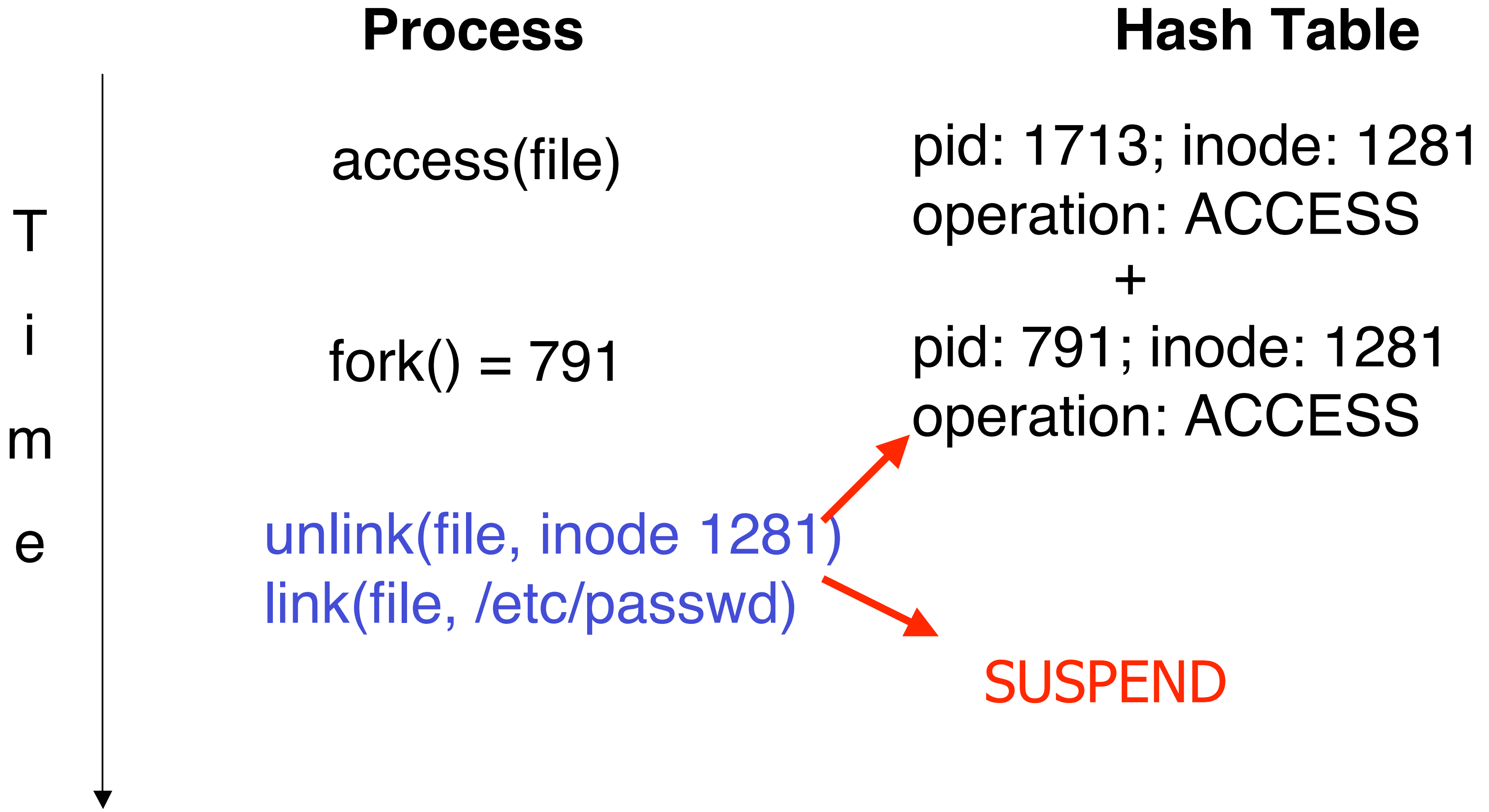
Implementation

- OpenBSD kernel module
- Mediates filesystem calls + fork, exec and exit
- Records all file operations in a global hash table

Implementation (2)

- Load average is used to calculate the timeout for
  - suspending processes
  - purging old hash entries

Implementation Example



Process

Hash Table

Time

access(file)

pid: 1713; inode: 1281
operation: ACCESS

+

fork() = 791

pid: 791; inode: 1281
operation: ACCESS

unlink(file, inode 1281)
link(file, /etc/passwd)

SUSPEND

Microbenchmarks

System Call	open	stat	fork
Stock Kernel, ms	2.55	3.28	86.17
Race Protection Kernel, ms	5.69	3.38	86.21
Total CPU Overhead (%)	123	3	0

Compile Benchmark

	Real Time	User Time	System Time
Stock Kernel, sec	427	363	37
Race Protection Kernel, sec	436	363	43
Total CPU Overhead (%)	2	0	16

Results

- Used on several machines over a period of three months
- No noticeable system overhead
- No false positives or false negatives after the initial policy adjustment (i.e. system training)

Demonstration

- Live Demo

Thank You

Source code is available at www.secarch.com/people/eugene/

eugene@securityarchitects.com