Study of Race Condition: A Privilege Escalation Vulnerability

Tanjila Farah, Rashed Shelim
Department of Electrical & Computer Engineering, North South University
Dhaka, Bangladesh

and

Moniruz Zaman, Delwar Alam
Department of Software Engineering, Daffodil International University
Dhaka, Bangladesh

ABSTRACT

The Race condition is a privilege escalation vulnerability that manipulates the time between imposing a security control and using services in a UNIX like system. This vulnerability is a result of interferences caused by multiple sequential threads running in the system and sharing the same resources. Race condition could occur due to sequence condition imposed by un-trusted processes or locking failure condition imposed by secure programs such as operating systems. The race condition is a common vulnerability in UNIX-like systems, where directories such as /tmp and /var/tmp are shared between threads. A study of Race condition vulnerability and its impact in UNIX like systems are presented in this paper. Also various types of Race condition attack and there detection, avoidance and prevention techniques are also discussed in this paper.

Keywords: Race Condition, Vulnerability, Privilege Escalation, Critical Section, Dirty COW, Semaphore.

1. INTRODUCTION

One very imperative aspect of performance of parallel computing is shared-memory. In various parallel programming models, processes/threads share a common address space, which they read and write to asynchronously [1]. In this model all processes have equal access to shared memory. Various mechanisms such as locks and semaphores may be used to control access to the shared memory. In parallel programming a program is a collection of threads where each thread has a set of private variables (i.e. local stack variables) and shared variables (i.e. static variables, shared common blocks, or global heap) [2]. Threads communicate by writing and reading shared variables and coordinate by synchronizing on shared variables.

A race condition occurs in a shared memory program when two threads access the same variable using shared memory data, and at least one thread executes a write operation. The accesses are concurrent (not synchronized) so they could happen simultaneously [12]. Race conditions may occur when shared data accesses are not synchronized properly but the execution result depends on the order of threads [3]. Access to shared memory data access is controlled by critical sections. Yet without synchronization, shared variables accessed through critical section may be corrupted by threads.

All systems comprising multiprocessing environment are vulnerable to Race condition attack. Race condition is also known as Time of check/time of use (TOC/TOU) binding flaw, Concurrency attacks, or Threadjacking [3]. Typical race condition attacks involve opening and validating a file, running a subprogram, checking a password, or verifying a username and more.

This paper attempts to present a study of the race condition vulnerability and its various exploits. Various detection, avoidance, and preventions techniques of this attack are also discussed in this paper. The paper is organized as follow. A literature review on race condition attack is presented in section 2. In Section 3 various key points of race condition are discussed. In section 4 an analysis of the process of race condition attack is presented. The detection and avoidance mechanism of Race condition is discussed in Section 5. Various exploits of race condition are discussed in Section 6. The prevention mechanism of race condition and its issues are discussed in Section 7. We conclude in Section 8.

2. LITERATURE REVIEW

Several papers [1], [2], [3] have defined Race condition. The following work [4], [5], [7], [8] deal with race condition detection techniques. Papers [7], [9] deal with possible solutions of race condition. In [3] authors explain the time to check time to use (TOC/TOU) method in the critical section. They also propose a run-time model that detects TOC/TOU binding flaws on the file space. In [4] presents an algorithm for dynamically detecting race conditions in programs. This algorithm computes the order in which synchronization operations are guaranteed to have occurred. In [7] authors provide formal definition of direct and indirect information flow. In [8] authors defined anomalies of race conditions that are causing concealing deadlocks and results in hangs. None of the previous works discussed various exploits of race condition and their techniques and effects in popular application. Our goal is to provide an analysis of various exploits of race condition. We also discuss the coding of these various exploits and there detection and prevention techniques.

3. RACE CONDITION ITS ELEMENTS

A race condition is a special condition that may occur inside a critical section handling multithread [1]. When the results of multiple threads executing in critical section differ depending on the sequence in which the threads are being execute, the critical section is said to contain a race condition. In this section, the elements of race condition and the process of race condition itself is discussed.

Thread
Thread is a process of separating the different applications that are executing at time in a computer. It is a program's path of execution. Threads execute processes. The operating system assigns processor time to threads for the execution of its tasks

[3]. A single process may contain multiple threads of execution. Threads maintain their own exception handlers, scheduling priorities, and a set of structures that the operating system uses to save the thread's context. Threads are different then processes because processes usually cannot directly share memory and data structures while threads can.

Multithread

Systems require multiple processes to run at the same time such as drawing pictures while reading keystrokes [4]. An operating system that allows multitasking creates the effect of simultaneous execution of multiple threads from multiple processes by dividing the available processor time among the threads in process[7]. This is known as multithreading. These threads share the process resources but execute independently. The operating system allocates a processor time to each thread one after another. The currently executing thread is suspended when it's time ends and another thread resumes. The threads are executed within the same program and thus reading and writing the same memory simultaneously.

Critical Section

A Critical Section of a program is where global shared memory is being accessed. It is a section of code that is executed by multiple threads [12]. Here the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section [8]. While executing multiple threads inside the same application, multiple threads might access (i.e. read, write) same resources. This might initiate race condition vulnerability for the system [9]. Figure 1 shows a subtraction function java code for critical condition.

```
public class Example {
    protected long count = 0;
    public void sub(long value){
       this.count = this.count - value;
    }
    }
```

Figure 1: A critical section Java code

If two threads, X and Y, are executing the sub() method on the same instance of the Example class it will not be possible to know when the operating system switches between the two threads. The code in the sub() method is executed a set of smaller instructions as shown in Figure 2.

```
this.count = 50;

   X:  Reads this.count into a register (50)
   Y:  Reads this.count into a register (50)
   Y:  Sub value 2 to register
   Y:  Writes register value (48) back to memory. this.count now
equals 48
   X:  Sub value 5 to register
   X:  Writes register value (45) back to memory. this.count now
equals 45
```

Figure 2: Set of instructions for Subtract function

The two threads wanted to subtract the values 2 and 5 to the Example. In Figure 2 both threads read the value 50 from memory and attempted to subtract 2 and 5 from 50 thus the value should have been 43 after the threads are executed. However, since the execution of the two threads is interleaved, the result became 45. Thus thread X was ignored in the process. Inside a critical section process has exclusive access to shared modifiable data. To reduce any possibility of errors in critical section semaphore is introduced.

4. RACE CONDITION ATTACK

Race condition manipulates the time to check/ time to use (TOC/TOU) time gap between the threads in the critical section and thus create disorientation in the shared data [3]. The example of race condition in a transaction scenario is shown in Figure 3. In a regular transaction only one request is executed in one thread. In Race condition attack multiple request is processed is processed on the same shared data as shown in Figure 3. In the scenario a two withdraw request is sent in same thread using the same shared data and only the second transfer information was saved in the system [9]. This happened because the time TOC between TOU for request1 is long enough that request2's TOC and TOU started processing.

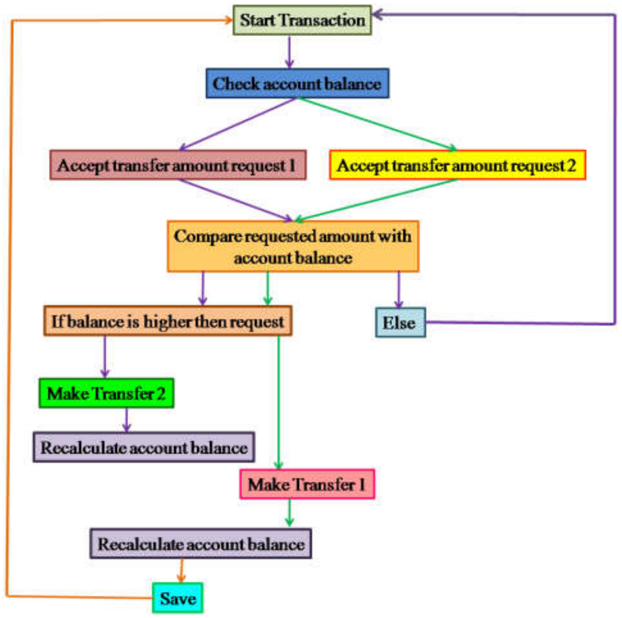
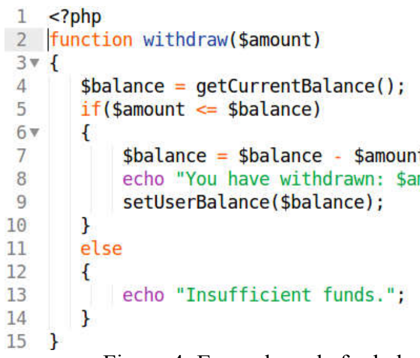

Start Transaction
Check account balance
Accept transfer amount request 1
Accept transfer amount request 2
Compare requested amount with account balance
If balance is higher then request
Else
Make Transfer 2
Recalculate account balance
Make Transfer 1
Recalculate account balance
Save

Figure 3: A race condition example

Usually two requests in the same thread won't take place, yet if huge number of request is send to the system within a very short time, the TOC/TOU of two threads will overlap at some point create race condition possible.

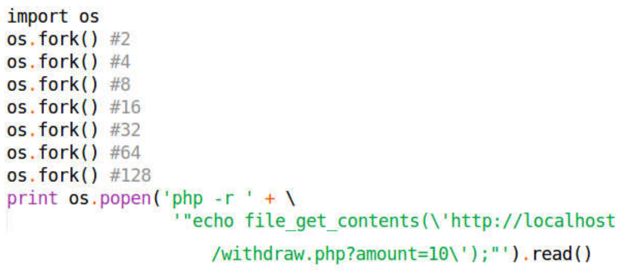
Code analysis: In this section we will analyze an example race condition code. Consider the following PHP code in Figure 4 for withdrawing money or credits from online account. For getting current balance, getCurrentBalance() function and for saving balance setUserBalance() function is used. getCurrentBalance() will check if there is enough money to withdraw ,and if so it will process the withdraw request and decrees the money. But this function takes less than half a millisecond to run and race condition exploit can be executed with in this half millisecond which will allow the attacker to withdraw money without decreasing the total balance.



```
<?php
function withdraw($amount)
{
    $balance = getCurrentBalance();
    if($amount <= $balance)
    {
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
        setUserBalance($balance);
    }
    else
    {
        echo "Insufficient funds.";
    }
}
```

Figure 4: Example code for balance transfer

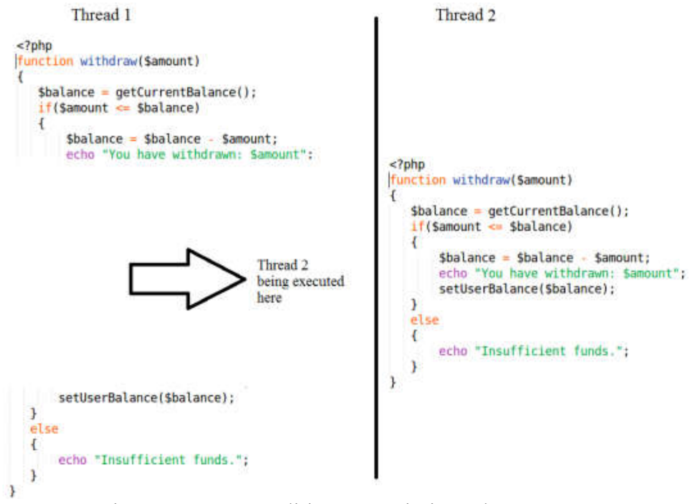
This above code section is our critical section. To implement race condition attack simultaneous requests needed to be sent to the system. By sending simultaneous requests it is anticipated that at least one request will reach the system within the half millisecond processing time. Python script shown in Figure 5 sends 128 simultaneous requests to the system for withdrawing $100 each time.



```
import os
os.fork() #2
os.fork() #4
os.fork() #8
os.fork() #16
os.fork() #32
os.fork() #64
os.fork() #128
print os.popen('php -r ' + \
                '"echo file_get_contents(\'http://localhost
                    /withdraw.php?amount=10\');"').read()
```

Figure 5: Example python script for representing attack

So if the starting balance is $30,000. After withdrawing 128 * $100 =$12800 remaining balance should be $17200. But due to race condition the final amount of money will not be $17200. More Credit will be left in user account as web server will execute queries asynchronously. Two request of executed in the same time will be interleaved by the operating system's CPU time sharing system. As a result, after processing 128 requests only $100 will be deducted from the user account. This is because the getCurrentBalance () function will be executed before calling setUserBalance($balance) function from the previous request. This process is shown in Figure 6.



```
<?php
function withdraw($amount)
{
    $balance = getCurrentBalance();
    if($amount <= $balance)
    {
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
```

```
<?php
function withdraw($amount)
{
    $balance = getCurrentBalance();
    if($amount <= $balance)
    {
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
        setUserBalance($balance);
    }
    else
    {
        echo "Insufficient funds.";
    }
}
```

```
        setUserBalance($balance);
    }
    else
    {
        echo "Insufficient funds.";
    }
}
```

Figure 6: Race condition scenario in code

5. DETECTION AND AVOIDANCE OF RACE CONDITION

The most common symptom of a race condition is unpredictable values of variables that are shared between multiple threads. Race conditions generally involve one or more processes accessing a shared resource (such a file or variable), where this multiple access has not been properly controlled. Due to the control issue a process may interrupt another running process between essentially any two instructions. Race condition could occur due to interference caused by un-trusted processes and interference caused by trusted processes.

Race condition caused by interference from un-trusted process is also known as “sequence” or “non-atomic” condition. This condition is caused by processes running various programs at the same time, which accesses shared memory between steps of the secure program. Loading and saving a shared variable are usually implemented as separate operations and are not atomic. So if the variable memory is shared the other process may interfere. Secure programs must determine if a request should be granted, and if so, act on that request. There must be no way for an un-trusted user to change anything used in this determination before the program acts on it. In Linux like systems the filesystem is a shared resource used by many programs, and some programs may interfere with its use by other programs. Secure programs should generally avoid using access(2) to determine if a request should be granted, followed later by open(2), because users may be able to move files around between these calls, possibly creating symbolic links or files of their own choosing instead [14]. A secure program should instead set its effective id or filesystem id, and then make the open call directly. It's possible to use access(2) securely, but only when a user cannot affect the file or any directory along its path from the filesystem root. Regular programs can become security weaknesses if files are not created properly. For another example, when performing a series of operations on a file's meta-information (such as changing its owner, stat-ing the file, or changing its permission bits), first open the file and then use the operations on open files. This means use the fchown( ), fstat( ), or fchmod( ) system calls, instead of the functions taking filenames such as chown(), chgrp(), and chmod(). Doing so will prevent the file from being replaced while your program is running (a possible race condition).

Race conditions caused by trusted processes are also known as deadlock, livelock, or locking failure conditions. These are conditions caused by processes running the ``same'' program. Since multiple processes may have the ``same'' privileges, they might interfere each other due to lack of proper control. Unix-like systems resource locking has traditionally been done by creating a file to indicate a lock, because this is very portable. Administrator privilege can easily to fix stuck locks. Stuck locks can occur because the program failed to clean up after itself. It's important that the programs which are cooperating using files to represent the locks use the same directory, not just the same directory name such as “/var/mail”. Another approach to locking is to use POSIX record locks, implemented through fcntl(2) as a ``discretionary lock'' [14].

6. VARIOUS EXPLOITS RACE CONDITION

Exploits based on race conditions are delicate. They typically require repeated attempts to be executed. Race condition exploits existed in Firefox 2007, Internet explorer 2011, Windows shortcut link 2010, and in Linux operating system as Dirty copy on write.

Firefox v2.0.0.10 race condition
Mozilla Firefox version older then 2.0.0.10 is vulnerable to race condition in the handling of the window.location property [13]. The referrer header of these older versions Mozilla Firefox are set to the window or frame in which script is running, instead of the address of the content that initiated the script, which allows remote attackers to spoof HTTP referrer headers and bypass referrer-based CSRF protection schemes by setting window.location and using a modal alert dialog that causes the wrong referrer to be sent.

Race condition in Pulse Audio
Pulse Audio is a network-enabled sound server is Linux operating system. Race condition in PulseAudio 0.9.9, 0.9.10, and 0.9.14 allows local users to gain privileges via vectors involving creation of a hard link, related to the application setting LD_BIND_NOW to 1, and then calling execv on the target of the /proc/self/exe symlink [13]. A user who has write access to any directory on the file system containing /usr/bin can exploit the race condition vulnerability to execute arbitrary code with root privileges.

Windows Shortcut Link
Windows Shell in Microsoft Windows XP SP3, Server 2003 SP2, Vista SP1 and SP2, Server 2008 SP2 and R2, and Windows 7 allows local users or remote attackers to execute arbitrary code via a crafted (1) .lnk or (2) .pif shortcut file, which is not properly handled during icon display in Windows Explorer [13]. The .lnk file is vulnerable to race condition as it executes the downloaded dll file 3 times simultaneously. This hinders the proper exploitation of the victim in case the payload dll tries to write any file on the disk or tries to access and change any other resource on the victim system.

Dirty copy on write (COW)
Dirty Copy on Write also known as Dirty COW is a Linux based Race condition vulnerability. This vulnerability allows attackers to escalate the file system protection of Linux Kernel, get root privilege and thus compromise the whole system [10]. The dirty COW exploit call the root file as a read only file. Using "mmap" function a new virtual space is created for the file to be changed and then MAP PRIVATE tag is used to create a copy of the original root file. Every time the exploit is ran, it will create a copy of the original file. This file is edited as please. The 'madviseThread' function is used to locate the memory address range assigned to the main root file. The "procselfmemThread" is used to trick the system in believing that the memory range of the main root file is empty and write the copy file [10]. Thus the access is gained.

Internet Explorer Race condition
Microsoft Internet Explorer 6 through 8 are vulnerable to Race condition attack [13]. These versions allow remote attackers to execute arbitrary code or cause a denial of service (memory corruption). This is also known as "Window Open Race Condition Vulnerability. An attacker compiles a web page with malicious code and when a user visits this page, the exploit happens. This is similar to cross site scripting attack.

Race condition vulnerabilities in the setuid-root/usr/ bin/at
This race condition exploit binary allows removing any file on the filesystem. "At" utility reads commands from standard input and groups them together as an "at-job", to be executed at a later time. Each "at-job" is kept in separate file in at spool directory. "At" jobs may be removed if "-r" option is used with a job-id parameter to the "at' command. However, there are two vulnerabilities within the code that removes "at" from at spool directory.At utility does not properly handle job ids specified as a parameter to the "-r" option. It allows removing jobs outside of "at" spool directory if relative path name is used. Only absolute path names are filtered out.

"At" verifies ownership of the file and limits the user to remove only its own "at". Unfortunately, a race condition occurs after "at" verifies the file and before the file is unlinked. By altering directory structure between these two system calls, "at" may be fooled to remove file other than it expects.Since this code is executed with full root privileges, these two vulnerabilities may allow unprivileged users to remove any files on the filesystem.

Linux kernel PTrace Race Condition
This exploit of Red Hat and affects Linux kernel up to 2.6.29 release. The vulnerability is located in ptrace_attach() routine of kernel/ptrace.c [11]. Here ptrace_attach() uses the current process' MUTEX to lock and serialize the two tasks (the current one, and the one passed to that function as an argument) as shown in Figure 7. However, the code incorrectly uses the current task's cred_exec_mutex instead of the task's to be traced. This creates a race condition which allows a user to ptrace(PTRACE_ATTACH) during the execution of an execve() call to a SUID binary [11]. But this new member (cred_exec_mutex) was added to the task_struct to avoid exactly this behavior. This can lead to local privilege escalation.

```
int ptrace_attach(struct task_struct *task)
{
     int retval;
      unsigned long flags;
     audit_ptrace(task);
     retval = -EPERM;
      if (same_thread_group(task, current))
            goto out;
      retval = mutex_lock_interruptible(&current->cred_exec_mutex);
    if (retval  < 0)
            goto out;
      retval = -EPERM;
repeat:
     ...
bad:      write_unlock_irqrestore(&tasklist_lock, flags);
     task_unlock(task);
     mutex_unlock(&current->cred_exec_mutex);
out:
     return retval;
}
```

Figure 7: Ptrace code with vulnerability

7. PREVENTION OF RACE CONDITION ATTACK

The typical solution to a race condition is to ensure that your program has exclusive rights to shared data while it's manipulating the process of gaining an exclusive right to the data is called locking [11]. Locks aren't easy to handle correctly. Semaphore system is used to synchronize the locks.

Semaphore
A Semaphore is a thread synchronization system that can be used either to send signals between threads to avoid missed

signals, or to guard a critical section [12]. A semaphore can act as a gate way to critical section. Status of the gate is either open (raised) or closed (lowered). If a thread wants to access data through critical section, it executes a P operation. If gate is open, process enters and closes gate behind it. Semaphore ensures only one thread get access to critical section data at a time. The java code for semaphore is shown in Figure 8.

```
public class Semaphore {
  private boolean signal = false;
  public synchronized void take() {
    this.signal = true;
    this.notify();  }
  public synchronized void release() throws
InterruptedException{
    while(!this.signal) wait();
    this.signal = false; }
}
```

Figure 8: Code for semaphore

Semaphore is also used to limit the number of threads allowed into a section of code. One common problem of lock is a deadlock in which programs get stuck waiting for each other to release a locked resource [12]. Most deadlocks can be prevented by requiring all threads to obtain locks in the same order.

Deadlock

A deadlock occurs when two threads each lock a different variable at the same time and then try to lock the variable that the other thread already locked [12]. As a result, each thread stops executing and waits for the other thread to release the variable. Because each thread is holding the variable that the other thread wants, nothing occurs, and the threads remain deadlocked. An example of deadlock is shown in Figure 9. In the example thread 1 locks A, and tries to lock B, and thread 2 has already locked B, and tries to lock A, a deadlock arises [12]. Thread 1 can never get B, and thread 2 can never get A. In addition, neither of them will ever know. They will remain blocked on each their object, A and B, forever. This situation is a deadlock.

```
Thread 1  locks A, waits for B
Thread 2  locks B, waits for A
```

Figure 9: A deadlock scenario

Race conditions exceptions are unchecked exceptions which occur only in the runtime, and cannot be detected by the compiler. In order to prevent them, synchronized block should be used around the shared resource to prevent multiple accesses at a time.

8. CONCLUSIONS

In this paper we have presented an analysis of race condition attack, its components, and an example technique. Moreover we have surveyed various exploits of race condition and their affected applications. We have found out, Firefox and Internet explorer web browser have been vulnerable to race condition attack. Various versions of Windows and Linux operating systems are still at risk of race condition attack. Exploit like Dirty COW has just been released in October 2016. This indicates that even after good number of research in the field there exists a gap between existing the detection and prevention techniques of race condition. In various cases this vulnerability exists due to the lack of programmer's knowledge. As per countermeasures the programmers should employ some synchronization primitives in order to explicitly serialize the process accesses to critical regions. They might also use a two phase save algorithm with fine tuned error handling to address the issue.

9. REFERENCES

[1] S. Carr, J. Mayo, and C.K. Shene, "Race Conditions: A Case Study", Journal of Computing Sciences in Colleges, vol. 17, no. 1, pp. 90–105, Oct. 2001.

[2] R. H. Netzer and B. P. Miller. "What are race conditions? some issues and formalizations". ACM Letters on programming Languages and Systems, 1(1):74–88, Mar.1992.

[3] K. Lhee and S. J. Chapin, "Detection of file-based race Conditions ". International Journal of Information Security, pages 105–119, February 2005.

[4] R. H. B. Netzer, and S.Ghosh, "Efficient race condition detection for shared-memory programs with post/wait synchronization". in Proceedings of the 1992 International Conference on Parallel Processing, St. Charles, IL, Aug. 1992.

[5] C. Flanagan and S. N. Freund, "Detecting race conditions in large programs," in Proceedings of the 2001 ACM SIGPLANSIGSOFT workshop on Program analysis for software tools and engineering, New York, NY, USA, pp. 90–96, ACM, 2001.

[6] M. Bishop, and M. Dilger "Checking for race conditions in file accesses". Computer System, pp 131–152.

[7] J. Rouzaud-Cornabas, P. Clemente, & C. Toinard, "An Information Flow Approach for Preventing Race Conditions: Dynamic Protection of the Linux OS". in Fourth IEEE International Conference on Emerging Security Information Systems and Technologies, pp. 11-16, Jul 18, 2010.

[8] A. T. Do-Mai, T. D. Diep, & N. Thoai, "Race Condition and Deadlock Detection for Large-Scale Applications". in 15th IEEE International Symposium on Parallel and Distributed Computing (ISPDC), pp. 319-326, Jul 8 2016.

[9] E. Tsyrklevich & B. Yee, "Dynamic detection and prevention of race conditions in file accesses," in Proceedings of the 12th conference on USENIX Security Symposium (SSYM'03), Berkeley, CA, USA, pp. 17–17, 2003.

[10] T. Farah, R. Rahman, M. S. Hossain, D. Alam, & M Zaman, "Study of the dirty copy on write, a linux kernel memory allocation vulnerability" in Proceedings of 3rd international conference on electrical engineering and information technology, Dubai, UAE, June 9-11, 2017.

[11] "Linux kernel PTrace Race Condition". [online] Available: https://xorl.wordpress.com/2009/05/08/linux-kernel-ptrace-race-condition. accessed 9th June 2017.

[12] "Race Conditions and Critical Sections". [online] Available:http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html. accessed 9th June 2017.

[13] "Race Condition Exploits". [online] Available: http://cecs.wright.edu/~pmateti/InternetSecurity/Lectures/RaceConditions/. accessed 9th June 2017.

[14] "Avoid Race Conditions". [online] Available: http://tldp.org/HOWTO/Secure-Programs-HOWTO/avoid-race.html. accessed 19thSeptember 2017.