CSE 509 Lecture 25

Prof. Rob Johnson, Scribe:Arun Rathakrishnan

December 4, 2013

1 Case Study: Algorithmic Complexity Attacks for UNIX File System Race Conditions

The above paper describes how different systems can interact to create vulnerabilities and how the combination of race condition, unix system call implementation, scheduling and cache systems can be easily exploited by an attacker to perform an unauthorised file access.

2 Race Condition

Consider a victim program and an attacker script that run side by side on a processor. The operating system scheduler can select a process to run for a specific time slice.

Here's the victim program, victim.c.

```
fd1 = open(filename,...);
...
fd2 = open(filename,...);
```

Here's the attacker's script.

```
rm filename
ln -s target filename
```

After the first open system call in victim.c completes, the process may be scheduled out, and it is possible for the attacker's script to get scheduled. If that happens, and the attacker script completes before the second open, it would turn out that $fd1$ and $fd2$ are in fact descriptor's to different files, the latter corresponding to the file pointed by the newly created soft link.

In practice *access* system call may be used in conjunction with *open*. *access* checks if the real user-id of the current process has permissions to open a file for certain operations. It is especially useful in programs which use *setuid*(0) to gain root access, but still want to access files with the same permission as the process's real user possesses.

```
if(access(filename, ...)){
    open(filename,...);
}
```

lpr command which allows a file to be sent to a printer, used a similar approach. It is possible for an attacker to exploit *lpr* as follows. The attacker can create a file for which the *access* call would succeed and start lpr. After the *access* call succeeds, but before *open* is called, if the attacker's script gets scheduled and deletes the file and creates a link, as in the previous example, attacker will be able to access /etc/shadow file, which is protected with root permissions.

```
ln -s innocent_file filename
lpr filename arg
rm filename
ln -s /etc/shadow filename
```

3 Counter measures

3.1 Eliminate access system call

We could allow the process to reset the effective user id before opening a file. But if the effective user id is set from 0 to another value, operating system can not reset it back to 0, again using *setuid*. But the OS provides a fix in the form of a *setres* system call, which remembers all previous values of effective user id, and allows the process to set a effective user id that a process possessed before.

3.2 Prevent scheduling till open returns

The only way for an attack to succeed is to make changes to the file system, between the execution of *access* and *open*. If we could prevent scheduling, after *access* is called but before *open* returns, we can effectively prevent this attack. But it turns out that each look up operation, which maps file name to inode on disk incurs I/O. It has been shown that for long paths, it could take several minutes to resolve the path and get an inode. Preventing scheduling can effectively freeze the system in such cases.

3.3 Transaction Based System

Such a system would begin a transaction before the *access* call and check that the inode object being referred to is the same during both the calls. If they are the same, the transaction is committed, otherwise it does not proceed.

3.4 Repeating checks

Though it may seem unlikely that an attacker can exploit race conditions to mount a successful attack, by choosing the input parameters (long file names), time of attack

(not many other processes are running alongside), one can better the odds. By repeating access check and open, ensuring that the unique identifiers of the file remain the same during each attempt, we can make it difficult for an attack to succeed. Instead of having to succeed once, the attacker must make changes to the file systems, d times between system calls to successfully exploit the vulnerability.

```
    for(i = 0 to d-1){
         if(access(filename, ...)){
             fd[i] = open(filename,...);
             st[i] = fstat(fd[i]);
             assert( i == 0 || st[i-1] == st[i]);
         }
    }
```

This fix is effective because, the open should be to the same file during all attempts, otherwise the assertion will fail. This also means that cache effects ensure that subsequent *open*'s succeed fast, possibly eliminating I/O and hence making it highly likely that after *open* returns, *access* gets called before the current process is scheduled out.

The problem with this approach is that, the attacker controls file name and by setting the softlink to different long paths that point to the same inode, he can make sure that open system call incurs I/O, despite having to open the same file d times.

3.5 Atomic Checks

The major flaw with the previous approach is that, there exists different paths to the same file name, and open tries to fetch an entire long path in one go. This buys time for the attacker's code to be scheduled, during each time the check is performed.

```
foreach atom in filename{
    for(i = 0 to d-1){
         if(access(atom, ...)){
             fd[i] = open(atom,... |O_NOFOLLOW);
             st[i] = fstat(fd[i]);
             assert( i == 0 || st[i-1] == st[i]);
       }
    }
}
```

By preventing the open system call from following a link and incrementally opening a file path on a component by component basis, we can ensure that open executes fast.

It is only possible to change the file name between *access* and *open*, if the time slice for the process expires at the end of a system call, and the attacker's process can be scheduled between these times, especially if it has a higher priority than the victim

process. In unix, if a system call has started before the end of a time slice, it can continue running beyond the time slice, provided it makes no I/O. But having to precisely schedule a process (by using *sleep* system call) and have it succeed multiple times, makes such an attack unlikely to succeed.

3.6 Algorithmic Attacks

Linux file system makes use of dcache as a in-memory hash table, which given a directory id, can return a inode for a file. Whenever *lookup* has to be performed during the *open* call, the dcache is searched to get the inode or dentry of recently accessed files. When multiple keys collide on the same slot in the hash table, chaining is used to resolve collision. This leaves the possibility of having a long chain of cached entries for the *lookup* call to search through, which can take longer than a scheduling time slice, thus eliminating any gains made through making lookups for short atomic paths during *open* system call. The attacker can thus increase the interval between *open* and *access*, and hope to schedule his process successfully between the two.

3.7 Birthday Attacks

Several operating systems like Linux, Open BSD, Solaris use similar dcache mechanisms. Old versions of perl had collision bugs that could be exploited to stage DoS attacks on webservers. The general hashing scheme used for dcache is

$$hash(dirid, name) = h_2(dirid, h(name))$$

If the *name*'s hash to the same slot, by placing them in the same directory, an attacker can ensure that they will have the same *hash* value. The h function is defined as below.

$$h(s_{n-1}...s_0) = \sum_{i=0}^{n-1} s_i 17^i \bmod 2^{32} = \sum_{i=0}^{\frac{n-1}{2}} s_i 17^i + \sum_{i=\frac{n-1}{2}-1}^{n-1} s_i 17^i$$

Thus,

$$h(s_{n-1}...s_0) = h(s_{\frac{n}{2}}...s_0) + 17^{\frac{n-1}{2}+1} h(s_{n-1}...s_{\frac{n-1}{2}+1}).$$

If we can find two strings S_1 and S_2 such that $h(S_1) = -17^{\frac{n-1}{2}+1} h(S_2) \bmod 2^{32}$, we can ensure that both strings have the same slot. This attack is called "Birthday Attack". Assuming that slot size is a 32 bit number, the total number of values a hash function can match is 2^{32}. If we can generate k random strings, compute their hash and store it in a table and sort the table by the hash key, by birthday paradox,

$$P[match] = \frac{2^k}{2^{32}}$$

$$E[numberofmatches] = 2^k \times \frac{2^k}{2^{32}}$$

By choosing an appropriate value for k, we can expect to have several names from among the randomly generated strings with matching h values, which can be used to cause collisions in dcache.