Last Class:
We discussed how to write wrappers to attenuate capabilities. If you have capability to access file system and you have to give another module to have access the file system with restricted access, then you can write wrapper object that can wrap your capability with others capability that will do checks to make sure your requirements are met. Then we can pass that wrapper to the subsystem. The other subsystem can access it through wrapper.
This is a capability based design model. It facilitates easy code review, as the only resources that a module can access, are those, which we pass through their interface.
It facilitates least privilege. When we include the module into the system then we can give least amount of power.
There is another principle going which is, secure default, by default a module has got nothing.

We have mutually destructing modules, which can exist in same program and can make calls to each other, with zero over head. Then we talked about hardware and language based capabilities.
One extra thing that I want to point out was if you think about design for the Taindsay project, where there is a secure button. When the user presses that button, it generates a classifier object and passes declassified object to the system.
This is capability-oriented design.
The constructor for the classifier object has to be protected.

Today’s Class: Case study

How diff systems can interact to create vulnerability?
We have to do little bit of crypto, scheduler stuff, file systems, and we will discuss, what’s going on in the attack. It will bring a lot of stuff in the systems.

This is based on the paper that professor wrote,

Algorithmic Complexity Exploiting for UNIX file system race condition


The race condition is the order of different process effect the outcome of result. File System race condition is a general class of bug. In UNIX, when we call ‘open’ we have to specify a file name.
Open(filename).

Unix has symbolic link, hard link, soft link. If the app does this,

Program1: Victim
fd1 = open(filename);

fd2 = open(filename);

i.e. open the same file name twice, the file descriptors that we get back may not refer to the same file. Because what can happen is suppose program1 is victim program and the schedule stops the execution of the victim and stops running it. It happens to switch to an attacker. The attacker manages to run between these open. Then the attacker knows what the filename is, and then the attacker can do,

Program1: 
fd1 = open(filename);

Attacker program:
rm filename
ln -s target filename

fd2 = open(filename);

Now this the second open will not open the filename specified by the victim, it will open that is specified by the attacker. It is called race condition coz the attacker has to win the race.
The most common kind of victim program that we want to attack is:

If (access (filename))
fd2 = open(filename)

Suppose this is a victim with setuid – root program. The victim is running with root privilege. The victim, no matter who ran it, when runs its running in root. The OS remembers a process with having an effective user id or a read user id. So its effective user id is 0 and real user id is (depending on who ran it) 1027.
Open, here is using effective user id. Access is intended to answer question could u open the file using our real root id? So the purpose of this approach is that here is a set user id root program that any user on the system can run, the user must be of specified type. Then this program is intended to check the person who invoked can open this file. I am opening this file on behalf of the invoker but actually I am ending up using the root privilege to open this file. Now, we can see how dangerous this race condition is as the person who executes the program if he is the attacker, then what he can do, run this program, and if he is lucky he will get schedule right here, i.e. in-between the two opens.

Program1:
fd1 = open(filename);

Attacker program:
ln –s innocent_file filename
rm filename
ln –s target filename

fd2 = open(filename);

So the attacker now has a new instruction ln –s innocent_file filename, to create the symbolic link of the filename before deleting it. It changes the target of the file. One target that we can consider is for example,

ln –s innocent_file filename
rm filename
ln –s /etc/shadow filename

The victim that had this bug is:
lpr

So what we can do there are some versions of printing program that set the user id to root, as they have to access parallel port. So we can run lpr to print, they will try to check that ‘I want to print a file and you are not able to read that file on the computer’. Lpr will check that and file will be printed. You can use this file to print etc/shadow.

So using ‘Access’ is evil if we use it in our program, so we should avoid its usage.
Solutions to the problem:
One way to solve this problem by using a system call ‘setuid’ which can be use to change your effective user id to real user id.

One way to solve irreversibility:
Fork of a child, let the child call setuid. Let the child open(), pass file descriptor back to parent and then the child dies. This is very complicated solution

There is also solution,
There is a function setresuid(). There is another userid, called saved userid. Lets set real effective save. We just need to specify all different user ids that we want to set.

So this way we can swap real and effective user ids.

One particular approach that people use to solve this kind of problem:
The attacker has to be really lucky, as he has to get between the two open system calls discussed above. We do the following,

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

‘fstat()’ gives unique identifier for each file. It gives device number and inode number. So if we have two different file descriptors, we will get the same device and inode numbers if we fstat them.
So what is done by the above program is that first access is called, then the file is opened, then it is checked whether the same file is opened before or not. So the sequences of execution are,

```
access
open
access
open
access
open
```

We have to make sure that all these open(), give the same file descriptors.

What the attacker has to do to fool this system?
1. One of the way suggested is to do it in the last iteration of the sequence below. What will happen?
2. Then filename either references innocent file or target file.

If it is a target file, it will
Get kill here.

```
access
open
access
open
access
open
access
open
```

this is OK
this is OK
this is OK
this is OK
Not OK
If it is referencing innocent file all the way down to the last access

So we have to take a note here, that whatever access we do here, open has to use that file which in this case is innocent file. And when open gets executed, filename has to be referred to a target file.

So there will be SWAPPING.

So, attacker has to win a lot of races.  
So the argument behind the system was, there were bunch of experiments. When the authors ran the attack, they won the race about 1/1000 times, but now they going to build a system where single race is 1/1000 times so now we have to win 10 races.

So, what is the probability that we win all 10?

Its, \( \left( \frac{1}{1000} \right)^{10} \). This is good; this is the idea behind the system. Its equally hard to break into crypto system.

**Assumption about these races**

What has to be true to multiply probabilities?

They have to be independent.

One thing we might worry about it is, it is easy to win a race but difficult to win another race. What the author’s thought was if we can win a race at first, and good
way to win a race is - If access require IO, then what happens when process requires anything that does IO? - It goes to sleep, something else gets scheduled. So great way for attacker to attack and get scheduled in between the two system calls.

This requires I/O then there will be Innocent file here. This is target file, may be this requires I/O too. At this time the attacker will switch it back to innocent file, which will be in memory from the last time. If you win first set of round, after sometime there will be no I/O as you go down the range. Therefore the odds of winning will get reduced.

What defender thought to do?

We can break this down, so that this filename is a single non-symbolic link component. The filename can be like ‘foo’. It should not contain any ‘/’ and furthermore it should not be symbolic link. We cannot play any dirty games with long chain. May be there is I/O initially, but not later. So this was the idea.

The defender will break the filename into pieces; if a single component is a symbolic link then it will break that into pieces. So it will perform this race on each path. So this trick of forcing IO is not possible anymore. This access would be doing just a small system call.

For e.g. we can say,

For each atom in filename
for (I = 0 to k-1)

    access(atom, NOFOLLOW)

    fd[i] = open(atom, NOFOLLOW)

    st[i] = fstat(fd[i])

    assert(i=0 || st[i] == st[i-1])

How kernel Operates?

The kernel contains something called DCACHE. The DCACHE is a hash table in memory. It do not maps full path with names. It maps atoms.

    (dir id, atom) \rightarrow inode

inode is the information of the file.

1. So when user space application calls open on this atom, this is relative to some directory and so what kernel immediately does, it check in the DEACACHE, the process current working directory and tries to look them up.
2. If it finds memory in cache then it is done (it found the file). It constructs a new entry in the file table and returns the file descriptor back.
3. If it doesn't find entry in DEACACHE, then it has to go onto the disk.
4. What will happen is, whenever we call the access system call, entry or that atom will get loaded into the DEACACHE.
5. Even if the attacker messes around with the filename or atom, it is messing around with it in the DEACACHE. So if anytime, anybody modifies it or looks it up, it is going to be brought back to DEACACHE.
6. So at this point open system call will be done very quickly and kernel is going to find results in the DEACACHE.
7. So we can get an idea that Linux standard Scheduling quantum is like 100th of a second. Time it takes to do this two-system call is like 100,000th of a second or may be 1000th of a second.

Till here we have talked about DEACACHE, file system, scheduler.

More information about DCACHE, if we have to implement a hash table, how will we do it?

- Have a hash function
- We will hash (div id, atom) \rightarrow inode
If we have a hash function, it will give us a bucket then we can insert or search inside the bucket. What goes into the bucket?

hash(dir id, atom)

The way the kernel works, until its I/O typically a system call made by an app, it will complete that system call. However at the end of the system call, before returning to execute the returning code, kernel checks, did this app timer clicked during the execution of the system call? The app gets schedule at this time, then the app calls open.

time

To

Victim gets schedule here when it calls open

calls open()

***

Scheduler run here
**Technically, if the that app gets time out, then what kernel does is, when it is about to return the control to application, it run scheduler right here.**

When victim gets schedule at the top, then it executes for sometime, then calls open. Now the kernel which is a multi purpose system, sets the clock with 100th of a sec and when that c=goes off, and if the app is still running then it will stop the app and run the scheduler. The kernel will complete the system call and run the scheduler.

There is another way this could happen. If there is another process which is sleeping and there is another process with higher priority. Then that process say that I want to sleep till the some point. What will happen?

Sleep for this duration

At this point the process wakes up, then the kernel will run the 2nd program rather than the first.

The thing that happen here is that one program is sleeping (higher priority) and the other is sleeping. At the open system call the other program wakes up, as this is of high priority, the scheduler will run that program and stops the execution of the first one. Therefore, kernel will then run the wrong program rather than the correct one.

Now, how can we start manipulating scheduler to behave as a bad guy?

- Bad guy wants to run straight after the access and then it has to run direct after the open also. So it has to run straight after every access open sequence.
- So if the bad guy predicts when the access is going to occur. Suppose it calculates that access can occur after 12ms, so then the bad guy can sleep for 12 ms.
- So victim will run and bad guy wakes up right in the middle of victims’ access or open system call.
- Then bad guy can do some work and then he can go back to sleep for 12 more ms.
- Then again bad guy wakes up and he get scheduled. Then he again sleeps.
To
bad
guy
sleeps
for
12ms
time

The bad guy will wake up after 12 ms and gets scheduled

Access

Bad guy sleeps for 12ms

So the priority makes it wake bad guy again and again. And do its work. The bad guy wakes up in the middle of victim next system call.

We are exploring, the sleeping process gets higher priority, we are exploring how the scheduler runs at the end of system call, how defense is done. The only thing which is difficult about this is, we discussed about when the open system call hits the DCACHE it is really fast. It is kind of hard to specify, as we are waking in the middle of something.

How we can make open take long time to reach atom?

Hash collision, this bring us to the algorithmic part.

If we can force the system into a bad state where the atom that the victim is looking over and over again is way down at the bottom of some chain, then even if it is in the DECACHE and this chain if it has 10,000 things and atom is the last one, it will take a long time to go there.
So this is the algorithmic complexity part of the paper. People have found bugs in Perl interpreters have these bugs. We make them so slow, that it takes several second to execute a small instruction.

We need a bunch of thing that collide. How will you find the filename that collide together? Kernel has this Hash function.

\[
\text{Hash (dir id, name) = h2(dir id, h(name))}
\]

It will hash directory id and name. Directory id (dir id) they use is like address in memory where directory information stores in the kernel. So if an attacker is running in the user space it doesn't know what it is. But the good news of this structure is that hash function is: If we make two names that collide under H and we put them into same directory, then they will still collide under that hash.

So it all boils down to finding all the names that collide under H. So finally we bring in the crypto.
\[ h(S_{n-1} \ldots S_0) = \sum_{i=0}^{n-1} S_i \cdot 17^i \mod 2^{32} \]

\[
= \sum_{i=0}^{(n-1)/2} S_i \cdot 17^i + \sum_{i=\frac{n-1}{2}+1}^{n-1} S_i \cdot 17^i
\]

\[ = h(S_{(n-1)/2} \ldots S_0) + 17^{n-1} h(S_{n-1} \ldots S_{n-1/2+1}) \mod 2^{32} = 0 \]

First equation says that this is like you treat these as coefficient of polynomial and we evaluate that polynomial at 17. So what we need to do is we have to find allot of string that collide.

So, the hash of the whole string = hash of the second string + \(17^{n-1/2+1}\) (hash of the first string)

**Birthday attacks can be done,**

<table>
<thead>
<tr>
<th>String</th>
<th>(H(S_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>H(S1)</td>
</tr>
<tr>
<td>S1</td>
<td>H(S1)</td>
</tr>
<tr>
<td>S1</td>
<td>H(S1)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>H(S1)</td>
</tr>
</tbody>
</table>
Here is the trick, we generate a bunch of random strings and we compute their hash, then we sort them by second column. Then we have a target hash I am looking for, then we can find that by binary search. After that we take this first string and take its hash, multiply this with that -17 number, now the number that I get after multiplying, I look at number in the table using binary search. If I find the answer, then we make them together to make a string that hashes to 0.

So there are $2^{32}$ different values. If we take $k = 2^{20}$ then, when we compute, then we look into the second column. Probability for the match is,

$$P(\text{match}) = \frac{2^{20}}{2^{32}}.$$

The expected number of matches would be,

$$E(\text{matched}) = \frac{(2^k)^2}{2^{32}}.$$

So if we want 65,000 matches, we can take $k = 24$ which is plenty to make this table big.

So we discussed, hash function, files system APIs, some crypto, some stuff about scheduler, some stuff about file system, some stuff about I/O, birthday attack, and we can generate these attacks by their interaction.

Class Concluded