1 Password

The key to this attack is the fact you can use page faults to help notify you of the validity of your guesses for the password.

So, what you do is you place your guess for the password straddling 2 page frames such that to check all the characters in your guess against the actual password, you're going to have to bring another page into memory. What you can then do, since you can run user code in response to page faults, is check, character by character whether you actually hit a page fault. If you make it to a page fault (which can notify you via something like a print statement that there's been a page fault), this means all the previous characters guessed against the password were correct. Conversely, if a character guessed is not correct, since we're checking character by character, the comparison against the password should immediately return false; you'll never make it to the page fault and thus know your guess was incorrect.

As an example, let's say the password is "dog". You set up your guess by placing in memory at the end of a page "a" and another "a" at the start of a different page:

```
(1)     "a" | "a"           /* '|' represents page border */
```

This above guess of "a" | "a" will evaluate to false, since the first character in the password is "d", and we'll know "a" is not right since we never get a page fault. After trying multiple guesses, we finally try:

```
(2)     "d" | "a"
```

This will now result in a page fault; "d" evaluates to true, so the program then tries to get the next character to compare against the password. Since this is on the next page, it page faults, thus we know that "d" was a correct guess.

Using this method, we can add more characters to the guess before the page fault:

```
(3)     "do" | "a"

(4)     "dog" | "a"       /* should authenticate before fault */
```

Thus checking character by character and eventually determining the password.

2 Directory-Path

2.1 scenario 1

Having inodes, instead of just path names, helps to uniquely identify files. Without this, if the path to a file changes while a user is editing a file in their local cache, when they then try to write this file back out to the server the previous path won't be there.

For example, let's say B has a file open, accessing '/home/hello.txt'. So B brings part of the file into its cache and starts editing it. A, however, moves '/home/hello.txt' to a new file '/home/mine.txt' and then replaces '/home/hello.txt' with a completely blank file. If B can only reference the file via the path name, when he writes his cache back out to '/home/hello.txt', it'll point to the blank 'hello.txt' instead of the one he was originally accessing (which is now along the path '/home/mine.txt')

2.2 scenario 2

I think this problem could also manifest itself on a single client if that client were making use of hard links. So a hard link essentially allows you to give the same file multiple names – so both 'foo.txt' and 'bar.txt' point to the same file.

Let's say there's a hard link on the server, so both 'A' and 'B' reference the same file on server. A client then opens both A and B, but since a directory-path is the only thing passed with file handles (instead of inodes), I think the the client would think it's accessing two separate files. Now, let's say the client edits A then B. Since the client thinks its referencing different files, when these files are written out from the client's data cache back to server, the write back from the cache might not happen as intended, with the changes to B being written first and then being overwritten by the changes to A, i.e. there could be consistency issues.

3 CIFS

3.1 a

No. I think the problem here relates to the grace period servers have once they come back up from a crash. Since the server doesn't hold any state information regarding locks in stable storage, when it comes back up from a crash it needs to ping clients that are connected to it asking for them to 'reclaim' locks. This period when clients can reclaim locks, but can't request new locks, is called the grace period.

A big problem with this, though, is let's say there's unexpected latency between client A holding a lock and the server. Client A, therefore, might not be able to reclaim their lock during the grace period – another client, B, can thus, after the grace period, request and get the lock before A can restore its state with the server to what it was before the crash. A, in this case, wouldn't be able to continue to operate as normal post-crash since it would no longer have the lock it needs.

As another example, let's say A holds a write lock to a file and then a server goes down. There's substantial latency between A and the server, so after the grace period B grabs a write lock for the file A was using and then starts writing to the file. The server goes down again, and when it comes up A is able to retake the lock it previously had. Now, A will continue writing to the file (something it wouldn't be able to do in scenario 1 if it was holding a write lock) but the file it's writing to has been edited by B, thus there are consistency issues.

3.2 b

I would think this would solve the above problem. If the state of the locks was held on server in stable storage, there would be no way for a client, other than the client holding a lock at the time of the server crash, to claim a lock. Therefore the lock state wouldn't get confused like in part a.

3.3 c

I think the answer to this is through something like a heartbeat, where the server periodically pings the client to make sure it hasn't crashed.

I know CIFS normally relies on communication between the server and the client as a means of detecting crashes. NFSv2, on the other hand, is more conservative and only acknowledges a client crash when the client notifies it that it's rebooting. Periodically pinging the client, however, provides much more granularity.

Let's say you want to revoke the opportunity locks of a client when it goes down. If you're waiting on the client to notify you that it's rebooting, there might be instances when the client is down for a prolonged period of time, which means the server can't clear the lock state of the client and distribute its locks to other clients, which is bad.

On the other hand, if the server is consistently pinging clients to make sure they are online, you run into the issue where, due to latency or network outages, perhaps a client is still running but can't respond in time to the server. Thus, the server clears its lock state while it's still running and redistributes those locks even though the original client is still trying to access a file. This is bad because the original client can't operate as normal since its locks were taken away while it was still using them.