1)

If Alice is currently browsing a site Eve owns, what Eve could do is embed on one of the pages on eve.com a request to bob.com. Now, whenever Alice visits eve.com, while loading the page Alice wants eve.com will make a request to bob.com. eve.com doesn't even have to make use of the response from bob.com, so Alice might never even be aware her browser is contacting another site.

If Alice is still logged onto bob.com, and if her connection isn't through HTTPS, then this request to bob.com coming from her visiting eve's website will contain her session cookies, which Eve could sniff, thus allowing Eve to acquire Alice's cookies.

2)

(a) For a session cookie scheme to be secure, I would think it would need a few properties:

- (1) The cookie of any particular user should be hard to guess, so others can't hijack the session of other users.

- (2) It should be the case that session cookies are unique. For example, you wouldn't want a system where two users accidentally end up with the same cookie s.t. they see each others' changes/preferences.

- (3) The cookies should map reliably to the user's preferences. It would be pointless if the cookies wasn't able to reliably save the state of the user.

I think this is really all you need. As long as you can ensure that cookies are unique and hard to guess (cryptographically infeasible), once user A has a session cookie it would be almost impossible for user B to access websites as A.

(b) So, I'm really not sure if I'm thinking about this the correct way, but what drew my attention in this problem was when it mentioned that the token will no longer simply be a random key. Also, I was thinking, to save a user's state, it can either be stored server side (like it is now) or it can be stored client side.

So, for example, let's say instead of a random key my session id is based off a hash of a password I provide or, even better, is a hash of my browser's digital fingerprint (we saw in class that many factors can go into identifying unique users and machines based on type of machine, browser addons, etc). If this is the case, when I make a request to a server, the server will know it's me since only I will have my particular electronic fingerprint. This is the unique session cookie (2). Certain preferences can then be saved client side and can interact with change of state operations that need to be server side (such as transferring money between bank accounts, etc).

3)

(a) Even though this photo is 'private' it's not implementing a security mechanism where it, say, checks the user id of the person visiting the page and if it's not Zoe's it refuses to render the page. Instead, what it does is it creates essentially a cryptographic string for this image. If you know what this authenticating value is you can view the image, however, facebook is relying on the fact that it would be infeasible to try and brute force the correct value.

(b) It appears they haven't implemented what I described above, whereby if you're not the correct user it checks and refuses to render the page. This seems like a case where Facebook isn't implementing improper path sanitization; I can go to the page even if I'm not supposed to.

(c) In order for a user to view the image, they essentially just need to be able to get access to this authenticating value, either by guessing the value correctly (which is unlikely) or perhaps seeing what the value you. If they were able to see what the value is, they'd be able to log in and see the image regardless of its privacy settings.

(d) I do think it's reasonable. Everything is a trade off: more valuable things you want to make more secure. Most likely, although users might want to keep certain photos private on fb, if someone else did see the image it wouldn't be as consequential as, say, someone else getting access to a user's bank account. Facebook's decision to make privacy dependent on this seemingly random cryptographic value makes successfully brute forcing the value unlikely.

I think this is similar to how google handles private google docs, as well. An added bonus to this, too, is if I have a private doc and want to share it with a single user, I can simply give them the URL, explicitly passing on the cryptographic value attached to the document.

4)

(a) From the wikipedia article, it appears that the CSP provides a way for owners of websites to declare acceptable origins for the content that is loaded on their sites. So, for example, the CSP would state what the acceptable origins of executable scripts are.

Let's say my browser trusts Alice s.t. executable scripts coming from Alice's server I have no problem running. Eve, however, is malicious and wants my browser to execute potentially harmful scripts coming from her server. What CSP does is it'll whitelist Alice's server since I know it's trustworthy, while blacklisting Eve's. In this way I'm only granting privilege to run scripts on my browser to an entity I know is good.

CSP will also disable things like <script>'ing blocks, inline scripts, DOM event handlers, eval(), etc. all tools that make launching an XSS attack easy. In this way it provides multiple ways of combating XSS.

(b) By whitelisting only acceptable sources of executable code, even if a malicious user was somehow able to inject malicious code into your site, your site would never run it. Additionally, if you have inline JS functions, you can whitelist particular functions. If I were to launch an attack like the one on Foo's website where in the comment box I inserted some code in <script> tags, since this wouldn't be whitelisted it wouldn't run. The key to CSP is explicitly marking what code and what sources are reliable and then only executing code from those parties.

(c) From the wikipedia article it says there have been ways to exploit older JS libraries on servers that have been whitelisted. So, for example, if google has been whitelisted but there are vulnerabilities in some of the JS libraries on their server, it might be possible to exploit these code bases before they're served to the client. In this way, although the client trusts google some of the code coming from google might be compromised.

Although CSP doesn't allow inline JS, if there're certain inline functions you need you can whitelist these particular functions; one way to do this is using a nonce. Wikipedia also mentions there have been ways to bypass this nonce, which would essentially allow you to inject inline executable code (like what I did on Foo's website) and still have the client run it.

Also, I'm not sure if CSP would protect against a reflected XSS attack since that code isn't executed by the application (which presumably is keeping track of the whitelisting) but rather is code within an HTTP response.

5)

(a) The reason this works is because even if Alice isn't on the bank's website, she's still logged in i.e. her cookie still validates her on the bank's website. So, let's say Alice logs into her bank, then opens another window and starts browsing mal.com. If there's a link on mal.com that, say, makes a GET request to bank.com transferring money out of Alice's account, since Alice is still holding a cookie validating her on the bank's site, if she clicks the link on mal.com her bank won't be able to differentiate this from a legitimate request.

(b) For example, let's say mal.com makes a request to bank.com trying to inspect elements of bank.com's DOM, perhaps trying to inspect account numbers on bank.com's DOM. This kind of request would be blocked since the SOP allows different websites to make requests to one another, but it doesn't allow them to inspect or edit other sites' DOMs.

(c) This does not necessarily block the kind of attack described in (a), however, where identifiers through a GET request are passed as url parameters to some function on another site. I used this when attacking Foo's site: you could embed links that used GET requests to delete users or assignments. Here, the malicious user is not trying to examine or edit the DOM, but simply executing some change of state function.

(d) So let's say I'm on the bank's site and I try to make a sensitive request. When I do, a synchronizer token will be included in this request to the server. Since I'm on bank.com, the token will match the one on the server and the request will handled.

Let's say, however, I'm on eve.com. A request from eve.com, even if it looks the exact same, won't be able to include a valid synchronizer token (these tokens are usually hidden fields in html forms, and are long, cryptographically strong, and unique per session id). Furthermore, eve.com won't be able to edit or inspect the DOM of bank.com (according to SOP), therefore there will be no way for eve.com to trick the server into accepting a fraudulent synchronizer token. A request from eve.com won't have the correct token and its request will be rejected.

6)

Unlike when a process creates a thread, when a process creates another process the memory between the two processes remains distinct. Like the problem says, though, you can control what arguments are passed to the process, change its pwd, and the child process will inherit the file descriptor table from the parent process.

So let's say hypothetically a setuid binary is supposed to execute in a particular directory. If I were to set the working directory of the parent process to something different before creating the child process executing the setuid binary, the child process will inherit the parent's working directory and might execute in the wrong place.

Likewise, you could do the same with file descriptors. Since a child process will inherit the fds of the parent process, if you execute the setuid binary in a child process you can manipulate where the executable is writing to. For example, instead of writing out to a protected file in a protected directory, you might get it to write to STDOUT where you could see it.

Also, if the setuid binary doesn't check its parameters, you might be able to pass it arguments causing undefined behavior or causing the executable to do something malicious.

Overall, though, I would think that a setuid binary would want to (1) be thorough in checking its arguments and (2) not rely on inherited data structures like file descriptors -- it seems like all the attacks above could be thwarted if the binary makes sure to freshly initialize anything that a child process might share with a parent process.