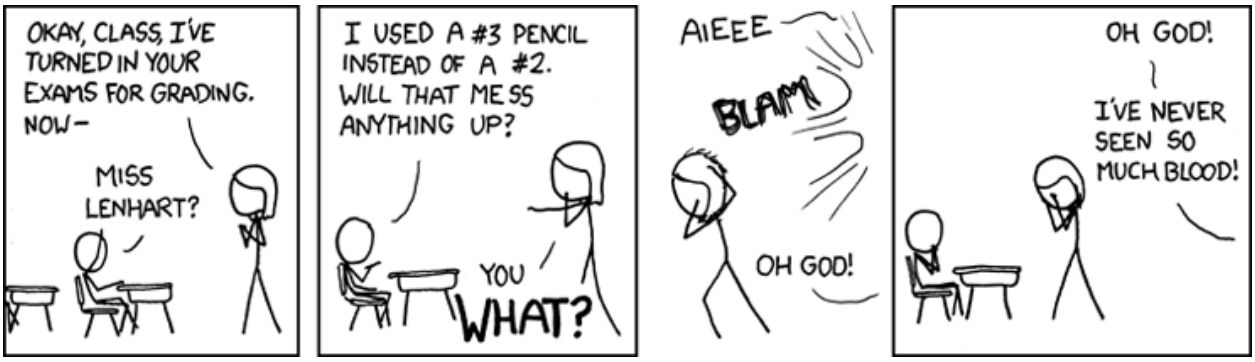
Name: _____ UNI: _____

Final Exam: May 2020
COMS W4182: Computer Security II

Rules

- **Important: write your name and UNI on your answer document**.
- Books and notes are allowed during this examination; computers may be used for word-processing only.
- Most questions can be answered in just a paragraph or two; if you think you need to write several pages, you're writing too much and may be on the wrong track entirely. If a question is worth only a very few points, that's a pretty good clue that the answer is pretty simple.
- *You must not communicate with anyone or look up anything online during this exam.*
- The total points add up to 100.
- Good luck, and may the Force be with you.

Question	Points	Score
1	10	
2	20	
3	20	
4	20	
5	15	
6	15	
Total:	100	



https://xkcd.com/499/

1. (10 points) Zoom has been criticized for poor security: bad cryptography, lack of end-to-end encryption, guessable meeting IDs that can lead to Zoom-bombing, apparently poor quality code per CyberITL's metrics, abuse of privileged mechanisms on Macs, etc.

   Suppose they respond by saying "we're going to put a firewall in front of our servers; all participants' computers should also be behind a firewall." Will this help? Explain.

   Answer:

   The help will be minimal or non-existent. With the possible exception of poor quality code, all of the other issues apply to connections that any firewall would have to permit. The necessary services also have poor code quality, so that code would have to be exposed. The only possible benefit of a firewall would be if their servers had open ports for internal purposes and that the services on these ports would be abusable before appropriate authentication.

2. SWIFT is a worldwide network used by banks to send each other financial transaction information. Consumers and businesses do not connect directly to SWIFT; only financial institutions can. It has been abused—in 2016, hackers allegedly working on behalf of North Korea tried to use it to steal US$1 billion via a Bangladeshi bank. *Ars Technica* wrote:

   > SWIFT's security stems from two major sources. Notionally, it's a private network, and most banks set up their accounts such that only certain transactions between particular parties are permitted. The network privacy means that it should be hard for someone outside a bank to attack the network, but if a hacker breaks into a bank—as was the case here—then that protection evaporates. The Bangladesh central bank has all the necessary SWIFT software and authorized access to the SWIFT network. Any hacker running code within the Bangladesh bank also has access to the software and network.

   You've been hired to do a penetration test of some bank's SWIFT gateway.

   (a) (10 points) What should your goal be? That is, what concrete, verifiable result would prove that you had successfully hacked your client's SWIFT gateway.

   Answer:

   Although there are many possible answers, the primary threat to SWIFT is stealing money. The goal, then, should be to transfer money from some test account that the "attacker" does not have access to—you don't want to take money from real users!—to some other test account.

   Another useful choice for a different style of penetration test: read a specified file that should only be available to `root` on that system.

   (A secondary goal is for the site: did their IDS detect your activities.)

   (b) (10 points) You of course do not want to damage the SWIFT gateway in any way, and especially not do anything that would leave it open to other attackers. What steps would you take to ensure that there was no damage?

   Answer:

   The best answer is not to do your work on the live system.

   The site should already have a test machine and code available—attack it. Failing that (and assuming it's not easy to create such a test environment), see if there's a disaster

recovery site. If so, fail over to it, and then attack the original system. However, make sure you back up the original before trying anything!

Last option: again, back everything up. Add more logging for everything you try, and—for steps that allow entry to any part of the system from outside—add your own authentication.

Logging, I should note, is crucial—if you succeed, it shows the client precisely how you did it.

3. We all know that people pick bad passwords, and that they often use the same bad password across many sites. Somone proposes using the blockchain to solve this problem: for every site S, user email address M, and password P, put

$$\{S, H(M, P)\}$$

on the blockchain, where H is a strong cryptographic hash function such as SHA-3.

(a) (15 points) Is this a good idea or a bad idea? Explain.

Answer:

This is a really bad idea. It would give any attacker with a list of email accounts—and spammers obviously have such lists—an easy way to do password-guessing for site S.

(b) (5 points) Regardless of whether it's a good idea or not, would it be better if sites put

$$\{H(S, M, P)\}$$

on the blockchain instead?

Answer:

This variant has an advantage and a disadvantage, but on balance is worse. The site name S isn't listed in the entries, which makes the attack slightly harder: for each candidate email address and password, you'd have to check the hash for each possible site. But there's a big disadvantage: if the goal is to let legitimate sites check for previously used passwords, they also have to iterate across all possible sites. In other words, though attackers are only slightly inconvenienced, the scheme loses much of its utility.

4. (20 points) In class, we discussed security tokens such as U2F. Imagine a site-programmable token: when you set up an account with some web site, it gets to install its own authentication code on the token, code that is used only for that site.

Assuming that each such site uses proper cryptography—and that's not a realistic assumption!—what other security features need to be present in the token's hardware and software to protect sites' cryptographic secrets from each other and from the user.

Answer:

This question was fatally ambiguous.

My intention—note that I said that the token was programmable*—was to ask about a situation where sites could download their own* software *to the token. That is, when I wrote "install its own authentication code" I meant "install its own authentication* software*". Almost no one*

read it that way, and checks with outside experts confirmed my fear that it was ambiguous. (Aside: I started to wonder when I saw that almost no one had gotten it "right", which I regarded as implausible.)

Consequently, I'm accepting as correct any answer for either interpretation of the question.

If this scheme is adopted, any malicious (or hacked) site now has the ability to run code on a platform that holds authentication secrets for other sites. To prevent theft of these secrets, there needs to be strong isolation between the different sites' program and storage areas. The isolation needs to cover both software and hardware attacks.

5. (15 points) SSOL is more or less the only major University web site that doesn't even support, let alone require, multi-factor authentication (MFA). It's also one of the most security-sensitive, since (among other things) it's how professors submit grades.

   The reason it doesn't use MFA is that due to an old platform incompatibility with the "Common Authentication System", SSOL had to do its own authentication. Partially to compensate, there is auditing, email confirmation of major actions, etc.

   Briefly discuss the advantages and disadvantages of this approach compared with MFA. (Note: *briefly*—more than 2–3 paragraphs is too much.

   Answer:

   MFA is much better. People don't always read confirmation emails, some activities, e.g., grade changes, can be legitimate or illegitimate and hence (by themselves) can't be distinguished simply by auditing, etc. Besides (and despite my occasional complaints), faculty regularly receive emails with clickable SSOL links; there's a lot of pro-phishing training...

   On the other hand, a robust, multiplatform MFA system is not easy to deploy. SSOL was crucial long before we had things like DuoSec, U2F, etc. MFA technology is older than the web—but it would have taken a lot of coding and support, to say nothing of faculty training.

6. (15 points) You're the CSO of a company. You've recently come to believe that the application that your company relies on for most of its revenue is buggy and probably insecure. However, the last two times someone tried patching it, a year or so ago, it didn't work at all, and people had to scramble to back out the patch before the company lost too much money. There are reports online that newer patches are also problematic, and of course you don't know how many security holes are not patched.

   The task force you have assembled to study the issue has come up with three options:

   (a) Apply all patches on a test system and try very hard to get it working.

   (b) Leave the application alone, but strengthen your internal logging, monitoring, and intrusion detection, so that you'll learn if the application was penetrated

   (c) Buy a competing product that does the same thing. However, the API to this competing product is very different, so it will take a fair amount of programming effort to make it usable in your environment. Additionally, the company that created it is new and doesn't have much of a track record, though its CTO and CSO are well known and highly respected

What would you do? Why? Remember that resources are limited; you can't do everything.

Answer:

As is often the case, there are a number of ways to approach the question. This is my preferred answer, but there are other possibilities, e.g., the possibility of using the threat of switching vendors to pressure the current vendor to improve their quality.

Option (b) is my preferred first choice, because it's something you should have been doing anyway and which you need in any event As noted, even a fully patched version of your current system may still have security holes, and you have no idea about the possible competitor's security holes.

You can't do (c) immediately—it's too risky to try something like that without a lot of advance planning and testing. Remember: you're betting the company. If you get it wrong or if this code is too buggy, your company can't bring in any revenue. Still, it's something to investigate as a long-term solution—if it works well, it's the proper choice for the future. After all, you have no idea if some latent bug in your current software will make it non-functional.

If you want to try (a), first invest in a good test environment. But you have a lot of information about it, and it's all bad news—the reputation of the package was and is bad, with no sign of a turn-around.

Other useful suggestions including hardening the environment around this application, e.g., by sandboxing it.