6.857
NETWORK AND COMPUTER SECURITY
PROBLEM SET 2

AARON EPSTEIN, CORY MONROE, JOHN WANG

Problem 2.1

Problem 2.1.a. To show that e_1 is not invertible, consider the two inputs 010 and 01. The resulting encodings will be 01 concatenated with 510 zeros, which are both the same. Thus, we have found a collision. To show that e_1 is not prefix free, consider the two inputs 1 and $1000\ldots0$ where the second string is of length 513. The first input will be padded to look exactly like the second input before any encoding and the second encoding will have 511 zeros padded to the end. Thus, the first input's encoding is a prefix of the second one.

Problem 2.1.b. To show that e_2 is not prefix free, consider the two inputs 1 and $11000\ldots0$ where the second input is of length 512. The first input gets padded with a one then 510 zeros, so it looks exactly like the unpadded version of the second input. When the second input gets padded, we see that the first input becomes a prefix of it.

To show that e_2 is invertible, suppose we have any $x_1 \neq x_2$. There are two cases.

In the first case, suppose WLOG that $|x_1| < |x_2|$. In this case, the end of x_1 will be padded with a single 1 and many zeros. The 1 will be at position $|x_1| + 1$. However, x_2 will also be padded with a one and many zeros, but the one will occur at position $|x_2| + 1$. Since $|x_2| + 1 > |x_1| + 1$ and all values greater than $|x_1| + 1$ in x_1 are zero, then we know that $e_2(x_1) \neq e_2(x_2)$.

Now suppose $|x_1| = |x_2|$. If this is the case, then we are done because already the padded values will necessarily be different because they only add bits to the end of the input, and the inputs are already different.

Therefore, we see that e_2 is invertible.

Problem 2.1.c. To show that e_3 is invertible, we simply note that e_2 is invertible. Thus, for $x_1 \neq x_2$, then $e_2(x_1) \neq e_2(x_2)$, which implies that $e_2(x_1)||EOF$ is not equal to $e_2(x_2)||EOF$, which is exactly showing that e_3 is invertible.

To show that e_3 is prefix free, we note that EOF is only found at the end of the sequence. There are two cases for $x_1 \neq x_2$. Case one is that $|e_3(x_1)| = |e_3(x_2)|$, in which case $e_3(x_1) \neq e_3(x_2)$ because e_3 is invertible, which means that they are not prefixes of each other. If $|e_3(x_1)| < |e_3(x_2)|$ then the EOF symbol of the first input occurs before the EOF of the second symbol. Since EOF appears only once in an encoding, we see that $e_3(x_1)$ is necessarily not a prefix of $e_3(x_2)$ because the position of the former's EOF symbol clashes with some symbol which is not EOF in $e_3(x_2)$.

Problem 2.1.d. By the paper, we know that f is prefix free. This means that for two inputs $x \neq x'$, then $f(x)$ is not a prefix of $f(x')$ or vice versa. We know that e_3 is invertible by our previous problem. This means that for two inputs $x \neq x'$ that $e_3(x) \neq e_3(x')$. Therefore, if we take two inputs to $f(e_3(.))$, say $x_1 \neq x_2$, then we know that $y_1 = e_3(x_1)$ is not equal to $y_2 = e_3(x_2)$ by invertibility of e_3. Therefore, we know that $f(y_1)$ and $f(y_2)$ are not prefixes of each other by the prefix property of f. This shows that $e(x)$ is prefix free.

Now, we must show that $e(x)$ is invertible. We already know that $e_3(x)$ is invertible. This means that for any input x, we have $y = e_3(x)$ which is uniquely determined. Therefore, we have uniquely determined inputs to f. If f is also invertible, then $z = f(y)$ is uniquely determined, which shows the $e(x)$ is invertible. This is because z is uniquely determined by the original x input. Thus, we need to only show that f is invertible for our inputs.

To show this, consider the algorithm for f with some input from derived from e_3. In the first pass of f, two elements paired together form a number with an alphabet of size $(B+1)^2$, which can be mapped into two elements of alphabets of size $B - 3i$ and $B + 3i$ (where i is the position in the input). At this point, the elements are still invertible because the two elements $x_1 \in [B - 3i]^*$ and $x_2 \in [B + 3i]^*$ can be used to uniquely identify the number $y \in [(B+1)^2]^*$ which can then be used to uniquely identify the two numbers $a \in [B]^*$ and $b \in [B]^*$ which we originally started with. In phase 2 of f, when elements are regrouped to the set $B > (B-3i)(B+3i)$, we can still invert the value of $k_1 \in [B]^*$ and $k_2 \in [B]^*$ to x_1 and x_2 because the mapping is injective. Therefore, the final result is invertible, which finishes the proof.

Problem 2.2

Problem 2.2.a. We choose $l = d$. The identity function $f(x) = x$ is then a hash which maps $\{0,1\}^l$ to $\{0,1\}^d$. It is infeasible to find a collision because no collisions exist by the reflexive property (the mapping is one-to-one and onto). However, it is trivial to find an x such that $h(x) = y$; simply choose $x = y$.

Problem 2.2.b. A function is one-way if it is "infeasible to find an x such that $h(x) = y$ for a given y chosen at random." Therefore, if we have an algorithm that breaks one-way for h, then that function must give us an x that hashes to a given y. To break collision resistance, we simply choose some random x_1 and hash it to a y_1. If we apply our one-way breaker to y_1, we'll get either x_1 or x_2, where x_2 is the other input that hashes to y_1. Because we chose x_1 at random, we are equally likely to get x_1 or x_2. Therefore, our algorithm will find a collision with 50 percent probability, which is certainly non-negligable.

Problem 2.3

Problem 2.3.a. After analyzing the code we realized we could simply overflow the password user input and fill the expected password hash value with our own SHA1 hash value. Each time a method is called, a so called stack frame is placed onto the stack in memory. The stack frame is organized such that all local variables are located right next to each other. Because the stack grows up, and the expected value of hashed password was defined right above the pasword user input, all we needed to do was create a small C script that could:

(1) Create a password that was the max length of the password buffer
(2) Take that password and create a SHA1 hash of it
(3) Concatenate these values (with a null terminator in between both values)
(4) Use this string as our pasword input.

Problem 2.3.b. Introducing a random canary variable after the return address should not prevent our approach from working. By introducing a random canary value, the code will check whether or not this value has been overwritten at runtime. Our approach only smashes the expectedhash value. We do not try and overwrite the return adress or any other variables besides one local variable; therefore, this value should not be changed.

Problem 2.4

Pass-The-Hash-Toolkit Attack. The Pass-The-Hash-Toolkit attack is an attack which focuses on Windows operating systems. The attack allows an attacker to pass a username and the hash of the user's password to the server to gain access to parts of the system. Instead of obtaining the plaintext password, it is sufficient in this context for the attacker to obtained the hashed version of the password. The attack is carried out by mimicking the code that sends logon credentials to the server.

In Windows, the logon credentials are stored in a linked list, each credential is a Struct that contains a username and hashed password (and other information). The attacker only needs to obtain a pointer to the system's LsaLogonSessionArray (the array that stores the logon credentials to be sent the server). Once the attacker has access to this array, he can append a new logon credential with a username and hashed password, which will log the attacker into the system even without the plaintext password.

To prevent this type of attack, one could make sure to encrypt all hashes as soon as they are created. This will prevent an attacker from being able to listen in on a communication line and obtain a hashed password, which would force the hacker to use brute force attempts to obtain the password. Second, one could make it harder to access the LsaLogonSessionArray except for the intended program. This could be done by randomly choosing a memory location to store the array and only allowing the intended program to store the reference to the array (making sure the reference location is also randomly chosen in memory).

Phising. One major attack that deserves attention is phishing. In these attacks, APT1 attackers send out emails that appear to be from company officials or customers that contain malicious attachments. These are effective because they attackers research who is most likely to send an email, and name the attachments so that they look genuine. There are several steps we can take to address this issue. As a cheap and immediate measure, we should make all employees aware of the issue by having managers inform them at their next meeting. However, more rigorous solutions involve securing our mail system. Because the danger is in the attachments and not the email, we could make it company policy (enforced by the mail client) to not send attachments at all, but share files through a central ftp server. In this case, we might prevent some of the phishing, but as we'd still need to open attachments from clients and send attachments to remote workers, an informed attacker could circumvent this policy. In addition, this may stifle collaboration between teams. I think the proper solution is to have cryptographically sign our emails. While employees can still send and receive emails from any source, they should disregard and report any email that appears to be from within the company but isn't properly signed. To deal with the emails that aren't from within the company, we'll need to license virus scanning software, which should be cheap with respect to the damage of an attack. In all cases, employees should be prevented from downloading executable attachments, as executables are never distributed through email.