Solution Set #6

CMSC/Math 456
Instructor: Daniel Gottesman

Due on Gradscope, Thursday, Nov. 10, 11:59 PM

Overview: The goal of this problem is to break a MAC. In problems 1 and 2, you will be attacking a MAC for small messages and for problem 3, you will attack it when used with a hash-and-MAC protocol.

Important reminder: You may collaborate on the problem set, but you *must* include a comment in your submission indicating anyone you worked with. You should also be writing your own code, not modeling it directly off of someone else's.

Instructions: You must upload to Gradescope a single Python file named "forge.py" containing all the functions with the specifications given below in the 3 problems. The file forge.py contains Python functions that implement the MAC and hash functions used for this problem. It also includes named functions with sample attacks (which don't succeed) for you to fill in the functions needed. You may use these functions to analyze the MAC and hash function, to test your attacks, and as subroutines in the attacks. You may also reuse code from them in your scripts.

Do not rely on uploading your file at the last minute. If there is a problem with your upload or a bug in your code that prevents the autograder from working, you will be left with a 0 score.

Scoring: Gradescope will pass your files through an autograder to give you a score and some feedback right away. You can resubmit as many times as you like to try to improve your score. However, I strongly recommend you do testing on your own system rather than try it repeatedly on the autograder, as it will be faster and consumes less communal resources.

Also, note that if we discover a bug in the autograder after the assignment is opened, it is possible we will have to rerun the autograder, in which case scores could potentially change.

The MAC: This description of how the MAC works is provided to help you understand it, but the true definition should be considered the Python code provided in forge.py, with one exception: The key in the true protocol is generated randomly.

The key k is a 16×16 array of numbers, each in the range 0 to 10,000. A message is a pair (m, n) with m and n both ranging from 0 to 15. Thus the pair specifies a location in the array. A tag is also a pair (s, t) given by the following formulas:

$$s = \sum_{i=0}^{m} k[i, n] \tag{1}$$

$$t = \sum_{j=0}^{n} k[m, j]. \tag{2}$$

The hash function and the hash-and-MAC protocol: Again, this description of the hash function is provided to help you understand, with the true definition given by the Python code in forge.py.

The hash function $H(x)$ is given by the Merkle-Damgard construction applied to a compression function $h(z,x)$ described below. The Merkle-Damgard construction was described in class and is in the book as well, sec. 6.2. The inputs and outputs x, z, and $h(z,x)$ are provided as lists of 6 bytes (i.e., integers in the range 0 to 255). Then

$$h(z,x)[i] = z[i] * x[i] + x[i+1] \bmod 256. \tag{3}$$

Here the indices are considered to be cyclic, so the index $i+1$ should be taken mod 6.

To tag a message m, m is padded as standard for the Merkle-Damgaard construction with a 1, then 0s enough to fill an even number of 6 byte blocks, and finally 2 bytes giving the length of the message in bytes, and then put through the hash function (i.e., through the compression function repeatedly). The initial value of z is a list of 6 bytes, all equal to 17.

Each byte in the output $H(m)$ of the hash function is broken up into a pair of two numbers 0–15 by taking the first and last 4 bits. This is in the form of a message above. Each such pair is used to generate a single tag t_i $(i = 1, \ldots 6)$.

The tag for the message m is then $(H(m), (t_1, t_2, t_3, t_4, t_5, t_6))$.

Python note: For those less familiar with Python, the name of the true MAC function (with the key which you are supposed to use to forge messages) is going to be passed to your attack functions as a variable named either MAC or MD in the example functions. You can call it by just treating it as a function named MAC or MD; the true name will be substituted in for that variable name.

Problem #1. Forge any Message (20 points)

For this problem, create a function "selective_forge(MAC)" which takes a function name as input MAC. The MAC function takes a message (as a pair (m,n), two numbers 0 to 15) and returns a tag (again a pair of two integers, although likely much larger than 15).

You may query the MAC one or more times and must return a pair (m,t) of a message m and a tag t. The tag should be a correct tag of the message you provided. The message can be anything of your choice, except that it cannot be a message that you used to query the MAC.

You may query the MAC as many times as you like but your score will be based on how many times you queried the MAC. If you successfully forged a tag with arbitrarily many queries, you will receive 10 points. This increases to 15 points if you can use less than 10 queries and increases further with fewer queries, to a maximum of 20 if you can match (or beat) the best attack we know.

Solution: There a variety of ways to gather enough information to forge a message, but all rely on using the information achievable from the two directions. The smallest number of queries needed is 1, which can be achieved when forging the message $(0,0)$. To do this, query the MAC for the message $(0,1)$. ($(1,0)$ would also work.) The tag for $(0,1)$ is (s,t), with $s = k[0,1]$ and $t = k[0,0] + k[0,1]$. Thus, we see that $k[0,0] = t - s$. Since the tag for $(0,0)$ is supposed to be $(k[0,0], k[0,0])$, we can return the tag $(t-s, t-s)$ and the forgery will succeed.

For ways to forge other messages, see the solution to problem 2.

Problem #2. Forge Specific Messages (20 points)

Define a function "universal_forge(MAC, msg)" that takes two inputs, a function name as input MAC and a message (a pair of two numbers from 0 to 15) as input msg.

You may query the MAC as in problem 1, but you now return only a tag (again, a pair of two integers). The tag must be a valid tag for the message your function received as input.

Be sure your function succeeds no matter which message you are passed. The exception is the message $(15,15)$, which cannot be forged. You will likely need to deal with multiple cases.

You will be asked to forge messages for 4 different cases. Each is worth 3 points for a successful forgery, increasing to 5 points if you can use the minimum number of queries for the best attack we know.

Solution: The autograder tested solutions on 4 cases: 1) message is $(0,0)$, 2) message is of the form $(0,n)$ or $(m,0)$ $(0 < m,n < 15)$, 3) message is of the form $(15,n)$ or $(m,15)$ $(0 < m,n < 15)$, and 4) message is of the form (m,n) $(0 < m,n < 15)$. The solution for case 1 is given for problem 1; it can be solved with just 1 query.

For case 2, the forgery can be done using 2 queries, which I will illustrate for the case $(m,0)$. The queries are to $(m+1,0)$, giving tag (s_1,t_1), and to $(m-1,0)$, giving tag (s_2,t_2). We wish to find the tag (s,t) for the message $(m,0)$. Applying the formulas for the tags, we have that

$$s_1 = \sum_{i=0}^{m+1} k[i,0] \tag{4}$$

$$s_2 = \sum_{i=0}^{m-1} k[i,0] \tag{5}$$

$$s = \sum_{i=0}^{m} k[i,0]. \tag{6}$$

We also have that $t_1 = k[m+1,0]$ and $t = k[m,0]$. We can thus conclude that $s = s_1 - t_1$ and that $t = s - s_2 = s_1 - s_2 - t_1$.

For case 3, we consider the example $(15,n)$. We need 4 queries, which will be to $(15,n+1)$ (with tag (s_1,t_1)), $(14,n+1)$ (with tag (s_2,t_2)), $(14,n)$ (with tag (s_3,t_3)), and $(15,n-1)$ (with tag (s_4,t_4)). Call the tag for $(15,n)$ (s,t). Then

$$s_1 = \sum_{i=0}^{15} k[i,n+1] \tag{7}$$

$$s_2 = \sum_{i=0}^{14} k[i,n+1] \tag{8}$$

$$s = \sum_{i=0}^{15} k[i,n] \tag{9}$$

$$s_3 = \sum_{i=0}^{14} k[i,n] \tag{10}$$

This allows us to determine that $k[15,n+1] = s_1 - s_2$ and $s = s_3 + k[15,n]$. We also have

$$t_1 = \sum_{j=0}^{n+1} k[15,j] \tag{11}$$

$$t_4 = \sum_{j=0}^{n-1} k[15,j] \tag{12}$$

$$t = \sum_{j=0}^{n} k[15,j] \tag{13}$$

Thus, $t = t_1 - k[15,n+1] = t_1 - (s_1 - s_2)$. We also see that $t - t_4 = k[15,n]$, so we have that $s = s_3 + (t - t_4) = s_3 + s_2 + t_1 - s_1 - t_4$.

For case 4, we consider the message (m,n) and try to get the tag (s,t). We again need 4 queries, which will be to $(m,n+1)$ (with tag (s_1,t_1)), $(m-1,n+1)$ (with tag (s_2,t_2)), $(m+1,n)$ (with tag (s_3,t_3)), and to $(m+1,n-1)$ (with tag (s_4,t_4)). Along the same lines as the above arguments, we find that $k[m, n+1] = .s_1 - s_2$ and that $k[m+1, n] = t_3 - t_4$. We also have that $t_1 = t + k[m, n+1]$ and $s_3 = s + k[m+1, n]$. Putting this together, we get that

$$(s, t) = (s_3 - t_3 + t_4, t_1 - s_1 + s_2). \tag{14}$$

There was another case which the programs weren't tested on, namely the $(15, 0)$ or $(0, 15)$ case. This can be done in 3 queries, for instance $(15, 1)$, $(14, 1)$, and $(14, 0)$ to forge the message $(15, 0)$.

Problem #3. Forge a Message for Hash-and-MAC (20 points)

Define a function "md_forge(MD)" that takes a single input, a function name MD which calls the hash-and-MAC protocol. The MD function takes as input a message which is now a list of up to 2^{16} bytes and returns a tag for that message.

You may query MD and must return a (message, tag) pair. The message is now a list of bytes (max length 2^{16}) and the tag is a list of 6 bytes plus 6 pairs of integers. The message you return cannot be one of the messages you queried.

You again get 10 points for a successfully forgery, rising with fewer queries to a maximum of 20 points with 1 query.

Solution: There were many possible different approaches to this problem. One possibility is to treat the hash function as a black box and try querying it until you get a set of outputs that let you forge a message, for instance using one of the solutions from problem 2. You will need to let one of the hash function outputs be the message to be forged and the others be the queries. This works, but needs a number of queries. Note that the planning for this is done completely off-line, and only the actual queries themselves need to be included in the submitted attack.

A better solution, requiring only 1 query, is to find a collision in the hash function, two inputs that give the same output. Again, this can be done off-line. Given the parameters in the hash function, it is workable to do this by brute force or with only minimal optimization, but there are a variety of cleverer approaches.

For instance, here is a strategy that will let you find a collision for any input message m: Our goal is to find an alternate m' that gives the same hash value as m.

First, note that for fixed $z[i]$, $h(z, x)$ is defined by a set of linear equations mod 256 from $x[i]$. A particularly simple case is when $z[i] = 0$ for all i. Then we have

$$h(z, x)[i] = x[i+1]. \tag{15}$$

If we can force $z[i] = 0$, we can easily achieve any desired output for $h(z, x)$ by putting in the appropriate input for x. We can use this strategy for all the middle calls for h. In particular, the next-to-last h has some output which we need to determine as the z input for the last call to h and the input will be x which is the desired z output shifted by one entry. We can then insert any desired number of 6-byte blocks which are all 0 to ensure that the next z output will be 0 as well.

Unfortunately, it won't work for the first and last h calls in the Merkle-Damgard chain. The first one has $z[i] = 17$ for all i as the input. If we wish $h(z, x)[i] = 0$ (so that the next z is all 0), then we have

$$0 = 17 * x[i] + x[i+1] \bmod 256. \tag{16}$$

This again can be easily achieved by having $x[i] = 0$.

This only leaves the final block, which includes some padding. The last two bytes of x, $x[4]$ and $x[5]$, are reserved for the length of the original unpadded message m'. One of the other entries of x must be 1, and then the entries following that must be 0. The ones before that we can choose as part of the message m'. And we have a specific value we want the output of h on the last block of input to give, namely the hash $H(m)$. We can pick arbitrary input z to allow this to happen and then ensure that using the previous message block as discussed above.

The equations will be simpler if we pick a convenient message length. If the length is 261, then we don't need to pad with any 0s, so we have $x[3] = 1$, $x[4] = 1$ and $x[5] = 5$. We can then pick $x[0]$, $x[1]$, $x[2]$ as desired as part of m'. We might as well choose $x[0] = x[1] = x[2] = 1$. We then have

$$\begin{align} H(m)[0] &= z[0] + 1 \bmod 256 \tag{17}\\ H(m)[1] &= z[1] + 1 \bmod 256 \tag{18}\\ H(m)[2] &= z[2] + 1 \bmod 256 \tag{19}\\ H(m)[3] &= z[3] + 1 \bmod 256 \tag{20}\\ H(m)[4] &= z[4] + 5 \bmod 256 \tag{21}\\ H(m)[5] &= z[5] * 5 + 1 \bmod 256 \tag{22}\\ & \tag{23} \end{align}$$

These equations are easily solved for $z[i]$. For $z[5]$, we need to divide by 5, but we can do that because 5 is relatively prime to 256.

In summary, we can find m' such that $H(m') = H(m)$ as follows: m' is a list of 252 0 bytes followed by 6 bytes determined by $H(m)$ as described above, then finally 3 bytes equal to 1. In the event that this is actually equal to m, we can pick different values for $x[0]$, $x[1]$, or $x[2]$ in the last block. For instance, we could have $x[0] = 3$ with $x[1] = x[2] = 1$ still. The only requirement is that they be relatively prime to 256, i.e., odd.