Exercise Set 3

helge206 Andrew Helgeson
wayt0012 Ethan Waytas
maure113 Brian Maurer

1.
(a) First of all, core network routers which have to handle very large amounts of data operate at the network layer for efficiency. It would be infeasible to keep track of all TCP connection state in a core network if you want to maintain the same traffic workload.

From a security standpoint you are still depending on the end hosts to cooperate. If both end hosts are in cahoots they could both agree to ignore RST packets and thus continue sending at their full rate. But, if one or both of the end hosts does respect the RST packet then this won't work.

If a particular host still wanted to send lots of data they could unfairly open many more TCP connections. TCP aims for service fairness on the level of a connection, not a host. So if one host opens many more TCP connections than someone else they should be given a higher percentage of the link's capacity. Each connection can obey the congestion control to prevent connection reset and their aggregate capacity could reach the speed it was before, while ignoring the congestion control.

If you are capable of doing so on both ends of the connection you could communicate with UDP. Since Bob's mechanism is based on TCP connections you can easily avoid this by using the connectionless UDP service. To get reliable transport you will need to build some TCP like retransmission on top of UDP.

The data for a given TCP connection is not guaranteed to follow the same path through the same routers. The adversary can alternate the path it uses to send packets so the view of the host's traffic is below the threshold to any given router.

(b) This still doesn't prevent an adversary from using a UDP protocol which avoids the connection state information of TCP which this method depends on to find unfair end hosts.

From the network administrator and router's point of view this would not be a scalable solution. Along with tracking TCP connection states it would incur too much overhead to maintain and check a blacklist of addresses. You could implement a policy of rolling over the list or keeping it to a certain length but then you reduce its effectiveness and adversaries can take advantage of this.

The majority of end hosts who are not using a static IP address but rather assigned one using

DHCP or similar through their ISP would create major problems when they get blacklisted. If an adversary gets blacklisted under a particular IP they can simply obtain a new one and continue on. The person who gets assigned their old IP is out of luck.

Along the lines of the previous example, a malicious user could connect to someone else's network that is using NAT and spam packets until the network's IP address is blacklisted. This is essentially a DOS attack.

Another DOS attack would involve spoofing source IP addresses. If you assume that the router is storing TCP connection state as a 4 tuple of (src addr:src port, dst addr:dst port) then it is feasible that an attacker could spoof the source address and port to match someone else who is using the router for transmission. When the router decides to blacklist the connection for unfairness it would end up blacklisting someone else's connection. Besides being a DOS on someone else's connection, this would also allow the adversary to continue its connection without interruption.

2.
(a)

Property 1: $A \oplus A = 0^{|A|}$
Property 2: $A \oplus 0^{|A|} = A$
Additionally, cbc-MAC uses an initialization vector of 0^x, where x is the size of a block (in this case x=128).

Alice first sends Bob T_0= $cbcMAC_{Ka}(0^{128})$; the entire message is $(0^{128}, T_0)$. Bob checks this message and is then convinced that Alice has sent him a message. For later it is important to note that T_0 signs a message of 0^{128}; we can use this knowledge to construct a valid message M' != 0^{128}.

A message M that would be accepted by Bob as from Alice can be constructed, even though Alice did not mean to say M. For example, assume Alice had previously sent (M, T_M) to Bob. Can we construct a message that Alice never sent, but appears as if she did? See below:

1. Use message M such that |M| = 128 (the block size).
2. Concatenate M with T_M to create a message MT_M=M' where |M'| = 256
3. Send M', T_0 to Bob, who will accept this message thinking Alice sent it

Why does this work? Well, it all happens because of how the $cbc\text{-}MAC_{Ka}$ encryption works. Below is a sample provided in lecture:

CBC-Encrypt(Block[] P, Block[] C) :

```
C[0] = 0^128;
for(i=1; i < P.len; i++)
        C[i] = Encrypt(P[i] XOR C[i-1]);
```

As each “chunk” of plaintext (P) is read in 128 bit sections, it is XOR’ed with the previous ciphertext (C) and then encrypted using some key. The MAC then becomes the last 128 bits of ciphertext. When T_0 is calculated, the initialization vector is 0^{128}, and the plaintext message is 0^{128}. So by doing $0^{128} \oplus 0^{128} = 0^{128}$ and then encrypting, we get a 128 bit encrypted message. Most likely, an attacker would not be able to decrypt this string, but the attacker does know, due to how Sly’s protocol works, that the plaintext for the encrypted message is 0^{128}. Using this fact, as mentioned above, the attacker can construct a string M’ and have it appear to be from Alice. To show this, we recall that |M| = 128 and that MT_M=M’ (concatenated) and |M’| = 256. So, in CBC-Encrypt, P[1] = M and P[2] = T_M (P is 1 indexed so P[1] is the first plaintext block). C[0] will be 0^{128}.

1. The first step is to XOR P[1] and C[0]. Due to *Property 1*, this will just be P[1] = M. M is then encrypted so that C[1] = Encrypted(M). Now, originally T_M = cbc-MAC_{Ka}(M), where |M| = 128. In short, T_M is really just Encrypted(M).
2. After step 1, C[0] = Encrypted(M), P[2] = T_M = Encrypted(M). When C[1] is XOR’ed with P[2], due to *Property 2*, the result will be 0^{128}
3. The structure of the ciphertext will be C[0] = Encrypted(M) and C[1] = Encrytped(0^{128}). The MAC will be assigned the last 128 bits, or in this case C[1] = Encrypted(0^{128}).

From the initial message T_0, we know that this corresponds to a MAC of Encrypted(0^{128}). So when Bob is comparing the MAC’s, he will find the MAC in Step 3 to be equal to the MAC sent and accept the message as from Alice (in this case (M’, T_0)). However, Alice never sent this specific message, so this is bad.

Why is it bad? Even if there is only one case where that “random garble” turns out to be something meaningful this is bad. This means that the receiver will be reading something that they believe came from one source but did not. It is also just bad that we can impersonate somebody even though we might not be able to send deliberately chosen malicious messages. This can reduce the trust in or perceived integrity of spoofed sender, i.e. Alice.

(b)
Sally’s method of authentication and encryption can be exploited just as easily and can in fact have bigger consequences than in 2.a. The consequences are bigger because Eve can insert her own message rather than simply creating a new one with random bits.

So how does it work?
Referencing the CTR Encryption algorithm from class:

CTR-Encrypt(Block[] P, Block[] C):

C[0] = randBits(BlockLen)

for (i=1; i<C.len; i++)

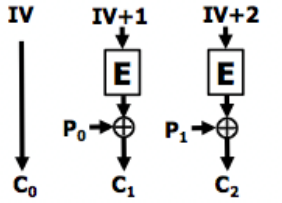
C[i] = Encrypt(C[0]+i) ⊕ P[i]

CTR-Decrypt(Block[] P, Block[] C):

for (i=1; i<C.len; i++)

P[i] = Encrypt(C[0]+i) ⊕ C[i]

Counter Mode (CTR)



To begin this attack there are a few things Eve needs to know:

- She needs the ciphertext, call it C, for some encrypted pair (M, H(M)). Given the described encryption this will be 512 bits.
- Eve also need to know the plaintext for message M.
- Eve also needs to have a message M' that she wants to replace M with.

To begin this attack we notice that in CTR mode encryption, the plaintext is used only after the IV+c has been encrypted. This means that to recreate a valid message block you only need to know what this encrypted IV is for a particular block. Once you know this you can XoR a different piece of plaintext with the encrypted IV to get valid ciphertext for your chosen plaintext.

So here are the steps Eve takes to create a new message:

Note: The ciphertext for the new message will be denoted by C'[]

- Eve takes the block of ciphertext corresponding to M (C[1] –> 256 bits) and XoR's it with her known plaintext.

  Encrypt(IV + 1) = C[1] ⊕ M

- Now to create the new message, Eve XoR's her new message with the encrypted IV discovered in the previous step.

  C'[1] = Encrypt(IV + 1) ⊕ M'

- Now we need to find the plaintext for the last 256 bits of the ciphertext which is simply H(M) and we know M so this won't be difficult. Here is how Eve finds the encrypted IV for the last 256 bits and creates new valid ciphertext

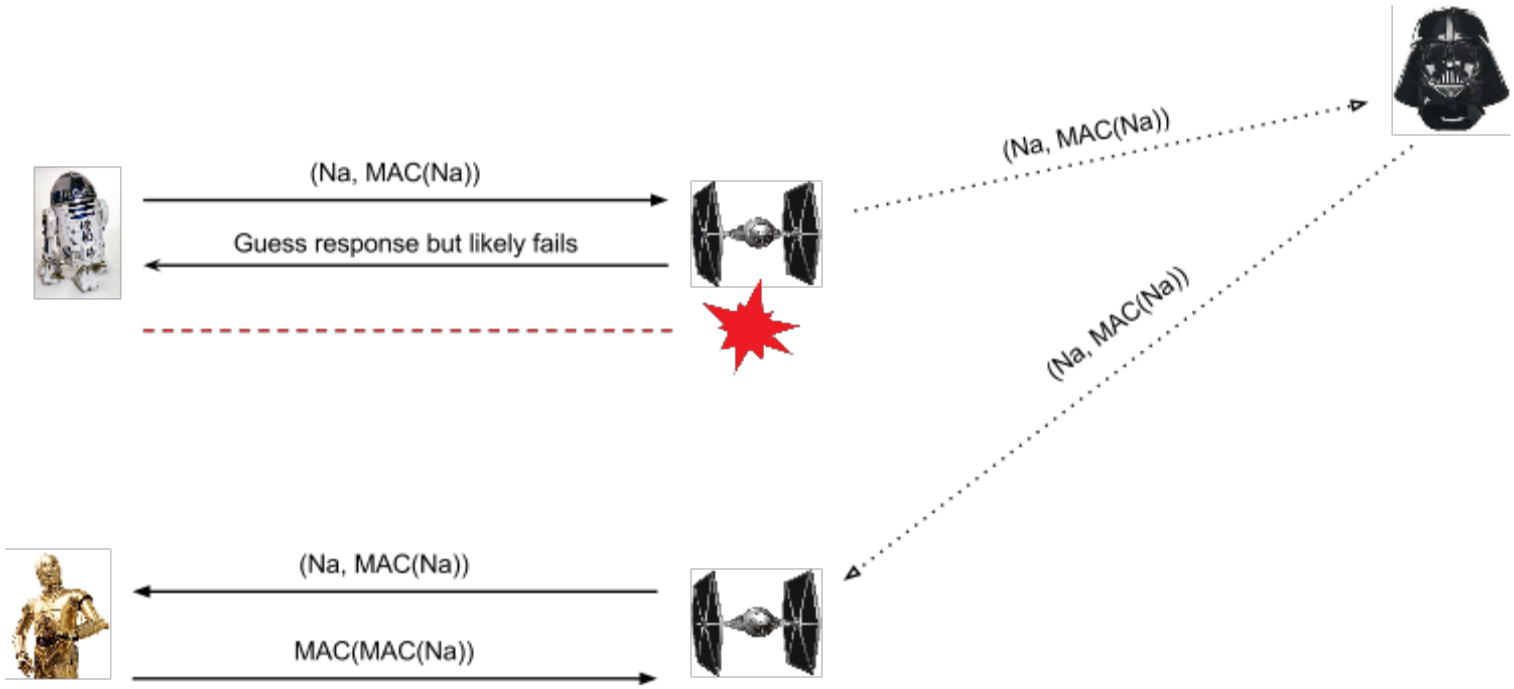
  Encrypt(IV + 2) = C[2] ⊕ H(M)

$$C'[2] = \text{Encrypt}(IV + 2) \oplus H(M')$$

- C[0] = C'[0] . We don't what to change what the IV is since we don't know the key to encrypt.

After these steps, Eve will have a new message pair (M', H(M')) that has been generated under the same encrypted IV's as Alice originally sent so they will be accepted by Bob. This is an even more devastating attack than the one in 2a since an attacker can insert any plaintext they want that fits within the block size.

3
(a)



Note: Above it should really be N_a and $MAC_k(...)$ but we cannot make subscripts in the diagram.

Here is a description of the displayed sequence of events for this attack.

1. First the emperor sends a fighter to meet artoo and receive a challenge. This fighter transmits the challenge (N_a, $MAC_k(N_a)$) back to the emperor. This fighter doesn't know the shared key *k* so he can only guess a response. He will most likely be blown up by artoo.
2. Armed with a saved challenge, the emperor sends another fighter (or he could go himself) to meet C3P0.
3. This fighter will initiate the protocol with C3P0 using the challenge previously obtained from artoo (call it (n,t)). Since this is a valid message from artoo, C3P0 will find that

$MAC_k(n) = t = MAC_k(N_a)$ so C3P0 will accept the TIE fighter as being artoo.

4. C3P0 can send back the next challenge response but the TIE fighter doesn't really care since it has already fooled C3P0.

(b)
After observing one instance of the authentication protocol, the emperor has two key pieces of information to use in further exploits, namely t and t'. In the context of the overheard authentication, here is what t and t' are equivalent to.

$t = MAC_k(N_a)$
$t' = MAC_k(MAC_k(N_a))$
and $t \neq N_a$, $t' \neq t$ (at least very unlikely so)

To later fool C3P0 into thinking he is artoo, the emperor can initiate the protocol with C3P0 using (t, t'). C3P0 will take $MAC_k(t)$ and compare it to *t'* and they will equal since *t'* is the MAC generated from *t* in the overheard authentication. At this point C3P0 will have been fooled since this is a valid challenge and with a very low probability has *t* been used as the random initial string before.

Once C3P0 is fooled he will send back a challenge response, call it t''. The emperor needs to save this for use later in spoofing. Later on when he needs to trick C3P0 again he can use (t', t'') as the protocol's new initial challenge. This will continue to work until there is a collision in the output space of the MAC function. If their *t-bit* output is significantly long this spoofing should last the emperor long enough for any purpose he needs.

Put more formally, here is how the emperor would use the initial values to continue spoofing:

$t_0 = MAC_k(N_a)$
$t_1 = MAC_k(MAC_k(N_a))$

On the i^{th} initialization of the protocol, this is the challenge to use:

challenge = (t_i , t_{i+1})
t_{i+2} = challenge response, $MAC_k(t_{i+1})$, received from C3P0

4.
(a) Alice likely would not want to sign any random message given to her. If she did so, it would give any person with the capability of requesting signatures from her the same power as Alice herself. If Alice makes no distinction between messages she should or shouldn't sign then she is allowing anyone else to act (sign) as her. As said in the problem statement we will just assume

for the remainder of this question that Alice would just sign a given random message.

There is also not much risk involved in allowing Eve to claim Alice signed a different random message. If you assume the messages are truly random then there likely isn't any meaning attached to them (or at least a very small chance that there is) so there isn't much that Eve could get from using either.

The *troublesome* part is the ease with which these two colliding hashes could be found. Using the birthday theorem approximation, for the 100-bit and 120-bit hashes described it would take on average around 2^{50} and 2^{60} computations respectively before a collision is found. So even if Eve couldn't do something malicious like drain your bank account it would still be troubling to think that this sort of forgery could happen with a relatively low amount of computation. In class it was said that 2^{50} would take roughly $1000 for a week on EC2 to crack, not much.

(b) From the question statement we are told the hash function *H* has an output length of 100-120 bits. For answering this question we will use a hash function with a fixed output size of 120 bits. The process for the birthday attack would be identical but likely require less time for the small hash outputs. Below is the size of the output space for the 120 bit hash output as well as the number of hashes we would likely need to compute before finding a free collision.

120-bit *H(m)* -> 2^{120}

> According to the birthday paradox(theorem) it would take 2^{60} hashes before a free collision was detected. This is using the $\sqrt{n}$ approximation for the 50% match probability. As we show below, the birthday theorem cannot be directly applied to this attack using 2^{60} as the target since we are concerned with pairs of values (*favorable, undesirable*).

We will start with the *favorable* and *undesirable* messages given in the question which are:

> *favorable*: "I will pay $5 to McDonald's for my lunch."
>
> *undesirable:* "I will pay $500,000 to Eve for her lunch."

Our goal is to generate many more messages until we find a corresponding (favorable, undesirable) pair with the same hash. We will generate new messages to test by varying the spacing between the words in the favorable and undesirable messages. We assume that from Alice's perspective this does not invalidate the favorable message.

As given in the question, if we vary the favorable message between 1 or 2 spaces, we end up with 2^8 or 256 different favorable messages. According to the birthday theorem we need to compute approximately 2^{60} hashes before we are likely to find a collision when not considering what side a message falls on, favorable or undesirable. Since we only want pairs that are opposite types, (*favorable, unfavorable)*, when applying the birthday attack we are actually

concerned with the number of pairs. In order to generate 2^{60} pairs of hashes we will need to produce 2^{61} hashes. Here is how we will generate this amount of hashes from the given messages above.

Variations of the *favorable* and *undesirable* messages will be generated by varying the spaces between words from 1-14 spaces. Each of these messages has 8 spaces between words (before our modification). This will give us 14^8 valid variations of the *favorable* message and another 14^8 for the *undesirable* message. 14^8 is approximately $2^{30.5}$ so after creating these variations we have a total of ~2^{61} hashes in total and ~2^{60} pairs of hashes, all of which are of the form (*favorable, undesirable*).

Using the birthday theorem approximation for the 50% collision probability we find that we need to compute 2^{60} hashes, or in this case pairs of hashes. We have shown that if we are allowed to vary the spacing between words in a message and retain its validity, we can create these 2^{60} hash pairs and give us a good probability of finding a collision.

(c) Once Eve finds the (*favorable*, *unfavorable*) message pair with the same hash, here is how the attack would proceed. Eve would present Alice with the *favorable* message. Alice will see nothing wrong with it and will hash/sign it. Eve can then take this signature and combine it with the *unfavorable* message and send it to someone such as the bank. The bank will verify the signature is Alice's and after recovering the hash they will also see that the *unfavorable* message's hash matches the one that was signed. Then Eve will be $500,000 richer assuming Alice had that laying around and the bank does not have any other security mechanisms.

Here is a more formal description of the attack:

Legend:

σK - Alice's private signing key

vK **-** Alice's public signature verification key

$H(m)$ - A cryptographic hash function which takes message *m* and produces a digest

Sig(m, σK) - A digital signature function which first applies *H(m)* and then signs the digest using the σK signing key

favorable - The favorable message in the collision pair

undesirable - The undesirable message in the collision pair

- Eve first gets Alice to sign *favorable* to obtain the value of Sig(*favorable*, σK).
- Eve then sends the following message to the bank (*undesirable*, Sig(*favorable*, σK))
- The bank verifies Sig(*favorable*, σK) using vK and also retrieves the hash *H(favorable)*

from the signature.

- The bank then finds the hash of *undesirable*, *H(undesirable)*. Since it was already found by Eve that *H(favorable) = H(undesirable)*, the bank will trust the integrity of the message and accept it.
- Eve is $rich$