CS 155
Homework 2

Rafael Moreno Ferrer
SUID#: 05490330

Problem 1. *(HTTPS vs. Signatures)*

Suppose a software company widgets.com sells a product P and wants to distribute a software update D. The company wants to ensure that its clients only install software updates published by the company. They decide to use the following approach: The company places D on its web server and designs the software P to periodically check this server for updates over HTTPS.

a) The company decides to buy a public key certificate for its web server from a reputable CA. Explain what checks P should apply to the server's certificate to defeat a network attacker. Is your design vulnerable to ssl strip?

Let S be the web server and C be the reputable CA. In my design P always makes sure that when it connects to S it is always over SSL (HTTPS). This would be a hardcoded check in the source code for P. This idea is similar to the prepopulated HSTS databases that browsers ship with, which make sure that a browser will never connect to certain websites in the clear. As we saw in class, this design (along with the following checks on S's certificate) defeats ssl-strip since the attack requires P to accept communicating with S over the clear.

Also in my design P will ship with C's public key baked in into its CA public key database. When P gets S's certificate $Cert$ in the Server-Hello HTTP response it will do the following:

1) Make sure $Cert$ is signed by a trusted CA (in this case C) and decrypt $Cert$ with C's public key (baked into P)
2) Check certificate is not expired (see $Cert$'s expiry date)
3) Check certificate has not been revoked by a CRL
4) Check that the CommonName in $Cert$ matches the domain name of S

If all of these checks pass then P will go ahead and continue with the client-key-exchange phase of the HTTPS connection setup.

b) How would you design the program P and the company's web server so that the update is secure against a network attacker, but there is no need to buy a certificate from a public CA? Your design should use an HTTPS web server as before.

In this alternative design, the program P is shipped with S's public key PK_S baked in into its CA public key database (and thus effectively considers S to be a reputable CA). S will create a self-signed certificate $Cert'$ with the same information I_S as the certificate in part (a) (expiry date, CommonName for S, **S's public key** PK_S, etc.) and self-sign it with its own private key SK_S to produce $Cert' = E_{SK_S}(I_S)$[1]. The HTTPS setup protocol will then unfold as follows:

1) P initiates a Client Hello message to S (P still is hardcoded to only open HTTPS connections to S)
2) S responds with Server Hello message and its self-signed certificate $Cert' = E_{SK_S}(I_S)$
3) P receives $Cert'$ and verifies that it came from a trusted CA, in this case S (since S's public key is baked in as a trusted CA public key). P decrypts $Cert'$ with PK_S to get $D_{PK_S}(Cert') = D_{PK_S}(E_{SK_S}(I_S)) = I_S$.
4) P does the checks listed in part (a) and proceeds only if none fail.
5) P generates a random key k, encrypts it with the public key it got from I_S which happens to be PK_S and sends $E_{PK_S}(k)$
6) S receives the message and decrypts using its private key to get $D_{SK_S}(E_{PK_S}(k)) = k$.
7) Connection setup is successfully finished and both parties proceed to communicate using symmetric key crypto with shared key k.

This approach is also secure against ssl-strip since P is also hardcoded never to connect to S unless its over HTTPS and defeats the network attacker since he cannot forge a trusted certificate since he does not possess SK_S.

[1] E_{SK} denotes the encryption function using the secret key SK and D_{PK} denotes the inverse decryption function using the corresponding public key PK.

c) Later on engineers at the company proposed the following very different design: Sign D using a widgets.com private key to obtain a signature s and then distribute (s,D) in the clear to all customers. The corresponding public key is embedded in the program P. The following two questions compare this new design to the design used in part (a):

i) If we want to distribute the patch D using a content distribution network like BitTorent, which of the two designs should we use? Explain why.

Let PK_W and SK_W be widget.com's public an private keys.

We should use the alternative design of distributing $(s, D) = (E_{SK_W}(D), D)$ and baking widgets.com public key PK_W into P.

Notice that in this situation all we care about is **a)** that P can be certain that the content D was produced by widgets.com and **b)** to ensure the integrity of D. First, note that we get these two properties by distributing (s, D) and having P verify that they "match". If P received (s', D'), in order to verify D' originated from widgets.com and has integrity it needs to decrypt s' with PK_W and check it matches D' bit by bit. If $s' = E_{SK_W}(D')$ then

$$D_{PK_W}(s') = D_{PK_W}(E_{SK_W}(D')) = D'$$

An attacker would not be able to produce a pair (s', D') that matched since he does not possess SK_W, and if he corrupted D' then the decryption of s' would not match D'. Hence if $s' = E_{SK_W}(D')$ then with high probability P can trust D' came from widgets.com and has integrity. Therefore it must be the case that $D' = D$, i.e. P got the original patch produced by widgets.com.

Next, note that we care only about ascertaining the identity of the entity that *produced* D and not the identity of the entity that *distributed* D, i.e. we do not care about who delivers D to P as long as the two conditions above are met. In particular we do not require that the parties that distribute and deliver (s, D) to the end program be trusted. This is why a CDN of not necessarily trusted peers that transmits (s, D) over the clear meets our needs.

The first approach is unnecessary here since if P were to establish HTTPS connections to talk to the peers in the CDN then effectively we would be requiring some authentication of the peers to P. As we said we do not care who the distributors are as long as (a) and (b) above are met so we do not need to validate the identities of the peers. Not to mention the added practical disadvantages of using HTTPS for communicating with the CDN: if we used the approach for part a) all peers would need to get certificates from a trusted CA which would be expensive, time consuming, and against the nature of peer to peer networks, and if we used the approach from part b) we would need to put all the public keys of the peers into P, which would defeat the flexibility of CDNs since we would need to know all trusted peers beforehand. Both of these cases would incur the costs of per-connection crypto computations (see next point).

ii) List the most time consuming crypto operations that the widgets.com web server does in each of the two designs? It suffices to only focus on pubic-key operations such as public-key encryption/decryption and digital signature generation /verification. Which of the two designs is more efficient?

The alternative design of distributing (s, D) is more resource efficient. Basically in this approach there is a single expensive crypto operation being done offline while in the first approach there are crypto operations being incurred per every connection.

In the approach that distributes (s, D) over the clear the most time consuming crypto operation being done by the widgets.com web server is that of **signing D to produce $s = E_{SK_W}(D)$**. This cost is payed offline, upfront, and only once before the patch is distributed and never incurred again afterwards.

In contrast, in the HTTPS approach there is a lot more crypto work done. Since we are establishing encrypted connections between every instance of P and the widgets.com web server we need to incur crypto operations per connection (which grows linearly with the number of downloads). In particular if we see the HTTP preamble/setup from part (b) we will see that in step 6 (client-key-exchange phase) S receives an encrypted message $m = E_{PK_S}(k)$ containing the random symmetric key generated by P and S **has to perform $D_{SK_S}(m)$ to decrypt it**. This, in addition to the symmetric key crypto operations thereafter (and possibly self signing a certificate if we follow part (b)'s approach), is an expensive crypto operation being incurred by S per connection. Hence the first approach is more efficient.

Problem 2. *(Stealth Port Scanning)*

Recall that the IP packet header contains a 16-bit identification field that is used for assembling packet fragments. IP mandates that the identification field be unique for each packet for a given (SourceIP,DestIP) pair. A common method for implementing the identification field is to maintain a single counter that is incremented by one for every packet sent. The current value of the counter is embedded in each outgoing packet. Since this counter is used for all connections to the host we say that the host implements a global identification field.

a) Suppose a host P (whom we'll call the Patsy for reasons that will become clear later) implements a global identification field. Suppose further that P responds to ICMP ping requests. You control some other host A. How can you test if P sent a packet to anyone (other than A) within a certain one minute window? You are allowed to send your own packets to P.

Step 1: Send an ICMP echo request from A to P at time $t = 0$. P will respond with an ICMP echo response packet and A will remember the counter value X in the identification field of the response.
Step 2: At time $t = 1min$ send another ICMP echo request from A to P. P will respond with an ICMP echo response and A will remember the counter value Y in the identification field of the response.
Step 3: If $Y = X + 1$ then we conclude that P has not been talking to anyone else since it has incremented its global identification field just once during that minute for the 2nd ping response it sent us[2]. If $Y > X + 1$ then we conclude that P has sent a packet to someone other than A during that minute since it incremented its counter once for the 2nd ping response to us and at least once for a packet it sent to someone else.

b) Your goal now is to test whether a victim host V is running a server that accepts connections to port n (that is, test if V is listening on port n). You wish to hide the identity of your machine A and therefore A cannot directly send a packet to V, unless that packet contains a spoofed source IP address. Explain how to use the patsy host P to test if V accepts connections to port n.
Hint: Recall the following facts about TCP:
– A host that receives a SYN packet to an open port n sends back a SYN/ACK response to the source IP.
– A host that receives a SYN packet to a closed port n sends back a RST packet to the source IP. A host that receives a SYN/ACK packet that it is not expecting sends back a RST packet to the source IP.
– A host that receives a RST packet sends back no response.

Step 1: Send an ICMP echo request from A to P at time $t = 0$. P will respond with an ICMP echo response packet and A will remember the counter value X in the identification field of the response.
Step 2: Immediately after that at time $t = 0$, have A send a TCP SYN packet to V with spoofed source address equal to P's IP address and destination port equal to n.

When V gets the SYN packet two things can happen:
a) if V is not listening on port n then it will send a TCP RST packet destined to P. When P receives this RST packet from V it will not respond with any packets.
b) if V is listening on port n it will respond to the SYN packet with a SYN/ACK packet destined to P. When P receives the SYN/ACK packet from V it will send back to V a RST packet since it was not expecting a SYN/ACK and finally V will not respond to the RST.

The differences between these two cases is that in **a)** P does not send any packets to V and in **b)** P sends 1 packet to V. Hence, now we can use our technique from part a) to detect which case holds.

Step 3: At time $t = 1min$ have A send another ICMP echo request to P. P will respond with an ICMP echo response and A will remember the counter value Y in the identification field of the response.
Step 4: If $Y = X + 1$ then we conclude that P did not send a RST packet to V, hence did not receive a SYN/ACK from V and hence that V is not listening on port n. If $Y > X + 1$ then we *infer* that P sent a RST packet to V, which implies V sent a SYN/ACK to P, which in turn implies V is listening on port n.

Note that in this analysis we assume two things: 1) in a 1 minute interval P will have received the SYN/ACK or RST packet that V will send. This seems like a reasonable assumption. 2) in order to infer from $Y > X + 1$ that P sent a RST packet to V we assume that at least one of the $Y - X - 1$ packets sent from P to hosts other than A during that 1 minute was

[2] we ignore the possibility of Y wrapping around to $X + 1$ since it would imply P sent around 2^{16} packets in a minute which highly unlikely

a RST packet to V. This is a reasonable assumption if we know P does not get much traffic. In reality we would probably want to decrease the interval to decrease the chances of getting spurious traffic into P and we would likely try this many times in order to filter out any spurious traffic into P.

Problem 3. *(WPAD)*

WPAD is a protocol used by IE to automatically configure the browser's HTTP and HTTPS proxy settings. Before fetching its first page, IE will use DNS to locate a WPAD file, and if one is found, will use its contents to configure IE's proxy settings. If the network name for a computer is pc.cs.stanford.edu the WPAD protocol iteratively looks for wpad files at the following locations:

```
http://wpad.cs.stanford.edu/wpad.dat
http://wpad.stanford.edu/wpad.dat
http://wpad.edu/wpad.dat     (prior to 2005)
```

a) Explain what capabilities were inadvertently given to the owner of the domain wpad.edu as a result of this protocol. Explain how personal information can be exposed as a result of this issue.

The owner A of the domain wpad.edu can create a file wpad.dat on its wpad.edu server which contains something like:

```
function FindProxyForURL(url, host){
   return "PROXY <IP of a web server S under control of A>"
}
```

Now, in the case where a IE browser[3] in a computer pc.cs.stanford.edu has the setting to "Automatically detect settings" on[4] it will look for these wpad.dat files to configure the web proxy settings. If it cannot locate the wpad.dat file on *http://wpad.cs.stanford.edu/wpad.dat* and *http://wpad.stanford.edu/wpad.dat* then it will find A's *http://wpad.edu/wpad.dat* file. The above code will basically tell the browser to use the web server S controlled by A as a web proxy for all browser connections. This will effectively give A the capability to monitor (and even intercept/modify) through S traffic between the browser and the end hosts the browser want to talk to. **Hence this scheme has effectively given A the capability to become a man-in-the-middle or network attacker.**

If A decides to stay a passive network attacker it is easy to imagine that A could be able to see a lot of personal information belonging to the users it is monitoring who are surfing the web. Personal data like search queries in search engines (medically related are specially private) are sometimes transmitted over the clear. Additionally, there are a lot of websites that transmit sensitive data over the clear (some even credit card info!) that A can learn along with the user's surfing history.

If A decides to tamper with the traffic and become an active network attacker it can also perform all the other man in the middle attacks like ssl-strip, injecting malicious javascript, XSS, CSRF, etc. if the situation permits (no encryption, serving resources on the clear for SSL pages, upgrading HTTP session to HTTPS, etc).

b) Are pages served over SSL protected from the problem you described? If so, explain why; if not, explain why not.

Yes, if and only if users browsers only connect to end hosts through HTTPS. If a user browser B only connects to an end host H through HTTPS (as would happen if the HSTS header was successfully sent on first use from H to B) then the attacker could not do any harm[5]. However, if B connects to any page not over SSL, if it loads resources on a page that are not through HTTPS, or if H at some midpoint tries to upgrade an HTTP connection to B to an HTTPS connection then the attacker can perform any of the classic man in the middle attacks like ssl-strip, injecting malicious frames, Wachovia-type login attacks, and so on.

[3] IE is most vulnerable but also other browsers configure its web proxy settings using WPAD and are also vulnerable to this attack

[4] which many IE browsers used to have enabled by default

[5] if during connection setup the attacker wanted to mess with H's certificate then B would recognize the tampering when decrypting the certificate and would not initiate the connection. After receiving H's certificate B encrypts all of its traffic with PK_H and then the shared symmetric key k. A will not be able to see the traffic or tamper with it without being noticed since he does not have SK_H (and hence cannot get k either)

Problem 4. *(CSRF Defenses)*

a) In class we discussed Cross Site Request Forgery (CSRF) attacks against web sites that rely solely on cookies for session management. Briefly explain a CSRF attack on such a site

Let V be a victim browser, S be a victim site, and M be a malicious site. Suppose S relies solely on cookies for session management. A typical CSRF attack would go like this: First, V visits S routinely, authenticates himself and establishes a cookie-based session with the site S. While the session is still active, V visits is lured into visiting the malicious site M (maybe in another tab, or window, lured in from a spam email). The designer of the malicious site presumably knows very well how the victim site S works (he may have an account there and inspected the source to see what forms are used for the site to POST sensitive data, make changes). When M serves the innocent looking content it also includes a hidden iframe with a form similar to the ones in the site S which POSTs to a url in S. This form presumably is state changing, has a desired malicious affect (think transfer money, etc.), has all right parameter names and action URL, and can be filled out without user input. The iframe then contains javascript to submit the form immediately. When V's browser submits the form to S, it will include V's cookies for the site S in the HTTP headers of the request. Finally, when S gets the request it does not know in which context the request originated from and only sees V's cookie, and the POST parameters. Since S uses the cookie solely to authenticate V's session then it will think that the request came legitimately from V and will perform the request for the attacker.

b) A common CSRF defense places a token in the DOM of every page (e.g. as a hidden form element) in addition to the cookie. An HTTP request is accepted by the server only if it contains both a valid HTTP cookie header and a valid token in the POST parameters. Why does this prevent the attack from part (a)?

First, by checking the HTTP cookie header the server can identify which browser a request came from. Second, by generating a secret token that the server includes as a hidden input field for every form that it serves, we can be confident that forms that come back with a valid secret token were served by the server to the user and were not crafted by an attacker. Implicit in all of this is the assumption that secret validations tokens will be hard to guess, and hence the unguessability will provide unforgettability for forms. If an attacker wanted to perform the attack from part (a) he would be stuck when creating the form in the hidden iframe that POSTs to the victim site since he would not know what token to fill in for the secret validation field.

c) One approach to choosing a CSRF token is to choose one at random. Suppose a web site chooses the token as a fresh random string for every HTTP response. The server checks that this random string is present in the next HTTP request for that session. Does this prevent CSRF attacks? If so, explain why. If not, describe an attack.

This prevents CSRF attacks. By generating a fresh random string for every HTTP response the web site makes it practically impossible to guess what secret token a given user has been assigned at some point in time (we would also probably like secret tokens to expire at some point so that old valid secret tokens cannot be used in the future and hence do not accumulate). As we mentioned in part (b) the unguessability of the secret token provides unforgettability to the forms sent by the server and prevents against CSRF attacks. In this case a fresh random token per HTTP request provides the secret validation tokens with unguessability.

d) Another approach is to choose the token as a fixed random string chosen by the server. That is, the same random string is used as the CSRF token in all HTTP responses from the server over a given time period. You may assume that the time period is chosen carefully. Does this prevent CSRF attacks? If so, explain why. If not, describe an attack.

This does not prevent CSRF attacks. To illustrate an attack suppose a victim website S chooses a fixed CSRF token X that it includes in all of its HTTP responses from time $t = 0$ to time $t = T$. The attacker A knows this is how the server chooses its secret validation tokens. Assume that the victim browser B has already logged in to the site S during that time period. The attacker A has an account on the site S as well and will also log in the time period between $t = 0$ and $t = T$. After logging in the attacker goes to some page on the site that contains a form[6] that he is interested in using during the exploit (think a transfer money form) The attacker then looks at the source for this page, finds the hidden secret validation token X for this form and remembers it. Next, the attacker A puts a hidden iframe into his malicious website with a form for the site S (transfer money form) that is preopulated to do something malicious, includes the secret validation token with value X, and has some javascript to automatically submit the form. The attacker A then lures B to the malicious website M during the time period $0 < t < T$ and when B visits M the hidden iframe will submit the malicious form with the correct secret

[6] or any other form really

validation token for that time period and the browser's cookie for S. The request will go through and the attacker will succeed.

e) Why is the Same-Origin Policy important for the cookie-plus-token defense?

Throughout this discussion we have mentioned that the malicious website M would have to guess the secret validation token for pages in the victim site S if he wanted to mount a CSRF attack. Implicit in this is the assumption that the malicious site M cannot simply reach out and access content (the secret validation token in this case) from site S's in the victim browser B. This is guaranteed by the SOP. Without the SOP CSRF attacks would be very simple. If a victim browser B was logged into a victim site S then the malicious page M could simply contain a hidden iframe that loads a page from the site/origin S which contains the form he is interested in. The attacker could then simply use javascript to reach into that iframe's content and extract the secret validation token, put it into his own malicious form and effectively create a legitimate form. Then he would proceed as before and successfully mount a CSRF attack.

So SOP is crucial to the cookie-plus-token defense since it ensures that a malicious website M cannot simply load content from the victim site S into the victim browser B and read off the secret validation token to create a legitimate form. Without the ability to do this the attacker has to resort to guessing the token.

Problem 5. *(Firewalls and Fragmentation)*

The IP protocol supports fragmentation, allowing a packet to be broken into smaller fragments as needed and re-assembled when it reaches the destination. When a packet is fragmented it is assigned a 16-bit packet ID and then each fragment is identified by its offset within the original packet. The fragments travel to the destination as separate packets. At the destination they are grouped by their packet ID and assembled into a complete packet using the packet offset of each fragment. Every fragment contains a one bit field called "more fragments" which is set to true if this is an intermediate fragment and set to false if this is the last fragment in the packet. We discussed the way small offsets can allow the data in one fragment to overlap information in the preceding fragment. However, overlapping fragments should not occur in normal network traffic.

In class we mentioned that when fragments with overlapping segments are re-assembled at the destination, the results can vary from OS to OS. Give an example where this can cause a problem for a network-based packet filtering engine that blocks packets containing certain keywords. How should a filtering engine handle overlapping fragments to ensure that its filtering policy is not violated?

Suppose that the filtering engine wants to filter out the keyword "communism" but not the words "community" or "small". If we want to send the word "communism" to a destination IP address inside the network "censored" by the filtering engine without the filtering engine blocking it we can do the following:

Step 1: Send an IP with packet ID X, packet offset equal to 0, the more fragments bit set on, and where the payload starts with the word "community".

Step 2: Immediately after send another IP packet to the same destination, with packet ID X, the more fragments bit set on, with packet offset equal to $sizeof(IP\ header) + 7$, and where the payload starts with the word "small".

Step 3: Finally send another IP packet to the same destination with packet ID X, the more fragments bit set off, with packet offset equal to $sizeof(IP\ header) + 9$, and where the payload starts with space characters.

Since none of these packets contain blacklisted words in their payload then none will be discarded by the filtering engine. However when they are grouped and reassembled into a single IP packet, the chosen packet offsets will make them overlap and will cause them to reassemble into the word "communism". Another approach would be to have the word "communism" be at the boundary of two adjacent non-overlapping fragments (i.e. the first fragment would have the characters "commun" as the last characters of its payload and the next fragment would have "ism" as the first characters of its payload).

I have several ideas on how the filtering engine should handle overlapping fragments in order to ensure the filtering policy is not violated.

Idea 1: In IPv6 packets are never fragmented by routers (when the package size exceeds some MTU the packet is dropped and an ICMPv6 Packet too Big is sent to the source host). Considering this we could have a policy on IPv6 packets that would just drop any fragmented packets at the filtering engine (packets with the more fragments bit set on). Normal traffic should not contain fragmented packets and hence when we encounter any we can be suspicious. By disallowing any fragmentation we can make sure that the filtering policy will not be violated since the two approaches described above are based on fragmentation.

Idea 2: The filtering engine will maintain state per connection in order to determine whether a fragment (an IP packet with the more fragments bit set on) overlaps with other fragments for the same packet ID. The filtering engine will forward along fragments as normal. Only when it comes across a fragment that overlaps with other fragments, the filtering engine will discard that packet and drop in on the floor. Since overlapping fragments should not occur in normal network traffic we can safely assume that it is fine to interrupt connections where that happens. Furthermore, this approach makes sure that the filtering policy is not violated since no overlapping fragments are let through. In order to do this the filtering engine will need to remember the packet offset and the packet length for all fragments for a given packet ID for a given open connection. With this information it can determine whether a given fragment coming in overlaps with previously received fragments for that packet ID. The filtering engine can garbage collect all this state when it has received the last fragment for a packet ID in a given connection.

Idea 3: Whenever the filtering engine at the network gateway comes across a packet with the more fragments bit set on it will buffer it and wait for all other fragments with the same packet ID to arrive. When the last one arrives with the more fragments bit set off then the filtering engine will perform the reassembly of the fragments and will decide whether to filter out the reassembled packet or not depending on whether its payload contains blacklisted keywords. This approach requires the filtering engine to potentially buffer several fragments per open connection every time there is fragmentation. However,

since IP packets are at most 65,535 bytes in length at least the buffering space needed per connection would be bounded since in total all fragments cannot be more than 65,535 bytes.

Problem 6. *(DNSSEC)*

DNSSEC (DNS Security Extensions) is designed to prevent network attacks such as DNS record spoofing and cache poisoning. Generally, the DNSSEC server for example.com will posses the IP address of www.example.com. When queried about this record that it possesses, the DNSSEC server will return its answer with an associated signature. If the DNSEC server is queried about a host that does not exist, such as doesnotexist.example.com, the server uses NSEC or NSEC3 to show that the DNS server does not have an answer to the query.

Suppose a user R (a resolver, in DNS terminology) queries a DNSSEC server S, but all of the network traffic between R and S is visible to a network attacker N. The attacker N may read requests from R to S and may send packets to R that appear to originate from S.

a) Why is authenticated denial of existence necessary? To answer this question, assume that S sends the same unsigned DOES-NOT-EXIST response to any query for which it has no matching record. Describe a possible attack.

In this attack we assume N knows the format of the unsigned DOES-NOT-EXIST response that S sends for no matching records (N can simply query S for a record that does not exist–just keep trying random hostnames until it gets one–and learn this). A simple attack would be that for every DNS request that R sends to S (which N will read) we will have N respond with the unsigned DOES-NOT-EXIST response, make it look like it originated from S, and race S's response, whatever it may be. In this way, if N's response arrives at R first then N will have poisoned the DNS cache entry for that record if the record did in fact exist. If this is done over a long time period one can imagine that N would be able to poison most of R's DNS cache. All of the users that use R as a DNS resolver would then get DOES-NOT-EXIST responses for all DNS queries, resulting in a DoS attack.

b) Assume now that S cryptographically signs its DOES-NOT-EXIST response, but the response does not say what query it is a response to. How is an attack still possible?

An attack is still possible. For this attack we can also assume N knows the signed generic DOES-NOT-EXIST response that S sends for no matching records (N can simply query S for a record that does not exist–just keep trying random hostnames until it gets one–and learn this). Now that N knows this, every time he reads a DNS query from R to S he will respond with the signed generic DOES-NOT-EXIST response and make it look like it originated from S. Again, if this response wins the race against the legitimate response from S then N will poison R's DNS cache in case that record did exist. In the same way as before this can turn into a DoS attack against the users of R.

c) A DNSSEC server may send a signed NSEC response to a query that does not have a matching record (such as doesnotexist.example.com). An NSEC response contains two names, corresponding to the existent record on the server that immediately precedes the query (in lexicographic order), and the existent record that immediately follows the query. For example, if a DNSSEC server has records for a.example.com, b.example.com, and c.example.com, the NSEC response to a query for (non-existent) abc.example.com contains a.example.com and b.example.com because these come just before and just after the requested name. To be complete, NSEC records also wrap-around, so a query for a non-existent name after the last existent name will receive an NSEC containing the last and first existent names.

How should the resolver use the information contained in NSEC records to prevent the attacks you described in previous parts of this problem?

The first attack is easily defeated since the DOES-NOT-EXIST response there was not signed. Hence, if there is no signature or if the signature does not match the response the resolver R will simply discard the DOES-NOT-EXIST response. For the second attack, if R queries S for a hostname that does exists and gets a signed DOES-NOT-EXIST response but that does not specify what query it is a response to (does not specify next and previous valid hostnames) then the resolver will not cache the association (it needs to verify that the hostname is queried for is lexicographically between the prev and next hostnames in the NSEC record). Alternatively if the attacker decides to include spoofed next and pref hostnames along with the DOES NOT EXIST response then since he does not have S's private key he will not be able to generate a valid signature for the message and hence the message will be discarded by R when it verifies the signature.

d) NSEC leaks information that may be useful to attackers on the Internet. Describe how an attacker can use NSEC to enumerate all of the hosts sharing a common domain-name suffix. How is this information useful for attackers?

An attacker can "map out" all the holes (and thus enumerate all the valid hostnames) sharing a common domain-name suffix X as follows:

0) Initialize $hostname = a.X$
1) The attacker send a DNS query for *hostname*
2) If he gets back a response that is not a DOES-NOT-EXIST response he knows that *hostname* is a valid hostname, remembers it, and proceeds to 1) with *hostname* equal to the next hostname in lexicographical order
3) If he gets back a DOES-NOT-EXIST response along the *prev* and *next* valid hostnames then he immediately knows than no hostnames lexicographically between *prev* and *next* are valid hostnames, and he remembers this. Additionally he knows *prev* and *next* are valid hostnames so he remembers them. He then proceeds back to 1) with the hostname that lexicographically follows *next*.
– The algorithm terminates when the attacker tries to iterate with a *hostname* it has seen before (a valid hostname it already knows about). At this point the attacker has mapped all valid hostnames in the $.X$ domain.

Attackers may be interested in enumerating all hosts sharing a common domain-name suffix for various reasons. If the domian-name belongs to an organization they want to do a DDoS attack on with a botnet and they want to make sure they DDoS all hosts within that network are attacked then this information is invaluable. Clearly the attacker could not try to attack every single possible hostname since there are exponentially many of these so it would be wasteful of resources and the cost would be prohibitive (most random hostnames in this space will probably not exist). By knowing all the valid hostnames the attacker can target specifically the valid hostnames and do a more effective DDoS attack. Another reason is that by enumerating all hosts an attacker could review the hostnames looking for particularly revealing/interesting names to attack, like mail servers or sensitive database servers. Organizations don't like to make public the hostnames of machines containing sensitive data and attackers would love to know these.

e) NSEC3 is designed to prevent DNS responses from revealing unnecessary information. NSEC3 uses the lexicographic order of hashed records, instead of their unhashed order. In response to a query without a matching record, NSEC3 will return the hashed names that are just before and just after the hash of the query. For example, on a server containing a.example.com, b.example.com, and c.example.com, if a hashes to 30, b to 10, c to 20, and abc to 15, the NSEC3 response to a query for abc.example.com would contain 10.example.com and 20.example.com. Hashed names are also assumed to wrap around, in the same way as unhashed names in NSEC.

Explain how a resolver should verify the validity of a response under NSEC3?

First it will check the signature of the response to make sure the response came from the legitimate DNS server and has not been tampered with.[7]. If this check passes and the response is not a DOES-NOT-EXIST response then the resolver will cache the hostname-IP association. If the response was a DOES-NOT-EXIST response then the resolver will hash the name of the hostname it originally asked for (in the example above it it asked for *abc.example.com* so it will hash *abc* to get 15) and make sure that this hash lies in between (lexicographically) the next and previous valid hashes that the DOES-NOT-EXIST record includes. If it does not fall inside this interval/gap then the DNS response will be discarded. If it does fall into the gap then it will cache the association hostname: DOES-NOT-EXIST.

[7] decrypt the signature with the public key of the DNS server and check it matches the response