Paxson Spring 2017 — CS 161 Computer Security — Homework 5

Due: April 28, 2017, at 11:59pm

(Version 1.0: released April 21, 2017)

Instructions. This homework is due April 28, 2017, at 11:59pm. You *must* submit this homework electronically via Gradescope (not by any other method). When submitting to Gradescope, *for each question* your answer should either be a separate file per question, or a single file with each question's answer on a separate page. This assignment must be done on your own.

Problem 1 ***Denial-of-service via email*** **(10 points)**

One day you try to email ten of your friends at Stanford your award-winning recipe for Big Game game-day cookies to their `stanford.edu` addresses. Unfortunately you typed two of their email addresses incorrectly so you received two failed-delivery notification (FDN) messages that include both the original text from the email you sent and copies of the attachment.

(a) Knowing this, how could you launch an amplified denial of service attack against some poor unsuspecting person via email?

> **Solution:** Send an email with an extremely large attachment to a large number of incorrect Stanford.edu addresses (this will ensure that there will be a FDN message for each incorrect email address and that each FDN will include a copy of the original attachment). Spoof the 'From' email address to be from the victim of the attack. When the Stanford email servers try to deliver your emails, they will deliver a bunch of FDN emails to the spoofed from address, one FDN message per incorrect email address. If you include enough invalid email addresses (to get a high amplification factor) and use a large enough attachment (to amplify), you could overload the victim's mail server.
>
> For example, if you send a 10 MB attachment and you have 1,000 invalid email addresses, you could cause 10 GB of email to be sent to the victim's mail server. You only had to send about 10 MB of data, and the victim received 10 GB, so the amplification factor is 1,000×.

(b) How could the developer of the Stanford mail server mitigate this issue?

> **Solution:** Only send 1 FDN message. If there are multiple incorrect email addresses, send a single FDN message (it can list all of the incorrect addresses

in the FDN).

Also, we could omit the original text and/or attachments from the FDN.

(c) Sketch another form of email-based DoS attack that works even if servers all employ the fix you sketched in the previous part.

Solution: One technique that's been used is to send a provocative and/or obnoxious email message to a large number (say, hundreds) of recipients, with the email From address spoofed to appear to come from the targeted victim. In practice, often a significant proportion of the recipients will fail to recognize the original email as a spoof, and angrily respond back to the victim, enough so as to seriously inconvenience the victim. This technique is known as "Joe-jobbing."

Another approach would be to sign up the victim to a huge number of mailing lists, such that they're overwhelmed by incoming email from the lists. (This only works for lists that do not first confirm the subscription with the purported recipient, a practice that is increasingly uncommon.)

Problem 2 *DNSSEC* (20 points)

DNSSEC (DNS Security Extensions) is designed to prevent network attacks such as DNS record spoofing and cache poisoning. When queried about a record that it possesses, such as when the DNSSEC server for `example.com` is queried about the IP address of `www.example.com`, the DNSSEC server returns with its answer an associated *signature*.

For the following, suppose that a user R (a resolver, in DNS parlance) sends a query Q to 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 send packets to R that appear to originate from S.

(a) Suppose that when queried for names that do not exist, DNSSEC servers such as S simply return "No Such Domain," the same as today's non-DNSSEC servers do. This reply is not signed.

Describe an attack that N can launch given this situation.

Solution: Since the "No Such Domain" reply has no signature, it can be easily spoofed by N. As a result, N can launch a denial-of-service attack on a user's access to any domain.

(b) Suppose now that when queried for a name Q that does not exist, S returns a signed statement "Q does not exist."

1. Describe a DoS attack that N can launch given this situation.

2. Can N still launch the attack you sketched in part (a)? If so, explain how this attack would work. If not, explain why the attack no longer works.

Solution:

1. N can make repeated queries for non-existing/made-up domains. Signing each response can use up the computational resources of the server, resulting in denial of service.
2. Imagine that `foo.example.net` doesn't exist today. N makes a request for it and receives a signed response stating "`foo.example.net` doesn't exist." N stores the response. If later, `example.net` deploys a service at `foo.example.net`, N can *replay* the stored packet that states "`foo.example.net` doesn't exist" and the recipient will incorrectly believe they cannot access the new service.

(c) One approach to address the above considerations is to use NSEC Records. As mentioned in class, when using NSEC S can return a signed statement to the effect of "when sorted alphabetically, between the names N_1 and N_2 there are no other names." Then if the name represented by the query Q lexicographically falls between N_1 and N_2, this statement serves to confirm to R that there's no information associated with the name in Q.

NSEC has a shortcoming, which is that an attacker can use it to *enumerate* all of the names in the given domain that do indeed exist. To counter this threat, in the April 17 section we introduced the NSEC3 Record, which is designed to prevent DNS responses from revealing other names in the domain. NSEC3 uses the lexicographic order of *hashed* names, instead of their unhashed order. In response to a query without a matching record, NSEC3 returns a signed statement of the hashed names that come just before and just after the hash of the query.

Suppose that the server S has records for `a.cs161.com`, `b.cs161.com`, and `c.cs161.com`, but *not* for `abc.cs161.com`. In addition, assume that `a.cs161.com` hashes to `dee60f2e...`, `b.cs161.com` to `80a4cb36...`, `c.cs161.com` to `c218f96a...`, and `abc.cs161.com` to `99f3e2ba...`.

If R queries S for `abc.cs161.com`, what will S return in response? Describe how R uses this to validate that `abc.cs161.com` does not exist.

Solution: S will return "No domain exists that hashes between `80a4cb36...` and `c218f96a...`," along with details of the hashing algorithm. All of this will be signed by S.

R will hash its requested domain (in this case `abc`). Since it hashes to `99f3e2ba...`, R will know that `abc.cs161.com` doesn't exist.

(d) The hashes in NSEC3 are computed as a function of the original name plus a *salt* and an *iteration parameter*, as follows:

```
Define H(x) to be the hash of x using the Hash Algorithm selected by
the NSEC3 RR, k to be the number of Iterations, and || to indicate
concatenation. Then define:

   IH(salt, x, 0) = H(x || salt), and

   IH(salt, x, k) = H(IH(salt, x, k-1) || salt), if k > 0

Then the calculated hash of a name is

   IH(salt, name, iterations)
```

The name of the hash function, the salt and the number of iterations are all included in an NSEC3 reply (that is, they are *visible* and assumed to be easily known). All replies from a given server use the same salt value and the same number of iterations.

Suppose an attacker has a list of "names of interest," i.e., names for which they want to know whether the given name is in a particular domain. If the attacker can get all of the NSEC3 responses for that domain, can they determine whether these names exist? If so, sketch how. If not, describe why not.

Solution: The attacker can hash each name of interest, and then look at the NSEC3 responses. If the hash of a name of interest is in range of hash values that do not exist (according to the NSEC3 responses), then the attacker knows that the name of interest doesn't exist. While it is possible to perform queries for all possible names, doing so is more easily detectable.

Attackers can construct *dictionaries* mapping names of interest to their hash values, and then use these as outlined in this question. Creating a dictionary is more expensive than the attack on NSEC, however. The next two questions deal with how to make precomputed dictionaries less of a threat.

(e) Why would a domain change their *salt* value in NSEC3 replies?

Solution: A salt makes sure that an attacker can't repeatedly use the same precomputed dictionary. Attackers instead need to construct a different dictionary every time the domain changes the salt value it uses. The domain could do this fairly frequently, depending on the size of the domain and therefore the computational cost of recomputing and re-signing the NSEC3 records.

(f) What is the purpose of the *iteration parameter* in NSEC3 replies?

Solution: Higher numbers of iterations make it that much harder (computationally expensive) to create a dictionary. For example, changing the number of iterations from 1 to 100 makes the already significant amount of work needed to create a dictionary 100 times more expensive.

(g) The specification of NSEC3 also sets an upper bound on the *iteration parameter*. What could happen if that upper bound did not exist?

Solution: Without an upper bound, a malicious DNS server could return an NSEC3 reply with a huge iteration parameter, inducing a denial-of-service attack on the DNS resolver that issued the request as it tries to then compute the hash value.

Problem 3 ***Detecting Worms*** **(20 points)**

Assume that you are working for a security company that has to monitor a network link for worm traffic. The link connects a large site with the rest of the Internet, and always has lots of traffic on it. Your company sells a monitoring box that can scan individual packets for fixed strings at very high speeds.

(a) Suppose that after careful analysis you discover that a particular TCP-based worm that you need to protect against generates traffic that always contains a fixed 4-byte sequence. You program your company's specialized hardware to generate an alarm whenever it detects a packet containing this 4-byte pattern.

Explain how this detector can exhibit false negatives.

Solution: False negatives occur when the detector fails to generate an alarm for an actual instance of the worm. If the 4-byte sequence is split across two packets, for example, then the detector will miss it, since it only analyzes single packets in isolation.

(b) Propose an alternative architecture for your company's monitoring box that fixes the problem from part (a). Your alternative should not increase false positives by more than a modest amount. Does your revised approach completely eliminate false negatives? Explain why or why not.

Solution: The monitor should order and reassemble packets into the TCP "bytestream," and then run its detection on this stream. Doing so requires the ability to buffer traffic, since the bytestream might not arrive directly in sequence.

Whether this approach can exhibit false negatives is a complex question. We saw in class an attack by which the monitor could encounter multiple possible

versions of the bytestream. If the monitor alerts if *any* of these possible versions matches its signature, then it can avoid false negatives. If on the other hand it picks one particular version, then it could indeed incur a false negative.

False negatives can also arise by stressing the monitor's processing, for example by giving it a very heavy traffic stream to analyze, to the extent that it can no longer keep up with full rate of traffic. The monitor will then wind up skipping some of the traffic for its analysis. If an attack was present in the skipped traffic, then that will result in a false negative.

(c) Suppose benign network traffic is uniformly distributed in terms of its content. That is, every payload byte is chosen uniformly at random. Calculate the expected time until a 150 MB/sec (megabytes per second) link will cause a false alarm.

Solution: Let X_i be a uniform random variable that represents the value of one byte, i.e., $X_i \sim \text{Uniform}[0, 2^8 - 1]$. Further, let $X = (X_1, X_2, X_3, X_4)$ be a random vector that denotes a 4-byte sequence in the network traffic and let $s = (s_1, s_2, s_3, s_4)$ be a fixed 4-byte signature.

We assume that each byte in the network traffic is independent of the previous one. Then the probability that random network traffic matches the signature (i.e., a false alarm) is

$$\Pr[X = s] = \prod_{i=1}^{4} \Pr[X_i = s_i] = \left(\frac{1}{2^8}\right)^4 = \frac{1}{2^{32}}$$

where the first equality comes from the independence assumption, and the second from the assumption that each byte is identically distributed.

For the next step in the analysis, we idealize the network traffic as a contiguous stream of bytes, each arriving one after the other. Think of it as a sliding window where we can always match 4 contiguous bytes.

Let T be a random variable denoting the time in seconds until the first false alarm. That is, $T \sim \text{Geom}[p]$ (geometric distribution), where $p = \Pr[X = s]$. The expectation is then $\mathbb{E}[T] = 1/p = 2^{32}$ time units, where 1 time unit is one opportunity the monitor has to observe the signature. Since we monitor a link carrying 150 MB/sec, the unit of time for a single opportunity is $1/1.5 \cdot 10^{-8}$. Thus, the expectation is:

$$\mathbb{E}[T] = 2^{32} \cdot 1/1.5 \cdot 10^{-8} = 28.633\text{sec}.$$

Hence on average it will take about 28 seconds until the first false alarm.

This analysis isn't quite right, because while the individual bytes are independent, the overlapping sequences of bytes are not. To actually establish that the

above expression still holds requires more powerful analysis tools from probability (Markov chains). Given that, we will also accept answers based on non-sliding windows (for which independence readily holds). In that case, the observational opportunities occur at one-fourth the rate of that for sliding windows, and therefore the above argument applies as:

$$\mathbb{E}[T] = 2^{32} \cdot (4 \cdot 1/1.5 \cdot 10^{-8}) = 114.532 \text{ sec.}$$

We also accepted for full credit variants of these solutions that interpreted "MB" as 2^{20} bytes rather than 10^6 bytes.

(d) Assume now that the signature length is 8 bytes. Calculate the expected time until a 20 GB/sec (gigabytes per second) link will cause a false alarm.

Solution: The analysis procedure is identical to part (c); only the values change. The signature length of 8 bytes results in a false alarm probability of $p = (2^{-8})^8 = 2^{-64}$. Thus, the expected time in seconds until a false positive amounts to:

$$\mathbb{E}[T] = 2^{64} \cdot 1/2 \cdot 10^{-10} = 922{,}337{,}203.685 \text{ sec}$$

or about 29 years. (Or, if using non-sliding windows, about 233 years.)

Again we also accepted for full credit variants of these solutions that interpreted "GB" as 2^{30} bytes rather than 10^9 bytes.

(e) What could a worm author do to try to ensure that their worms do not have many fixed-byte sequences?

Solution: The worm author could incorporate one of the two general approaches we discussed in class: *polymorphism*, by which the worm reencrypts most of its code (all but the decryptor) each time it propagates; and *metamorphism*, by which the worm **rewrites** its **entire** code each time it propagates. The former will have a few fixed-byte sequences due to the presence of the decryptor; the latter in principle might not have any.

Problem 4 *Feedback* (0 points)

Optionally, feel free to include feedback. Whats the single thing we could do to make the class better? Or, what did you find most difficult or confusing from lectures or the rest of class, and what would you like to see explained better?

If you don't want to submit any feedback, just select your last page to use for this question when submitting to Gradescope.