Understanding Signature Schemes: How Data Sources Are Authenticated, Secured, & Spoofed

One of the biggest problems in data security is authentication of data and its source. How can Alice be certain that the executable in her inbox is from the venerable Bob, and not from the not-so-venerable Oscar? Clearly Alice wants to know because if this file is actually sent to her by Oscar, the file might not be a game, but a trojan that can do anything on her computer such as sift through her email and passwords, upload her honeymoon pictures, or even turn on microphones and webcams. This is where I think a real life story might be useful for illustration.

In mid-2012, I read an interesting story from my PacketStorm feed. Apparently a new worm had begun infecting computers all over the world. This isn't news. Furthermore, this worm was so complex, that experts said that this had to be the work of a nation state, or

at least sponsored by the same. Even this was not news, as Stuxnet was discovered more than a year prior. What was interesting was that this malware was highly modular, and really did have the ability to record (and presumable stream) audio and hijack web cams. It could download any updates and modular programs it needed for any data sifting and analysis needed. It could even erase itself so that there would be no trace of it to be found by even a seasoned data forensics professional.

How did this malware spread? Well, in addition to several exploits (including a zero-day), it was abusing a bad implementation of a digital signature scheme. The signature scheme used the outdated and broken MD5 hashing algorithm which has been susceptible to collision attacks for several years before this point. The program used these collisions to produce a set of 'updates' from Microsoft that would appear to be legit (more on collision attacks later). Needless to say, Microsoft moved quickly to close this hole one it was discovered, but not before an undetermined number of computers in the Middle East and east Europe were infected. Hopefully this true story helps to explain why signature schemes are important to data security.

How Do Signature Schemes Work?

To help understand how we authenticate in the digital world, we might wish to understand how we do it in meat space.

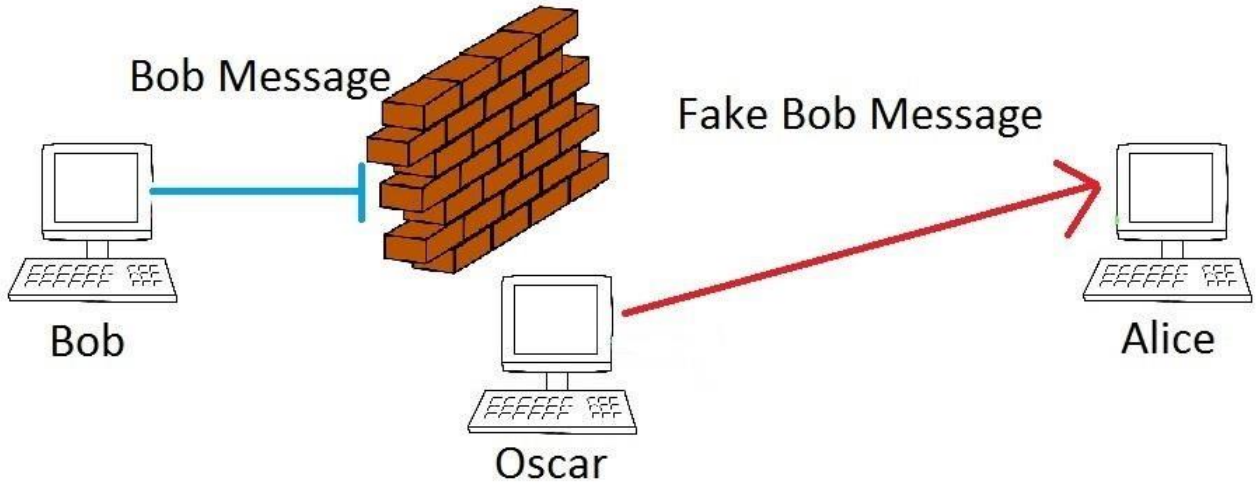
Image via jinshuchaoshi.com

When we wish a legal document to carry weight in the personally identifiable sense, we require signatures. When something is signed, it becomes harder to claim the document was not viewed and agreed to by you. Because we attribute special significance to a signature, we of course try to make them difficult to reproduce. An entire field (document analysis) exists solely in order to detect faked signatures.

The reasons to forge a signature are as varied as the uses signatures have. A criminal may wish to impersonate another person in order to withdraw money from someone's bank account, or to sell a house that does not belong to him, or even to make end-of-life decisions on their behalf. Worries like these are what authentication strength is about. However there are more things a skilled forger can do. For example, a former could intentionally tamper with one of his own signatures in order to invalidate a contract. This is the problem of non-repudation (inability to claim you didn't sign something). Clearly a good signature will be strong against these two problems.

Digital signatures, like in most applied cryptographic systems, relies upon mathematics. This can be related to the factorization problem (RSA) or Elliptic Curve Cryptography (ElGamal, DSA). In both cases, we are looking at trap-door public key algorithms. The trap-door part refers to the fact that they rely upon problems that are difficult to solve without special information (a trap-door), and the public key part refers to the key distribution method.

Ultimately, all signatures work the same. There are two functions that will either sign data, or will verify a signature. The function that verifies a signature is public (usually published) and the function that does the actual signing is kept private. In this way, anyone can theoretically verify a signature in order to determine the authenticity of a chunk of data, and only the user has the ability to sign data. Further, if Oscar were to try to alter the data in transit, the data will fail the signature check (ideally). This is in contrast to a physical signature which has none of these properties.



Obviously, our biggest worry is the difficulty involved with Oscar making a fake signature that will pass the verification algorithm. If Oscar can do this, then Oscar will have created a forgery. As I mentioned before, it is hard to do this in the same way that Bob would when sending Alice a message. However, signature schemes are not secure by themselves.

An example of an attack that can be done on RSA is a multiplicative attack. This attack can create a chosen message forgery by taking advantage of the multiplicative property of RSA. Put as simply as I can, for any combination of signature verification pairs like Sig(x1) = y1, Ver(x1, y1) = True & Sig(x2) = y2, Ver(x2, y2) = True, it will also be the case that Sig(x1x2) = y1y2, Ver(x1x2, y1y2) = True. Or ultimately, if Oscar takes any two pieces of data, and their corresponding signatures, their respective products will also evaluate true. It is much easier for Oscar to generate a pair of messages (x1, x2) that will equal the desired forged message. This is one of the many reasons why a signature system requires the usage of a hashing system in addition of a signature scheme.

To illustrate, y1 is not equal to Sig(x1). Instead y1 = Sig(h(x1)), where h(x1) is the hash of the message x1. So going back to our previous problem. Oscar has to generate two messages such that their products will equal the intended message. However, in a real system, it is not their messages products that must be equal, but their hashes products that must be equal. So Oscar must find an x1 and x2 such that h(x) = h(x1)h(x2). Clearly, if Oscar could do this, then he would also have to be able to compute a x1 or x2 for a given hash (and do so easily). In other words, the strength of the scheme is equal to the strength of the hash in relation to this attack.

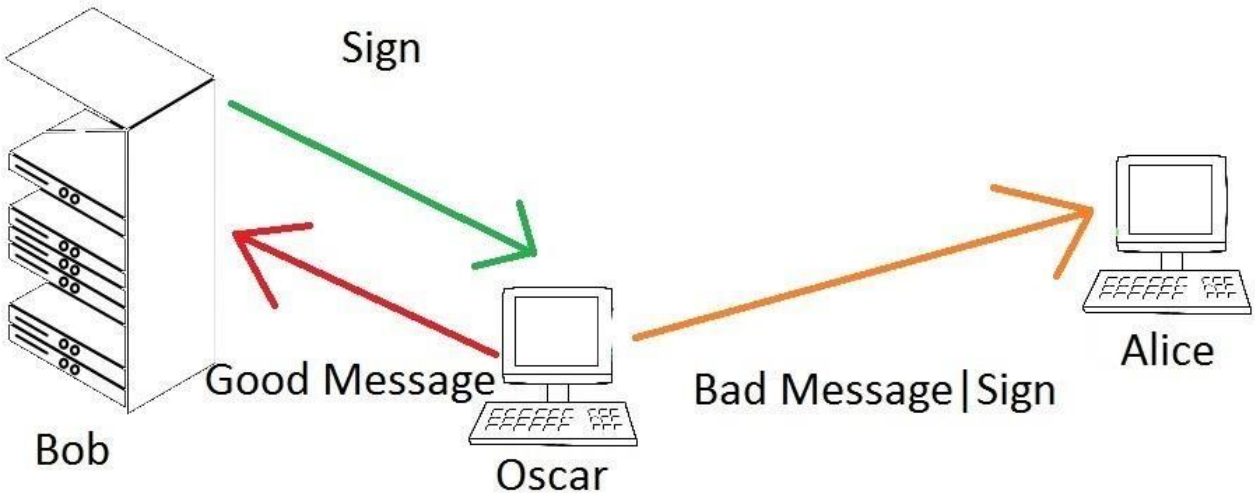
Hashes and FIPS

As mentioned earlier, Oscar's goal is to generate a valid signature for his message so that it will appear to be from Bob. As the signature is based upon the hash and not the data directly, Oscar needs only to create a hash that is equal to the target hash. Such a hash is said to "collide" with the other hash. Hence the term *hash collision*. If Oscar finds a message value x1 and x2 such that x2 != x1 but h(x2) = h(x1), then it is clear by substitution that Ver(h(x2), y2) = True. Finding a hash collision is not as easy as it sounds though. Assuming he does not have a shortcut, Oscar would need to make 2^n attempts, where n is the number of bits in the hash in order to find a particular hash. For SHA-1, this is 2^160 which is around a 1 with 46 zeros behind it. Even with the best computers in the world, Oscar does not have the time. This is where the Birthday Paradox comes in.

Image via co.uk

The Birthday Paradox is an interesting math problem that has a counter-intuitive result. Suppose you are throwing a birthday party for yourself and you invite a bunch of friends. Suppose you stop and wonder how many people you need to invite in order to have a 50% chance that someone else has your birthday. If we assume random chance, the number that must be invited is 253 people (1-(355/356)^253 = .51). That is a lot of people. Imagine we only cared about whether or not anyone shared a birthday with anyone else. Then the number is 23. This is less than a tenth of the previous number.

The same is true of hashes. The number of hashes involved in finding one hash that is equal to another is large. Extremely large. However if one were to manipulate not just the message you are trying to match, but also the original, then the number of iterations drops dramatically. The number of tries required to find a hash collision in this case is 2^n/2 or 2^80 for SHA-1. How would Oscar go about this? Well, he can't do this in the previously illustrated scenario. To do that, he would have to convince Bob to sign a message with his own signature system so that Oscar could impersonate Bob. Assuming Bob is not an idiot, this won't happen. So let's look at a new scenario.



Bob is now a server. All he does is check data to see if it matches certain criteria (e.x. is not malware). If it matches his criteria, he will sign the data and send it back to Oscar. Now Oscar can send his message to Alice. How does Alice know Oscar's message isn't a virus? Because Bob says so! In this case, Oscar can use the birthday attack to forge two messages. One will be an innocuous file such as a kitten, the other will be a trojan of some sort. Since Alice will not open the file from Oscar unless it is signed by Bob, Oscar needs to make both messages such that their hashes are equal.

- First Oscar creates both messages and appends a counter or random number to both.
- Then Oscar calculates both hashes
- Then Oscar needs to check if they match each other
- Assuming they don't, Oscar now changes both hashes.
- Now Oscar can check both the new hashes and the old to check for matches.
- Assuming they fail again, Oscar repeats the process

- He keeps doing this until he finds a match (or an error occurs)

Following this method, Oscar will eventually find two messages that have equal hashes. Now when Bob signs the innocuous hash, Oscar can use that signature for either message and Alice will accept.

If you wish to learn about the specific implementation of this security, I would suggest reading the FIPS specifications.

- **SHA-1 FIPS 180-4**
- **DSA FIPS 186-4**

Out of the Pan and into the Flame

Now that we understand what a signature is and what a hash collision is, we can understand what Flame really did. Just as Oscar had two messages, Flame also used two messages. In this case though, Flame took advantage of a Chosen Prefix Attack. The Chosen Prefix Attack uses two prefixes p1 and p2 to which the messages m1 and m2 are appended (p1||m1, p2||m2). Unfortunately, Microsoft had one server that was still accepting MD5 signatures way back when. This meant that the creators of Flame could leverage a cryptographic flaw in MD5 that allowed the calculation of hash collisions in only 2^50 iterations (less than 2^64 for chance). Flame then used this to trick Microsoft systems into thinking that it was a legitimate update. Using our new lingo, the systems were fooled using a forged signature created from a hash collision, caused by a Chosen Prefix Attack