Aaron Toponce

{ 2016.02.16 }

Checksums, Digital Signatures, and Message Authentication Codes, OH MY!

I recently submitted a bug to the Vim project about its Blowfish encryption not using authentication. Bram Moolenaar, the lead developer of Vim, responded about using checksums and digital signatures. I hope he doesn't mind me using him as an example here, but I want to quote the relevant bits (emphasis mine):

> The encryption is meant to avoid other people, who don't have the key, from reading the text. It does not have the goal of protecting manipulation of the text, that is something else. **You could add a checksum** even when not using encryption. **I believe it's called signing**.

Unfortunately, Bram is confusing checksums, digital signatures, and message authentication codes, all rolled up into one. I don't blame him. This is a topic that is not well understood by those not intimately familiar with cryptography. In a nutshell, each provide data integrity at the core. Where they differ is whether or not you're using encryption keys, and whether or not those encryption keys are symmetric or asymmetric. So, in this post, I would like to break it down.

Checksums

Checksums do not require any sort of encryption key. They are simply digests, or "fingerprints" that represent some data. When you download a piece of software from the Internet, there may be a file with an MD5, SHA-1, or SHA-256 hash of the file. This is the software vendor providing a way for you to verify that you got all the correct bits when the download completes.

For example, suppose you wish to download the latest Debian 8.3.0 amd64 ISO from https://mirrors.xmission.com/debian-cd/8.3.0/amd64/iso-cd/. Notice that there are the following files: MD5SUMS, SHA1SUMS, SHA256SUMS, & SHA512SUMS. Part of the SHA256SUMS file looks like this:

```
$ head SHA256SUMS
1dae8556e57bb04bf380b2dbf64f3e6c61f9c28cbb6518aabae95a003c89739a  debian-8.3.0-amd64-CD-1.iso
89facfbb5039e49d4e3eeff1cca6ab55e9121ff46affeb46ed510c11731acf41  debian-8.3.0-amd64-CD-10.iso
7f6bc807d3636975374b937c2724353f7468ecd7a61e60f2a8b71f92eeefe629  debian-8.3.0-amd64-CD-11.iso
bd99b7c274ea400b50960ab9e46dd23bad76f87574d2ceee1e8e43859fbd045b  debian-8.3.0-amd64-CD-12.iso
e85679304a509593526cffa77ff0d675329565eb4430444ee2c0d2cdd87842a8  debian-8.3.0-amd64-CD-13.iso
69f727bceb0460957bbd5023fe79749c6bf9f0e3a1b89945e6c63c6b3f04f509  debian-8.3.0-amd64-CD-14.iso
d1dab389f8cb794013986d2da8a6dc72c0be8bc932fcc6d7291cb09b418724d5  debian-8.3.0-amd64-CD-15.iso
913b5d89322b500a02f699d44778901cb59aae909f09bff64963115143c2a6ca  debian-8.3.0-amd64-CD-16.iso
0638aca6f59a8f5bec6d1cd4d272cea01758c2b2d6ec1412048ecb78ef684a77  debian-8.3.0-amd64-CD-17.iso
6f17742fbc82828f04da39f66647e958b0ac667cb4d2a40c9888c749680f1eb8  debian-8.3.0-amd64-CD-18.iso
```

So, when downloading "debian-8.3.0-amd64-CD-1.iso", I can use the sha256sum(1) command to verify the file:

```
$ sha256sum debian-8.3.0-amd64-CD-1.iso
1dae8556e57bb04bf380b2dbf64f3e6c61f9c28cbb6518aabae95a003c89739a  debian-8.3.0-amd64-CD-1.iso
```

The digest matches, so the download was successful and all the correct bits exist. Another way would be to download the SHA256SUMS file, and use the "-c" switch for the utility to verify the checksum automatically, rather than you eyeballing it:

```
$ sha256sum -c SHA256SUMS
debian-8.3.0-amd64-CD-1.iso: OK
```

The important thing to understand about checksums, is they are completely and totally anonymous. There is no secret shared between the server where I downloaded the software and myself, and there is no identity attached to the checksum. This means that anyone can change the original file and recalculate the checksum. So, if transferring data over the Internet, there is nothing preventing a man-in-the-middle attack from replacing the bits you're downloading with something else, while also replacing the checksum.

In other words, checksums provide data integrity, but they do not offer any sort of authentication. However, there are a number of checksum hashing functions, both cryptographically secure and not, such as CRC, MurmurHash, MD5, SHA-1, SHA-2, SHA-3, and so forth. For non-authenticated data integrity, a cryptographically secure hash function isn't always desirable, which is why non-cryptographic hash functions exist.

Digital Signatures

Digital signatures are a form of checksum, in that they provide data integrity, but they require asymmetric encryption to also provide authenticity. Digital signatures are away to attach an identity to the checksum. This implies a level of trust between you and the 3rd party, such as Debian with our example above. If you have met with the 3rd party, or dealt with them enough to establish some level of trust, then you can install the 3rd party's public key into your system. Then, when they provide data attached with a digital signature, you can verify that the data did in fact come from the 3rd party, and no other source.

Notice that a man-in-the-middle attack is no longer valid here, if I already have the 3rd party's public key installed on my system. Going back to our example with Debian, I have the Debian signing public key already installed. So, I can now download the MD5SUMS.sign, SHA1SUMS.sign, SHA256SUMS.sign, or SHA512SUMS.sign file, along with the checksums file I already downloaded, and verify that the checksums are those intended by Debian:

```
$ gpg --verify SHA256SUMS.sign
gpg: assuming signed data in `SHA256SUMS'
gpg: Signature made Sun 24 Jan 2016 11:08:33 AM MST using RSA key ID 6294BE9B
gpg: Good signature from "Debian CD signing key <debian-cd@lists.debian.org>"
```

```
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: DF9B 9C49 EAA9 2984 3258  9D76 DA87 E80D 6294 BE9B
```

If we look at the contents of the SHA256SUMS.sign file, we get the following:

```
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQIcBAABCAAGBQJWpRMhAAoJENqH6A1ilL6bYz4P/3ZNCR8N+rrlSSgTN/AkpSVt
WXWg2BTflY3cPYmKK/osJUvLT7HTPDhabPiuQY2jJxrHYJhq5sCOrhbgc4eSRmIf
IsSm7OxQ9TXqde4mg9DVsxmIRui/rVbhEjkAVu47A0eGDUrRxczgJUo14En3jO0Z
qhXypCIN90y8HWaqy6OMe+eCsPyGxmXpWRT1XEH9tOX21wCAaxUl6ZHkiqNqdt8u
Erojls77nlaBR/tvB9CHXTkUmqsocdYD+n5UsvtmLlYN0nz85b7NhrLEW2QtugLd
MJngeI5eJvI4Hjyas0HfSlsdoBAvF+Uw3Dn9aHiTIeWVIeCYUKhdXmLww0dL0n95
jVuBSuMavQwOKKRGTbvG++RET9s/2U/G95wK0Vfx5fsf1neKVJgYf9q9iyObgcH8
dRLAqkgWJBNkvm9oXmpcy7jAq8jlXzDfaPz8plAyqDuIXoOSCHpJ5KAbAS1cYLIT
9U2cQLKTbCPrWJT5xZzOMuCPWu1CzfluDEafFsNzurWG5vCmFEJ+vV9strkeEIuX
tFeKVDkkhVEZYQKSbIlidXBa/WP2Q0g1KvlKXb+nsnWDtWAjLUPD621F3ZjUcjlX
aDPv3J+7kqfryA/7qYMVTH67KY3DwKIDKt6XtquxSf7HuYqEwXKIXp2De7zCCEqH
csWVPFNUQyOdetIC/l/w
=TjXr
-----END PGP SIGNATURE-----
```

The details of a PGP signature aren't important for this blog post. Suffice it to say that it requires the signer's private key to produce the signature, and the signer's public key to verify it. The sender will hash the message with their private key, and append the signature to the message. When the recipient receives the message, they will separate the signature from the message, hash the message, and verify that the two signatures match using the sender's public key. If they do match, the recipient knows that only the sender signed the data, and no one else.

Because the private key is required to create the signature, and because only the 3rd party should have access to the private key, this means that a man-in-the-middle attack is no longer effective. A miscreant should not be able remove the signature, apply a new signature, and have the recipient still verify the signature as "good" from Debian, unless that miscreant also had access to Debian's private signing key.

To make an example of an existing software vendor, Arch Linux was under heat about this. A core developer strongly disagreed about digitally signed packages. They would provide their software from their repositories with MD5 checksums only. The packages were not digitally signed.

So, when your local Arch Linux installation would request packages from the Arch Linux software repository, unless served over HTTPS, a man-in-the-middle could interject their own bits with their own MD5 checksum. Your pacman(8) package manager would verify that the MD5 is valid, and proceed to install the software with root privileges, because that is what you told it to do. By also digitally signing the package with an Arch Linux signing key, this attack is no longer possible.

Eventually, Arch Linux fixed the vulnerability, and closed a very large security hole, by digitally signing their packages.

As with checksums, digital signatures really should be using a cryptographically secure hashing function as part of the protocol. This can include RIPEMD160, SHA-2, SHA-3, BLAKE2, Skein, and others. MD5 and SHA-1 are no longer considered cryptographically secure, and should not be used with digital signatures (thus why SHA-256 SSL certificates instead of SHA-1).

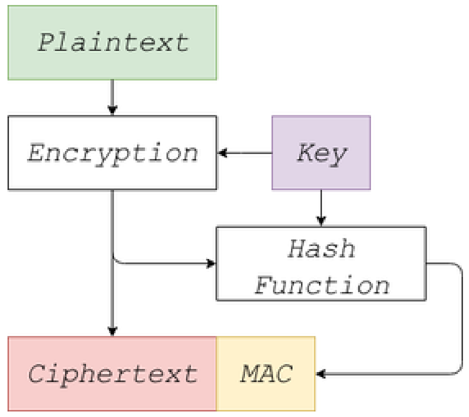
Message Authentication Codes

Finally, message authentication codes (also called "MAC tags") are another way to provide data integrity with authentication, but this time using symmetric encryption. Where digital signatures imply a physical identity behind the authentication, MAC tags provide anonymous authentication. Generally, symmetric keys don't have identities associated with them. They're usually short-lived and shared via complex key exchanges, such as the Diffie-Hellman key exchange.

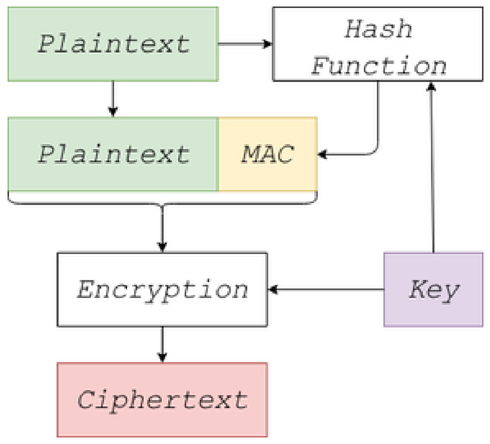
A MAC tag is keyed, meaning a shared secret is used when calculating the digest. There are a number of different implementations of MACs, such as CBC-MAC, HMAC, UMAC, & Poly1305, among others. The differences between each of those isn't important for this post. What is important, is how they are calculated, and how they are used with encryption.

The sender of some message will apply a cryptographic hashing function to the message, and append (or prepend) the resulting digest to the message, and send the full payload off. Because both the ciphertext and the MAC were calculated with a shared secret key, a man-in-the-middle cannot strip the MAC tag and apply their own without knowing the shared secret. Because, when the recipient receives the payload, they will strip off the MAC tag, rehash the message with the same keyed hashing function, and see if the two MAC tags match. If they do match, the message can be acted upon. If they do not match, something happened to the data in transit, and the payload can be safely ignored.

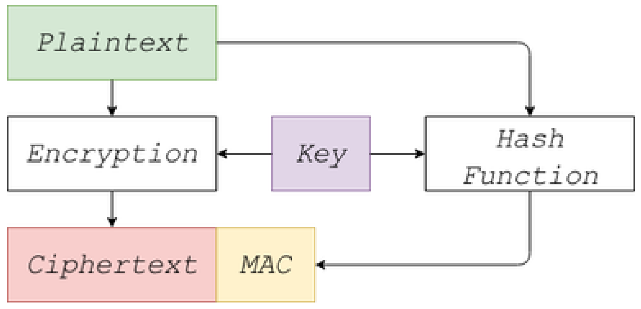
There are three main ways to apply MAC tags to messages: encrypt-then-MAC, MAC-then-encrypt, and encrypt-and-MAC. The first, encrypt-then-MAC, is considered "best practice" for message authentication. First the message is encrypted, then the ciphertext is authenticated, and the resulting MAC is appended to the ciphertext. This provides both ciphertext and plaintext integrity. The big advantage to this approach, is that if the MAC tag does not match the newly calculated MAC tag during verification, the ciphertext does not need to be decrypted. This is the default approach with IPsec and modern versions of OpenSSH. RFC 7366 standardizes this for TLS (yet to be implemented by OpenSSL last I checked). Also an ISO/IEC 19772:2009 standard.



The next approach, MAC-then-encrypt, means authenticating the plaintext, appending the resulting MAC tag to the plaintext, and then encrypting the full plaintext and MAC tag payload. While this approach offers plaintext data integrity, it does not offer ciphertext integrity. As such, the ciphertext must be decrypted before the MAC tag can be verified. This is the default behavior in older versions of OpenSSH.



Finally, encrypt-and-MAC, means authenticating the plaintext first, then encrypting the plaintext. The resulting MAC tag is appended to the ciphertext. Again, like MAC-then-encrypt, this approach offers plaintext data integrity, but it does not offer ciphertext integrity. So, you must detach the MAC tag first, then decrypt the ciphertext, then verify if the MAC tag is valid. This is the default behavior with OpenSSL.



As I understand it, there are no known vulnerabilities with MAC-then-encrypt and encrypt-and-MAC MACs. However, by having both ciphertext and plaintext integrity with encrypt-then-MAC, as well as not needed to decrypt the ciphertext on failure, is why encrypt-then-MAC is the preferred way to handle message authentication.

As with digital signatures, MACs should be calculated with cryptographically secure hashing functions, such as RIPEMD160, SHA-2, SHA-3, BLAKE2, Skein, etc. MD5 and SHA-1 would not qualify (although we could get into a discussion about HMAC-MD5 and HMAC-SHA1, but we won't).

Conclusion

No doubt, it's confusing to separate checksums from digital signatures from message authentication codes. Things get even a bit more hairy with blind signatures (used primarily in digital currencies) and Merkle trees (used primarily in peer-to-peer networks and copy-on-write filesystems), but they're special cases of the primary three functions discussed above. However, if you can get checksums, digital signatures, and message authentication codes cleared up, then you're that much closer to implementing cryptographic protocols correctly.

Posted by Aaron Toponce on Tuesday, February 16, 2016, at 10:01 pm. Filed under Cryptology, Security. Follow any responses to this post with its comments RSS feed. You can post a comment or