Dubious security vulnerability: Discovering the salt

devblogs.microsoft.com/oldnewthing/20161214-00

December 14, 2016

Raymond Chen

A security vulnerability report came in that went something like this:

> The XYZ component has a feature where it can cache the password used to access a network resource. It encrypts this password with the `CryptProtectData` function, using the default options so that only the user who encrypted the data can decrypt it. It then saves this encrypted password in a file that is accessible only to the user.
>
> The security vulnerability is that when the component encrypts the data, it uses a predictable entropy, namely the name of the network resource. This means that any program running as the user can open the file that has the encrypted password, pass the name of the network resource as the entropy, and then call the `CryptUnprotectData` function to decrypt the password.
>
> The XYZ component needs to use an entropy that cannot be predicted. Otherwise, any program running as the user can extract the password.

When you protect data with the `CryptProtectData` function, you can optionally pass some additional data that will be used as part of the encryption key. The `CryptProtectData` function calls this extra data *entropy*, but a more common name for it is *salt*.

It is not important that the salt be unpredictable. What's important is that the salt is different for each network resource.

The salt serves multiple purposes.

One use for the salt is to prevent someone from taking an encrypted password from one network resource and applying it to another network resource. The idea here is that somebody might get their hands on the file, and even though they cannot decrypt the password, they can copy the encrypted password from the entry for one network resource and paste it into the entry for another network resource, trying to take advantage of the fact that people often reuse passwords.[1] Alternatively, the attacker can expend effort to decrypt one password, and once they do, they can look for the same encrypted data elsewhere and know that they have the password for that other resource as well.

Salting the encryption with the name of the network resource foils this trick. Since each password is encrypted with a different salt, if you take an encrypted password from one network resource and try to decrypt it for another network resource, the decryption will fail because the salt (and therefore the decryption key) does not match.

Another purpose for the salt is to make it more computationally expensive to precalculate passwords. A common way of attacking a stolen password database is to take a dictionary of possible passwords, and encrypt each individual password. You then invert the table so that the encrypted password is the lookup key and the decrypted password is the value.

Building this giant table takes a long time, but once you have it, you can attack a large number of passwords at once because you can just take every stolen encrypted password and look it up in the table. If it's in the table, then you have the decrypted password almost immediately.

The salt foils this attack: Since each password is encrypted with a different salt, somebody who wants to mount a reverse-lookup attack would have to have multiple reverse lookup tables, one for each salt. And if you give each password a different salt, then the reverse-lookup attack completely dissolves, because your reverse-lookup table is good for attacking only one password.

Another clue that the salt is not sensitive information is the fact that on classic Unix systems, the master password database `/etc/passwd` is readable by everyone. It encrypts each password with a different salt, and the salt is *readable by anyone*.

So if you're concerned that some Windows component doesn't do a good job of protecting its salt, then you should be even more concerned that classic Unix systems don't even try to protect the salt at all!

Bonus chatter: This report is even more bogus even if you get past the fact that the salt is easily guessed. Even if the salt were hard to guess, you can still figure out what it is because you can reverse-engineer the code to the XYZ component and see how it calculates the salt. And then you can replicate this calculation in your own rogue program.

The algorithm for generating the salt needs to be deterministic, because you have to be able to generate the same salt that was used to encrypt the data in order to decrypt it. And that algorithm must operate on publically-known data, because it is generating the inputs to the decryption function! If the salt were itself encrypted, then you've begged the question: How do you decrypt the salt? Is it salt all the way down?

Bonus bonus chatter: An attacker wouldn't even need to extract the salt in order to get the password. The weak salt is a complete red herring. Since the XYZ component runs in-process, the decrypted password is already in memory in the same process. So the attacker just needs to load the XYZ component and ask it to connect to the network resource using the cached password. The XYZ component will decrypt the password, and now the decrypted password is in the attacker's process. It just needs to go copy it.

Now, the attacker could spend a lot of effort studying the XYZ component to see where it stores the decrypted password in memory. Or it can just detour the `CryptUnprotectData` function! The XYZ component will call `CryptUnprotectData`, and that will call the detour. The detour passes the call through to the original function, and then inspects the result. Bingo, instant password.

[1] Or create a new entry for a network resource that the attacker controls, and copy the encrypted password to it. If you can convince the user to connect to your rogue server, then you will receive the decrypted password.

Raymond Chen

Follow