Windows NT services are assigned a SID based on an SHA-1 hash, but what about the risk of collision?

devblogs.microsoft.com/oldnewthing/20190717-00

July 17, 2019

Raymond Chen

Windows NT services are assigned an identity (SID) based on an SHA-1 hash. We also know that SHA-1 is deprecated due to research showing that it is vulnerable to collision attacks from well-funded opponents. What does this mean for Windows NT services?

For the purpose of this discussion, we'll put Windows NT services into two categories. The first is the traditional Windows NT service that runs as a single instance for the entire system. The SHA-1 hash for these services is generated from the name of the service.

The other category of Windows NT service for this discussion is the per-user service. The SHA-1 hash for these services is generated from the name of the service, combined with a locally-unique 64-bit identifier (LUID) that was generated and assigned to the logon session.

Both categories of service require an administrator to install. The difference is that there is only one instance of a system service, but each user gets a separate instance of a per-user service.

To get a collision between two system-wide services, both services need to be installed. The name of the service is determine when it is installed, and you need to be an administrator to install a new service, so installing a service with a crafted name that triggers a SHA-1 collision requires you already to be on the other side of the airtight hatchway and is therefore not particularly interesting.

Suppose you could force a collision between a per-user service and a system-wide service. There may be resources which grant access only to that specific well-known service. But since your per-user service has the same SID, it also gets access to those resources. The per-user service runs as the logged-on user, so the logged-on user could take over the per-user service and inject code that takes advantage of the SID in the token to access those special resources.

Similarly, a collision between two per-user services means that one user can access resources that would normally be assigned to another. This may be interesting if the victim is an administrator, but the administrator isn't going to be trying to force collisions. The administrator’s per-user services pose static targets, the same as per-system services.

If you control two users, then you can try to create a collision between one user’s per-user service and the other user’s per-user service. But in this case, you’re already on the other side of the airtight hatchway, because you control both accounts already.[1]

Therefore, when we’re looking at the odds of collisions, we are looking at a targeted attack from a per-user service to a system-wide service or to a per-user service for a specific targeted victim. We’re not looking at a birthday attack of per-user services instances. This means that you don’t get the benefit of a roughly quadratic reduction.[2]

Okay, so how likely is such a collision?

Let’s say that there are S system-wide services, U per-user services, and V potential signed-in victims. You the attacker can try 2^{64} versions of each U in the hopes that one of the versions will match one of the S‘s, or maybe one of the $U \times V$ active victim services.

Note that this is significantly more constrained than the general SHA-1 hash collision attack. In the general case, you have much more freedom in trying to find source data that results in the same hash. In our case, however, you have access to only 64 bits of data. Therefore, the problem is not really related to the mathematical soundness of the hash, because the constraint is not in flaws in the hash function that permit manipulation of the hash by feeding it carefully crafted data. In this case, you don’t get to feed it carefully crafted data. The data you control is only 64 bits of the total input to the hash, so it’s a much more mundane issue of how many hashes you have access to in the first place.

Let’s say that there are at most 1000 system services, at most 1000 per-user services, and at most 1000 signed-in victims, resulting in 1,001,000 possible service targets. This seems rather generous, seeing as it means that we have over a million services running on a single system.

The attacker has U services, each with 2^{64} possible LUIDs, so the attacker can reach $2^{64} \times U$ possible SHA-1 hash values. Let’s assume that the attacker has access to infinitely powerful computing resources and can calculate hashes in zero time.

The SHA-1 hash space is 2^{160}, which is even larger than the space for GUIDs, and we saw earlier that the GUID space is incomprehensible large.

This means that the fraction of the SHA-1 space that the attacker can reach is

$2^{64} \times U \div 2^{160} = U \div 2^{96}$

The attacker has $S + U \times V$ potential victims, so the odds of a hit are

$(S + U \times V) \times U \div 2^{96} \leq (1000 + 1000 \times 1000) \times 1000 \div 2^{96} \approx 10^{-20}$

The odds of winning the Powerball lottery with a single ticket is around one in 300,000,000 or about 3×10^{-9}.

You are more likely to win the Powerball lottery two times in a row than you are to have even a potentially-exploitable collision in the service name space.

Looking at it another way: Let's take a super-high estimate of one trillion Windows computers. That's 10^{12} computers. And every single one of them has a thousand users running a thousand services each. The odds that a potentially-exploitable collision exists at all is one in 10^8, or one in 10 million.

But let's say you're incredibly lucky and not only did that ten-million-to-one shot actually come through, but you were able to find that one potentially-exploitable computer hiding somewhere in the world, and you by another miracle have logon privileges to that computer.

How hard will it be for an attacker to turn the potential exploit into an actual one?

The problem here is that the attacker doesn't directly control the LUID. Rather, each time they log on, the system generates a new LUID and assigns it to the logon session. If the LUID they get isn't the one the attacker wants, they'll just have to log off and back on again to try again with a new LUID.

The rate limiting factor is therefore not within the attacker's control. The attacker could have all the computing resources in the world available to try to identify potentially-exploitable systems, but the only computing resources that are significant for generating the actual collision are those on the system under attack, since that is the system that is generating the LUIDs.

To get a new LUID, the attacker needs to log off and back on. That logoff/logon cycle is going to slow down the attack considerably. For one thing, it's a logoff/logon cycle, which is not exactly the fastest operation. For another thing (and more important), there's no way to parallelize it or do the work offline. The LUIDs are generated by the system under attack; the attacker doesn't get to pick them. The attacker just has to keep spinning the LUID roulette wheel and hope for a winner.

Let's say that the attacker has somehow managed to find a way to log off and back on a rate of one million per second. (Even just getting that far is already super-valuable and they could probably sell "microsecond logon" to corporations for a ton of money.)

At a rate of a million LUIDs per second, the expected time to get the LUID they want is around 300,000 years. So they'll have to hope that the system doesn't reboot for over 3,000 centuries, or they'll have to start over.

What did we end up with? Well, in order to have a system that is even potentially-exploitable, you need to win the Powerball lottery twice in a row, and given a generous upper limit on the number of Windows systems in the world, the odds of such a potentially-exploitable system even existing at all is one in ten million. If you manage to find such a system, you'll need to convert that potential to reality, and you'll need to keep logging on and off the target system for about 3,000 centuries on average.

Good luck with that.

[1] I'm assuming that there is only one attacker at a time. If there are multiple attackers, then maybe one of them can collide with a SHA-1 created by another. No attacker gains administrator privileges, but one attacker can assume the identity of another attacker. Which seems like poetic justice.

[2] But say we want to consider collisions between user-services. The probability of a birthday collision is approximately

$1 - (1 - 1/d)^{n(n-1)/2}$

where $d = 2^{160}$ is the size of the search space and $n = U \times 2^{64}$ is the number of guesses.

The exponent simplifies to

$n(n-1)/2 \le n^2 / 2 \le U^2 \times 2^{128} / 2 \le 2^{20} \times 2^{127} = 2^{147}$

Let's plug this into the overall exponentiation operation.

$(1 - 1/d)^{2^{127}} = (1 - 2^{-160})^{2^{127}} \approx 1 - 2^{127} \times 2^{-160} = 1 - 2^{-33}$

Therefore, the total probability is

$1 - (1 - 2^{-33}) = 2^{-33}$

That's one in eight billion. The odds of winning the Powerball lottery is one in 300 million, so this is still twenty times harder than winning the Powerball lottery.

But wait, now that you've found a potential collision, you still have to realize it. And instead of forcing just one LUID, you now have to force two, so you're going to need around 6,000 centuries to get that collision.

Your odds of finding a potential collision are better (one fewer Powerball lottery), but you'll have to wait a lot longer to redeem the winning ticket.

Raymond Chen

Follow