Persistence with Windows Services

gtworek.github.io/PSBits/services.html

When it comes to hacking, Windows Services are priceless due to couple of factors:

- They natively work over the network-the entire Services API was designed with remote servers in mind,
- They start automatically when the system boots up,
- They may have extremely high privileges in the OS,
- They run undercover and the GUI reveals only small part of their real configuration.

Of course, you must have high privileges to manage services, and an unprivileged user rarely can do anything more than looking at the configuration. But when it comes to make your attack persistent-it is still one of the better choices and this did not change for over 20 years.

On the bit more technical side, the Windows Service is a special (because it wants to communicate through the Services API) process, launched and maintained by Service Manager- `services.exe` process, launched quite early at the OS boot. Services Manager reads the Services configuration from the database stored in the registry (in `HKLM\ System\ CurrentControlSet\ Services\`) launches his child processes according to the rules defined, and communicates with them through `ControlService()` calls, using codes numbered from 0 to 255. Some of these codes are known as `Stop` , `Pause` , etc, others are less common, and anything above 128 is left for developers to be used freely.

And here comes the need for persistence. If an attacker is already in, if it has the power over the OS, the typical steps to be done are:

- Spread to other machines, also known as “lateral movement”,
- Make the presence permanent, ideally staying undercover and survive across reboots.

Even if the first goal may be quite often easy to achieve with Windows Services, we focus here on the later one.

In theory it is simple: if an attacker has high privileges, he can create a service doing whatever he needs, and ideally allowing him to take control back, any moment he needs. But admins are not that dumb and such service may be spotted minutes later, disabled, removed, analyzed, used to tighten defense procedures, trace the attacker etc.

Let’s then create a perfect configuration for persistence-oriented service for the Red, providing at the same time some tips how Blue can stay smarter. I can give you some spoiler: Blue wins usually, when it comes to the detection, but it is bitter-sweet win, as the system was already hacked and no one should trust it fully anymore. It is not that hard to imagine

that attack relied on other techniques too, and Blue will never know if he checked really everything. Format is the next logical step, if some highly-privileged Red actors were observed within the OS.

Let's go over technical details now:

1. **Service name**. This is usually the very first thing Blue sees. And Red wants to make his service looking legit. Red will not name it with randomly generated name and will not suggest it is something worth digging deeper. Good name can refer to some deeply-hidden OS mechanisms, suggesting the investigator "do not touch me and just pass-by". Something like *PnP Enumerator*, *Transport Layer Security Helper*, etc. The more misguiding Google results for such name, the better. Additionally, Red should remember that each service requires two names actually: the `Display Name` and the `Service Name`. Both are displayed in the Service Properties window in the `services.msc` console and in multiple other places. For Blue, the name itself is virtually useless. It may be misleading or just stolen from another legit service, Red has deleted or misconfigured. Sure, it's worth taking a look if Red is really stupid, but it is the only scenario. Blue can display all service names (locally and remotely) with PowerShell and WMI- `gwmi Win32_Service | select Name, Displayname`
2. **Service description**. This one is really interesting because descriptions are visible in GUI but quite hidden for scripting and applications. It results from the fact, that descriptions were added to Windows Services relatively late-in Windows 2000. Effectively, the API used to add new services does not need any description to create fully working service. Blue can do very simple check: open `services.msc` and sort the list by the `Description` field. Or run `gwmi win32_service | where-object {$_.description -eq $null}`. Services without descriptions immediately stick out and should be carefully watched first. Of course, some legit 3rd party services are installed such way as well, and you should blame their developers, and of course some malware provide descriptions too. But taking the first steps into an investigation, Blue definitely should pay attention at the description, and at the same time do not rely on it. And Red should fill it up. Nothing very elaborate, just a phrase or two, somewhat aligned to what the service name says. In most cases no one will read it with an intention of understanding.

3. **Service binaries**. In theory, Blue should know every single executable file in his environment. No excuses, especially as he has PowerShell and can do this in an automated way. And collecting such information must happen before any attack, otherwise, the reference data will be spoiled, and effectively useless. Red, in turn, can rely on fact that even if Blue collected all *.exe files, he probably left *.dll for later. Such approach is very common as DLLs are more unpredictable, more common, their number is higher, they change often, exist in multiple versions at the same time etc. While Blue knows well what is the purpose of most EXEs in the system, he cannot say the same about DLLs. And even if Blue collected DLL-related data, it is probably obsolete already. And here comes the Red superpower: services can be run from DLLs too. It is how Windows launches most of his own services since Windows 2000. Service Manger launches `svchost.exe` and the `svchost.exe` loads the service DLL and executes its code. Effectively, DLLs are as good as EXEs for services, but in addition to the lack of proper DLL monitoring, Red has two great advantages. The first advantage is a fact that `svchost.exe` is used only by Windows, not by 3rd party applications. It makes Blue think "*it is svchost, it means it is part of the OS, it means I will check elsewhere*". Quite what Red wants to make Blue think about his stuff. The second benefit results from the monitoring tools design: they focus mainly on processes and not on libraries loaded. And from the process perspective, Blue has the well-know, Microsoft-made, whitelisted, and commonly used `svchost.exe`, which exists in dozens of instances on every single Windows machine. The only hope for Blue, wanting to detect unwanted binaries, is to watch services DLLs as well. And it is the high time, so I am providing a ready-to-use script for listing service DLLs. And Red can enjoy the sample service DLL source code at GitHub as well.
4. **Service account and privileges**. Each service has its identity. If no one configures it, it inherits the security token from the parent (`services.exe`) running on `LocalSystem` account. Effectively most of services runs with the highest privileges you can observe in the OS. If Red wants to stay undetected, he can leave his service running such way. If Red wants to be tricky, he can also intentionally configure his service on less privileged identity such as LocalService, leaving privilege-based backdoor, to regain full power when needed. Some interesting cases are already described and ready to use but other privileges may be used as well, according to the table explaining how to use them to get power back. For Blue, it means not only the identity is important, but the set of privileges too. Especially if such less privileged account has some superpowers. On top of it, we should know that services can run on regular accounts (both local and domain ones) but it should not be used by Red as such configuration is relatively rare and sticks out at the first sight.

5. **Service activity**. From the Red perspective, the goal of having his service running is to keep a possibility of running his actions with high privileges. Usually, it is not needed all the time and rather on demand. In the meantime, the service should stay as quiet as possible. Unless necessary, Red should not open ports, should not connect over the network, should not try to open files or registry keys. Blue can easily spot such activities i.e. with Process Monitor from SysInternals, and start asking questions leading to the full disclosure. Of course at some point, Red needs to do this, but by making such events rare, he lowers the chance of being caught red handed. For Blue, two scenarios may exist: if something looks strange and the real-time monitoring (i.e. with ProcMon) answer questions, or if the extensive logging leaves historical traces for analysis. Anyway, the properly designed service-based persistence will be significantly harder to spot if it works fully asynchronous, acting only when intentionally asked and doing absolutely nothing otherwise. Both, Red and Blue, should also remember, that some misleading actions (such as periodical reading of PnP-related registry keys by the service masking his intentions under "*PnP Something*" name) may be taken. In such case it all depends on the Blue personality and knowledge. Will such behavior be more suspicious than total silence? It may be, or may be not.
6. **Service communication**. The Services API is perfectly designed for asynchronous communication with a service quietly waiting for orders. It may be tempting for the Red, to leave his service waiting for signal sent through Services API, but it would require service permissions manipulation. It's an important topic on its own and I am covering it at the end of this article. Some interesting ways may open if a service is asking for being notified by Windows about different conditions such as file or registry manipulation, low memory signals etc. If Red can restart the OS, or just wait - one of the smartest place to hide his malicious steps is the service startup routine. In most cases it happens before Blue logs on, making the observation a bit harder. General rule for Red should be "*wait patiently instead of actively asking*". It will make the investigation significantly harder. When the proper signal is detected, I would strongly recommend to read a file with instructions what to do next. It gives Red two very important benefits: first benefit is about nothing being hardcoded inside the service binary, which makes analysis impossible, the second benefit is the elasticity, as virtually anything may be put into such file without recompilation and replacing binaries. To keep it simple I could suggest to work with *.cmd files. They are easy and powerful and I am covering file-based actions in the next paragraph. For Blue it may be hard to guess how the service works unless fully reverse engineered. It's why I would recommend other ways of spotting bad services. Of course if you identify one, the reverse engineering should focus on communication as well, but it remains far outside of scope of this short article.

7. **Service strings and API calls**. When Blue spots the suspected service, he tries to look inside to understand how bad such service is. The best option for Red is to leave no traces. There are different obfuscation methods, but using them provides clear signal for Blue: something smells bad. So Red should be very open and just have nothing interesting inside. And a lot of things may be found by Blue, just with two actions: reading strings from the binary file and monitoring the service behavior within the OS. If we assume the Service tries to read his orders from "*orders.cmd*" file, Blue has both: the filename from strings and the filename from the API call (does not matter if successful or not) itself. If Red hides his steps, none of these is possible. Instead of opening "*orders.cmd*", the Service may list all files from multiple locations and if the right file name is not observed, there is no reason for trying to open it. Additionally, instead of checking same name each time, the Service may look for different (calculated) name each day. Red knows the name for today, places the properly named file in the right location and makes service pick it up. As string comparisons (required eventually to check if the right file is really present) are relatively easy to spot by Blue, the code should verify names differently, for example one character at a time. Another approach may rely on registry keys. To keep the example very simple: Service lists some keys, checks if "*__YYMMDD__something*" exists on the list just read, and if yes-launches the "*something.cmd*" from the disk. And again-for Blue it will require a lot of debugging and monitoring to understand what just happened.
8. **Service self-defense**. When Blue spots something suspicious, one of the very natural actions is to stop this and eventually remove. Of course, for Red, it does not sound very good, so it would be great to provide some protection. As the real protection is not really possible, Red should act in a way telling Blue "leave me alone and go further". My favorite method of protecting services rely on two mechanisms: allowed methods and critical processes. As each service reports to the Service Manager what is allowed, it is perfectly possible to report that "`Stop`" action is not allowed as well. Such configuration will make the service unstoppable, and the Service Manager will report "*1052: The requested control is not valid for this service*". Of course, Blue can kill the process responsible for the Service, but here the second protection may help. The process in Windows can call an undocumented `RtlSetProcessIsCritical()` function, telling the OS it is critical. If someone kills such process, the OS will invoke Blue Screen. I can promise you, Blue will try only once. For Blue, it means "*just check elsewhere*". Actually it should mean the opposite: critical processes exist, but the list of them is really short and a comparison against healthy computer should help. The general advice for Blue says also "*suspend instead of killing*". It is not a bad way for any malware, not only running as a service. Process Explorer can easily do this with right click.

9. **Service digital signatures**. As previous paragraphs could leave Blue a bit sad, now we have something bringing the hope back. Checking digital signatures is very easy (i.e. with PowerShell) and it immediately exposes all strange cases. If something is run by `svchost.exe` and it is not digitally signed by Microsoft, it is high time to dig deeper. Red can do only one thing here: generate self-signed certificate pretending to be Microsoft one, add it to the `Trusted Root Certificate Authorities` store, and use it for signing the malware. It is quite unlikely, but if Blue wants to be really sure, he should check binaries signatures on a healthy machine.

And that's it. Who wins? The smarter one. If both sides are equally smart, the Blue wins when it comes to the detection (file hashes and digital signatures are priceless) and with an attack itself, the Red wins as he already has a foot in the door and Blue should never trust such computer anymore.

The thing I have promised to cover is Services ACL. As it can be fully controlled with `sc sdset` command, Red can manipulate this as well. When such manipulation is an option, the entire scenario of establishing persistence may rely on simple manipulation, allowing regular user to re-configure some service. If it is possible, regaining privileges is easy: set service to run on `LocalSystem` and provide own binary. It can by achieved with one of three privileges for the service: `DC` -reconfigure service, `WD` -change permissions, to make adding DC possible, `WO` -take ownership, allowing to take full control as well. It's Blue job to list service permissions periodically and check if someone strange appeared on the list. I am providing a simple PowerShell script, greatly helping here.

This text first appeared on https://grzegorztworek.medium.com/