The Rise of Potatoes: Privilege Escalations in Windows Services

Antonio Cocomazzi
System Engineer, SentinelOne
→ System Engineer @ SentinelOne

→ Passionate about IT security and constantly trying to learn and experiment new cool stuff, especially on Windows Systems

→ CTF player and proud member of @DonkeysTeam
Why this talk

➔ Windows Service Accounts usually holds “impersonation privileges” which can be (easily) abused for privilege escalation once compromised

➔ “Rotten/JuicyPotato” exploits do not work anymore in latest Windows releases

➔ Any chance to get our potatoes alive and kicking, again?
Agenda

→ Basic Concepts:
  ◆ Windows Services
  ◆ Windows Service Accounts
  ◆ WSH (Windows Service Hardening)
  ◆ Impersonation

→ From Service to System
  ◆ RogueWinRm
  ◆ RoguePotato
  ◆ Juicy2
  ◆ Other non-”potatoes” techniques

→ Relaying potatoes authentication

→ Mitigations

→ Conclusion
What is a service?
- Particular process that runs in a separate Session and without user interaction.
- The classic Linux daemon, but for windows

Why so important?
- Most of the Windows core components are run through a service
- DCOM, RPC, SMB, IIS, MSSQL, etc...
- Being daemons they will be an exposed attack surface

Must be run with a Service Account User

Configurations are under HKLM\SYSTEM\CurrentControlSet\Services
Windows Services

→ How you recognize a service?
  ♦ Child process of services.exe (SCM)
  ♦ Process in Session 0
  ♦ From source code perspective: SvcInstall(), SvcMain(), SvcCtrlHandler(), SvcInit()…

→ How the NT Kernel recognize a service...
  ♦ S-1-5-6  Service
    A group that includes all security principals that have logged on as a service.
Windows Services Accounts

→ Windows Service Accounts have the password managed internally by the operating system

→ Service Account types:
  ♦ Local System
  ♦ Local Service / Network Service Accounts
  ♦ Managed Service & Virtual Accounts

→ Allowed to logon as a Service, logon type 5

→ Could be also a normal user who has been granted the right “Log on as a Service”
Windows Services Accounts
Windows Services Hardening (WSH)

→ Until Windows Server 2003/XP every service was run as `SYSTEM`

→ If you compromise a service you have compromised also the `whole` machine

→ WSH to the rescue, at least that was the initial goal

→ Great references by @tiraniddo [1] and @cesarcer [2]

Windows Services Hardening (WSH)

→ Limited Service Accounts
  ♦ Introduction of the LOCAL SERVICE and NETWORK SERVICE accounts, less privileges than SYSTEM account.

→ Reduced Privileges
  ♦ Services run only with specified privileges (least privilege)

→ Write-Restricted Token

→ Per-Service SID
  ♦ Service access token has dedicated and unique owner SID. No SID sharing across different services

→ Session 0 Isolation

→ System Integrity Level

→ UIPI (User interface privilege isolation)
Impersonation

→ “Impersonation is the ability of a thread to execute in a security context that is different from the context of the process that owns the thread.” MSDN

→ Basically it allows to execute code on behalf of another user

→ Token forged by impersonation are known as secondary token or impersonation token

→ Your process token must hold the **SeImpersonatePrivilege** (“Impersonate a Client After Authentication”) to perform the impersonation

→ It is the prerequisite for all the techniques will be shown
Impersonation

Impersonation assigns a token to a thread, replace the token used in access checks for the majority of system calls [1]

**Direct Setting**
- SetThreadToken()
- ImpersonateLoggedOnUser()
- NtSetInformationThread(…)

**Indirect Setting**
- ImpersonateNamedPipeClient()
- RpcImpersonateClient()
- CoImpersonateClient()

**Kernel Setting**
- PsImpersonateClient()
- SeImpersonateClient/Ex()

Impersonation

→ You are wondering now: what is the link between Services and the impersonation privileges?
From Service to SYSTEM
RogueWinRm

→ **Release Date:** 6 December 2019
→ **Authors:** @decoder_it - @splinter_code – 0xEA (@DonkeysTeam)

→ **Brief Description**
  ◆ Force the BITS service to authenticate to a Rogue WinRm HTTP server in a NTLM challenge/response authentication resulting in a SYSTEM token stealing. [1]

→ **Requirements**
  ◆ WinRm Port (5985) available for listening
  ◆ By default impact only Windows clients, no Windows Servers

[1] https://decoder.cloud/2019/12/06/we-thought-they-were-potatoes-but-they-were-beans/
RogueWinRm

→ When a BITS object get initialized a weird behavior happens
→ BITS object could be created through a DCOM activation using its CLSID or by a simple "bitsadmin /list"
RogueWinRm

→ RogueWinRm is a minimal **webserver** that performs NTLM authentication over HTTP

RoguePotato

→ **Release Date:** 11 May 2020  
→ **Authors:** @decoder_it - @splinter_code

→ **Brief Description**

- Tricks the DCOM activation service in contacting a remote Rogue Oxid Resolver to force RPCSS writing to a controlled named pipe getting a NETWORK SERVICE token. After that it uses Token Kidnapping to steal a SYSTEM token from the process space of RPCSS [1]

→ **Requirements**

- The machine can make an outbound connection on port 135  
- SMB Running  
- DCOM Running

RoguePotato: the attack flow 1/4

**Step 1**

Trigger Istorage  
(Account with Impersonation privs)
RoguePotato: the attack flow 1/4

→ Tricking the DCOM activation service [1]
  ♦ Pick a CLSID to create an object activation request
  ♦ Once the object is created, initializes it to a marshalled object. (IStorage)
  ♦ In the marshalled object (OBJREF_STANDARD) we specify the string binding for a remote oxid resolver. This will be the ip of our remote rogue oxid resolver
  ♦ When the COM object will unmarshal the object (CoGetInstanceFromIStorage) it will trigger an oxid resolution request to our rogue oxid resolver in order to locate the binding information of the object

[1] Credits to @tiraniddo --> https://bugs.chromium.org/p/project-zero/issues/detail?id=325
RoguePotato: the attack flow 2/4

Step 1

Trigger Istorage (Account with Impersonation privs)

Step 2

Fake ResolveOxid2 (Anonymous Logon)
RoguePotato: the attack flow 2/4

→ Forward the resolution coming to the remote host (port 135) back to the local host where the Rogue Oxid Resolver runs

→ Write the code of the malicious ResolveOxid2() in order to return a poisoned answer:
  ♦ Force the usage of RPC over SMB (`ncacn_np`) instead of RPC over TCP (`ncacn_ip_tcp`)
  ♦ Return the binding information exploiting a path validation bypass [1]:
    
    `ncacn_np:localhost/pipe/roguepotato\pipe\epmapper`

→ Result: the activator (RPCSS), instead of using the default named pipe `\pipe\epmapper`, will use a non-existent named pipe `\pipe\roguepotato\pipe\epmapper` for locating the endpoint information

[1] Credits to @itm4n and @jonasLyk --> https://itm4n.github.io/printspoofer-abusing-impersonate-privileges/
RoguePotato: the attack flow 3/4

**Step 1**
Trigger Istorage (Account with Impersonation privs)

**Step 2**
Fake ResolveOxid2 (Anonymous Logon)

**Step 3**
Fake epmapper pipe (Impersonate Network Service)
→ Create listener on the free named pipe `\\\pipe\roguepotato\pipe\epmapper` and wait for the connection from RPCSS, then we call `ImpersonateNamedPipeClient()` to impersonate the client

→ Should we expect a surprise?
RoguePotato: the attack flow 3/4

<table>
<thead>
<tr>
<th>Process</th>
<th>Thread ID</th>
<th>User</th>
<th>Impersonation Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoguePotato.exe</td>
<td>7460</td>
<td>13824 NT AUTHORITY\NETWORK SERVICE</td>
<td>Impersonation</td>
</tr>
</tbody>
</table>

Token Viewer

<table>
<thead>
<tr>
<th>Name</th>
<th>Privileges</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILTIN\Users</td>
<td>Mandatory, Enabled</td>
</tr>
<tr>
<td>Everyone</td>
<td>Mandatory, Enabled</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Mandatory, Enabled</td>
</tr>
<tr>
<td>NT AUTHORITY\Authenticated Users</td>
<td>Mandatory, Enabled</td>
</tr>
<tr>
<td>NT AUTHORITY\LogonSessionId_0_52500</td>
<td>Mandatory, Enabled, Owner, LogonId</td>
</tr>
<tr>
<td>NT AUTHORITY\NETWORK SERVICE</td>
<td>None</td>
</tr>
<tr>
<td>NT AUTHORITY\SERVICE</td>
<td>Mandatory, Enabled</td>
</tr>
<tr>
<td>NT AUTHORITY\This Organization</td>
<td>Mandatory, Enabled</td>
</tr>
<tr>
<td>NT SERVICE\RpcSs</td>
<td>Enabled, Owner</td>
</tr>
<tr>
<td>NT SERVICE\RpcSs</td>
<td>Owner</td>
</tr>
</tbody>
</table>
RoguePotato: the attack flow 4/4

**Step 1**
Trigger Istorage (Account with Impersonation privs)

**Step 2**
Fake ResolveOxid2 (Anonymous Logon)

**Step 3**
Fake epmapper pipe (Impersonate Network Service)

**Step 4**
RPCSS Token Kidnapping (Impersonate SYSTEM)
RoguePotato: the attack flow 4/4

→ The last step of the chain, the Token Kidnapping [1]

→ Get the PID of the “RPCSS” service

→ Open the process, list all handles and for each handle try to duplicate it and get the handle type

→ If handle type is “Token” and token owner is SYSTEM, try to impersonate and launch a process with CreateProcessAsUser() or CreateProcessWithToken()

[1] Credits to @cesarce --&gt; https://dl.packetstormsecurity.net/papers/presentations/TokenKidnapping.pdf
RoguePotato: SYSTEM shell popping :D

POC: https://github.com/antonioCoco/RoguePotato
Juicy2

→ **Release Date:** 30 May 2020
→ **Authors:** @decoder_it - @splinter_code

→ **Brief Description**
  ◆ Tricks the DCOM activation service in contacting a remote Rogue Oxid Resolver to force a specific DCOM component to authenticate to an arbitrary RPC server, resulting in a SYSTEM token stealing [1] [2]

→ **Requirements**
  ◆ The machine can make an outbound connection on port 135
  ◆ DCOM Running
  ◆ By default affects only Windows clients, no Windows Servers

[1] https://decoder.cloud/2020/05/30/the-impersonation-game/
Juicy2

Step 1
Trigger Istorage (Account with Impersonation privs)

Step 2
Fake Resolve0xid2 (Anonymous Logon)

Step 3
Fake epmapper pipe (Impersonate Network Service)

Step 4
RPCSS Token Kidnapping (Impersonate SYSTEM)

→ Similar to RoguePotato, but uses RPC over TCP (ncacn_ip_tcp) instead of RPC over SMB (ncacn_np)
→ JuicyPotato reloaded, it works for windows > 1803 with some limitations
**Juicy2**

**Step 1**
Trigger IStorage (Account with Impersonation privs)

**Step 2**
Fake Resolve0xid2 (Anonymous Logon)

**Step 3**
Fake IRemUnknown2 RPC Server (Impersonate in SecurityCallback)

\[ncacn_ip_tcp:127.0.0.1[9999]\]
Juicy2

→ Most of CLSIDs returns an **Identification** token, pretty useless...
→ Why this behavior?

```c
typedef struct _RPC_SECURITY_QOS {
    unsigned long Version;
    unsigned long Capabilities;
    unsigned long IdentityTracking;
    unsigned long ImpersonationType;
} RPC_SECURITY_QOS, *PRPC_SECURITY_QOS;
```

→ By default: **ImpersonationType**=RPC_C_IMP_LEVEL_IDENTIFY
→ Can be overridden at code level (server side) or by controlling the regkey `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost`
Any CLSID that override this behavior?

<table>
<thead>
<tr>
<th>CLSID</th>
<th>USER</th>
<th>TYPE</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>{354ff91b-5e49-4bdc-a8e6-1cb6c6877182}</td>
<td>DESKTOP-172UGPP\andrea</td>
<td>impersonation</td>
<td></td>
</tr>
<tr>
<td>{38f441ef-3d16-4227-8750-b2dacec5cfc5}</td>
<td>DESKTOP-172UGPP\andrea</td>
<td>impersonation</td>
<td></td>
</tr>
<tr>
<td>{90f18417-f0f1-484e-9d3c-59dceee5dbd8}</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>impersonation</td>
<td></td>
</tr>
<tr>
<td>{c41b1461-3f8c-4666-b512-6df24de566d1}</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>impersonation</td>
<td></td>
</tr>
<tr>
<td>{f8842f8e-dafe-4b37-9d38-4e0714a61149}</td>
<td>DESKTOP-172UGPP\andrea</td>
<td>impersonation</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>impersonation</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>impersonation</td>
<td></td>
</tr>
<tr>
<td>134</td>
<td>DESKTOP-172UGPP\andrea</td>
<td>impersonation</td>
<td></td>
</tr>
</tbody>
</table>

ActiveX Installer service, no Windows Server 😞
Other non-”potatoes” techniques

**Network Service Impersonation**
- **Release Date:** 25 April 2020
- **Authors:** @tiraniddo
- **Brief Description**
  - If you can trick the “Network Service” account to write to a named pipe over the “network” and are able to impersonate the pipe, you can access the tokens stored in RPCSS service (which is running as Network Service and contains a pile of treasures) and “steal” a SYSTEM token. [1]

**PrintSpoofer**
- **Release Date:** 2 May 2020
- **Authors:** @itm4n - @jonasLyk
- **Brief Description**
  - An exposed RPC interface of the Print Spooler service is vulnerable to a path validation bypass in which you can trick the service to write to a controlled named pipe and then impersonating the connection resulting in a SYSTEM token stealing. [2]

Relaying Potatoes Authentication
Basic idea

→ What if we relay the RPC authentication triggered by a potato exploit instead of impersonating? --> No more impersonation privileges required!
→ Machine authentication (NETWORK SERVICE/LOCAL SYSTEM) is not that useful...
→ Some CLSID to the rescue! If activated from session 0:
  ♦ BrowserBroker Class {0002DF02-0000-0000-C000-000000000046}
  ♦ AuthBrokerUI {0ea79562-d4f6-47ba-b7f2-1e9b06ba16a4}
  ♦ Easconsent.dll {5167B42F-C111-47A1-ACC4-8EABE61B0B54}
  ♦ ....
→ We can trigger an NTLM authentication over RPC from the user interactively logged on in Session 1 :D
DCE/RPC NTLM Relay cross protocols

→ “NTLM relay is a technique of standing between a client and a server to perform actions on the server while impersonating the client” [1]

→ In recent years most of the research/mitigations about NTLM Relaying were on SMB, HTTP, LDAP... What about RPC?

→ RPC -> HTTP and RPC -> LDAP cross protocol relay works!
  ♦ It requires the RPC authentication level is set to RPC_AUTHN_LEVEL_CONNECT (0x2)
  ♦ We need to deal also with NTLM mitigations: MIC and SIGNING
  ♦ In our scenario two interesting NTLM authentications took place:
    - Oxid Resolution (IObjectExporter::ResolveOxid2 call)
    - IRemUnknown Interface (IRemUnknown2::RemRelease call)

Dealing with MIC and SIGNING restrictions

Oxid Resolution
Dealing with MIC and SIGNING restrictions

```c
error_status_t ResolveOxid2(
    handle_t hRpc,
    OXID* pOxid,
    unsigned short cRequestedProtseqs,
    unsigned short arRequestedProtseqs[],
    DUALSTRINGARRAY** ppdsaOxidBindings,
    IPID* pipidRemUnknown,
    DWORD* pAuthnHint,
    COMVERSION* pComVersion
)

*pAuthnHint = RPC_C_AUTHN_LEVEL_CONNECT;
```
Dealing with MIC and SIGNING restrictions

```c
error_status_t ResolveOxid2
{
    handle_t    hRpc,
    OXID*      pOxid,
    unsigned short  cRequestedProtseqs,
    unsigned short  arRequestedProtseqs[],
    DUALSTRINGARRAY** ppdsaOxidBindings,
    IPID*      pidpRemUnknown,
    DWORD*     pAuthnHint,
    COMVERSION* pComVersion
}

sprintf_s(endpoint, MAX_PATH, "127.0.0.1[\%s]", port);
(*ppdsaOxidBindings)->aStringArray[0] = 0x07; //ncacn_ip_tcp
(*ppdsaOxidBindings)->aStringArray[securityOffset] = RPC_C_AUTHN_WINNT; // 0x0a
```
Dealing with MIC and SIGNING restrictions

Oxid Resolution
RemotePotato0 - EOP use case by relaying potato authentication to LDAP protocol
RemotePotato0: Demo
Mitigations

→ Change the sid type of the service to “WRITE RESTRICTED” [1]
  
  `sc.exe sidtype SampleService restricted`

→ Use virtual service accounts [2] (or create your own [3])
  
  `sc.exe config SampleService obj= "NT SERVICE\SampleService"`

→ Remove the impersonation privileges by specifying the only required privileges for the service(Least-Privilege) [1] [2]
  
  `sc.exe privs SampleService SeChangeNotifyPrivilege/SeCreateGlobalPrivilege`

Conclusion

→ For **Sysadmins**: never rely on default WSH configuration for segregating the services. Remember that also MS do not consider it a security boundary but just a “safety boundary”?????

→ For **Penetration Testers**: always run “whoami /priv” when you land to a new server and check for the SelImpersonatePrivilege. It’s a 1 click privesc to SYSTEM :D

→ For **service providers**: do not sell web servers (IIS) by creating a new virtual host on a shared machine, please...

→ “if you have Impersonation privileges you are SYSTEM!” @decoder_it
Thank You

Feel free to reach out! :D

@splinter_code
splintercod3@gmail.com