Bypassing LSA Protection (a.k.a. RunAsPPL) in Userland

Abusing the DefineDosDevice API actually has a second use, it's an Administrator to Protected Process Light (PPL) bypass. - James Forshaw (2018)
Who am I?

• Clément Labro
• Pentester @ SCRT
• Passionate about Windows Security
• MSRC's 2020 Most Valuable Security Researcher
• Maintainer of PrivescCheck

• Blog: https://itm4n.github.io
• Twitter: @itm4n
• Github: @itm4n
We all know LSA Protection, or do we?

What I knew back then.

- Configure a simple registry key and reboot, that's it!
- From now on, other processes (Mimikatz, procdump, ...) can't access LSASS
- ...unless you go from Ring 3 to Ring 0, a.k.a. the Kernel, using a custom driver.

How/why does it work?

- Binaries that are not signed can't open LSASS?
- ...🤔 Well, let's do some research...
How to enable LSA Protection

Configure the **RunAsPPL** value in the registry and reboot

- HKLM\SYSTEM\CurrentControlSet\Control\Lsa -> RunAsPPL = 0x00000001

Remarks / Limitations

- **ℹ️** Only available starting from **Windows 8.1 / Server 2012 R2**
- **⚠️** If **Secure Boot** is enabled, the setting is persistent (stored in the UEFI firmware)!
- **⚠️** Prevents non-signed plug-ins and drivers (**smart card readers**, **password filters**, etc.) from being loaded in LSASS.

How good is this LSA Protection?

LSA Protection against Mimikatz - Round 1

✔ The current user is an administrator
✔ The current user has `SeDebugPrivilege`
❌ `0x00000005` = "Access is denied"

OpenProcess failed, the Kernel refused to return a process handle to the caller.
How good is this LSA Protection?

LSA Protection against Mimikatz - Round 2

- Mimikatz is shipped with a **signed driver**: mimidrv.sys (load it with `!+`)
- Use the command `!processprotect /process:lsass.exe /remove`
- This drops the protection flag of the Process object in the Kernel memory
- Easily flagged by AV/EDR

2021-10-12 - Bypassing LSA Protection (a.k.a. RunAsPPL) in Userland - Clément Labro
Protected Processes (Light)

Protected Processes (PP)
- Introduced with Windows Vista / Server 2008
- Objective: protect media content and comply with Digital Rights Management!
- The image file had to be signed with a special Windows Media Certificate

Protected Processes Light (PPL)
- Introduced with Windows 8.1 / Server 2012 R2
- A protection level is added (signer type)
  - => Some processes are more protected than others
Protection levels & Signer types

Source: https://googleprojectzero.blogspot.com/2018/10/injecting-code-into-windows-protected.html

2021-10-12 - Bypassing LSA Protection (a.k.a. RunAsPPL) in Userland - Clément Labro
## A few examples

### Windows Defender - `MsMpEng.exe`

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU</th>
<th>Private Bytes</th>
<th>Working Set</th>
<th>PID</th>
<th>Description</th>
<th>Company Name</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>MsMpEng.exe</td>
<td>0.03</td>
<td>499,324 K</td>
<td>152,952 K</td>
<td>5312</td>
<td>AntiMalware Service Executable</td>
<td>Microsoft Corporation</td>
<td>PsProtectedSigner/Antimalware-Light</td>
</tr>
</tbody>
</table>

### LSASS when RunAsPPL is enabled - `lsass.exe`

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU</th>
<th>Private Bytes</th>
<th>Working Set</th>
<th>PID</th>
<th>Description</th>
<th>Company Name</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>lsass.exe</td>
<td>0.01</td>
<td>9,952 K</td>
<td>9,956 K</td>
<td>732</td>
<td>Local Security Authority Proc...</td>
<td>Microsoft Corporation</td>
<td>PsProtectedSignerLsa-Light</td>
</tr>
</tbody>
</table>

### A critical process - `winint.exe`

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU</th>
<th>Private Bytes</th>
<th>Working Set</th>
<th>PID</th>
<th>Description</th>
<th>Company Name</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>winint.exe</td>
<td>1.700 K</td>
<td>384 K</td>
<td></td>
<td>1004</td>
<td>Windows Start-Up Application</td>
<td>Microsoft Corporation</td>
<td>PsProtectedSignerWinTcb-Light</td>
</tr>
</tbody>
</table>

### SgrmBroker - `SgrmBroker.exe`

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU</th>
<th>Private Bytes</th>
<th>Working Set</th>
<th>PID</th>
<th>Description</th>
<th>Company Name</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>SgrmBroker.exe</td>
<td>5.900 K</td>
<td>4,756 K</td>
<td></td>
<td>10644</td>
<td>System Guard Runtime Monitor</td>
<td>Microsoft Corporation</td>
<td>PsProtectedSignerWinTcb</td>
</tr>
</tbody>
</table>
How is the protection level determined?

The image file's certificate contains a special "EKU" field.
Protected Processes in a nutshell

- **Protection level:** Protected Process (PP) or Protected Process Light (PPL)
- **Signer type:** WinTCB > Windows > Lsa > AntiMalware > Authenticode
- **LSA Protection:** if RunAsPPL=1 => LSASS runs as a PPL with the signer type Lsa

Here are the basic rules:

- **X** A "standard" process **cannot open a PP(L)**
- **!** A PP(L) can open another PP(L) only if its protection level is **greater or equal**
- **i** A PP(L) can be created by any user as long as the **image file is signed by MS** and its certificate contains the appropriate **EKU** values.
- **✓** If I'm able to **run arbitrary code inside a PPL with WinTCB level**, I can open any PPL.
How do PPs and PPLs handle DLL loading?

The EXE must be **digitally signed by Microsoft** "impossible" to run arbitrary code.

... but what about **imported DLLs**? They must be signed as well but...

**DLL search order reminder:**

1. DLLs already loaded in memory  
2. Known DLLs  
3. Application's directory  
4. System directories (C:\Windows\System32, C:\Windows\System, ...)  
5. Current directory  
6. `%PATH%` directories
Known DLLs

Known DLLs are Section objects that are stored in the Object directory `\KnownDlls`
Known DLLs: PP vs PPL

Protected Process (PP)
- Known DLLs are loaded from the disk. ➡️ The digital signature is always verified.

Protected Process Light (PPL)
- Known DLLs are loaded from the existing Sections. ➡️ No signature validation!

2021-10-12 - Bypassing LSA Protection (a.k.a. RunAsPPL) in Userland - Clément Labro
Create your own Known DLL entry!

As an administrator, create a new Section object in `\KnownDlls` and map your own image file ➔ DLL hijacking for the win! 😎

Hmm... It's not that simple! 😞 The `\KnownDlls` directory and the `KnownDlls` registry key are protected with a "Process Trust Label".

```
PS C:\Windows\system32> Set-ExecutionPolicy Bypass -Scope Process -Force
PS C:\Windows\system32> Import-Module MIOleObjectManager
PS C:\Windows\system32> $KnownDlls = Get-Item "\KnownDlls"
PS C:\Windows\system32> $KnownDlls.SecurityDescriptor.ProcessTrustLabel | f1
```

Type : ProcessTrustLabel
User : TRUST LEVEL\ProtectedLight-WinTcb
Sid : S-1-19-512-8192
Flags : None
Mask : 00020003
Introducing the DefineDosDevice API

Abusing the DefineDosDevice API actually has a second use, it's an Administrator to Protected Process Light (PPL) bypass. - James Forshaw (2018)


```c
BOOL DefineDosDeviceW(DWORD dwFlags, LPCWSTR lpDeviceName, LPCWSTR lpTargetPath);
```

Examples: plug a USB key, map a network share, etc.

```c
DefineDosDeviceW(dwFlags, L"E:", "\Device\HarddiskVolume5");
```

DefineDosDevice is a wrapper for an RPC function exposed by the CSRSS service.

The CSRSS service is executed as a PPL with the signer type WinTCB!
A TOCTOU vulnerability in DefineDosDevice

```
DWORD dwFlags = DDD_NO_BROADCAST_SYSTEM | DDD_RAW_TARGET_PATH;
DefineDosDeviceW(dwFlags, L"DEVICE_NAME", L"TARGET_PATH");
```

1. **Impersonate the client**, try to open `\??\DEVICE_NAME` and *revert to self*.  
2. If it exists, determine whether it's *global* (i.e. object path start with `\GLOBAL??\`).  
3. If so, **disable impersonation**. (i.e. exec as `SYSTEM` + PPL/WinTCB)  
4. If the symbolic link (step 1) exists, delete it.  
5. (If impersonation is *enabled*, *impersonate the client again*.)  
6. **Create the symbolic link** `\??\DEVICE_NAME` -> `TARGET_PATH`.  
7. (If impersonation is *enabled*, *revert to self*.)  
8. Mark the new symbolic link object as "*Permanent*".
A TOCTOU vulnerability in DefineDosDevice

Two operations:

- **Step 1/2**: a check is done in the context of the RPC client.
- **Step 6**: the symbolic link could be created in the context of the service.

The same path in both cases but \??\DEVICE_NAME = ...

- \GLOBAL??\DEVICE_NAME for **SYSTEM**
- \Sessions\0\DosDevices\0000000-XXXXXXXX\DEVICE_NAME for any other user

We need to find a value for DEVICE_NAME such that \??\DEVICE_NAME resolves to: ...

- A global object ( \GLOBAL??\... ) when the caller is impersonated.
- \KnownDlls\foo.dll when interpreted as **SYSTEM**
The exploit

We can exploit this **TOCTOU** using a path such as `GLOBALROOT\KnownDlls\foo.dll`.

1. The service will open `\\GLOBALROOT\KnownDlls\foo.dll` as the RPC client.

   ```
   \GLOBALROOT\KnownDlls\foo.dll = \Sessions\0\DosDevices\00000000-XXXXXXXX\GLOBALROOT\KnownDlls\foo.dll
   -> \GLOBAL??\KnownDlls\FOO.dll
   ```

2. This object does not exist but we can create it, and its path starts with `\GLOBAL??\`.

3. The object is considered as "global" so **impersonation is disabled**.

6. **Create the symlink as SYSTEM** `\\GLOBALROOT\KnownDlls\foo.dll`.

   ```
   \GLOBALROOT\KnownDlls\foo.dll = \GLOBAL??\GLOBALROOT\KnownDlls\FOO.dll
   -> \KnownDlls\FOO.dll
   ```

🏆 Enjoy your new symbolic link `\KnownDlls\foo.dll`!
Running arbitrary code inside a PPL

**Objective** - Hijack a DLL of an EXE we can execute as a PPL with the level WinTCB.

- Only 4 built-in executables match this criteria.
- The best candidate is by far services.exe (SCM).
- It loads several DLLs which are not Known DLLs (depends on the OS version).
PPLdump

https://github.com/itm4n/PPLdump

```plaintext
c:\Temp>PPLdump64.exe -v lsass lsass.dmp
[*] Found a process with name 'lsass' and PID 712
[*] Requirements OK
[*] DLL to hijack: EventAggregation.dll
[*] Impersonating SYSTEM...
[*] Created object Directory: '\GLOBAL??\KnownDlls'
[*] Created symbolic link: '\GLOBAL??\KnownDlls\EventAggregation.dll'
[*] Created symbolic link: '\??\GLOBALROOT -> \GLOBAL??'
[*] DefineDosDevice OK
[*] Impersonating SYSTEM...
[*] The symbolic link was successfully created: '\KnownDlls\EventAggregation.dll' -> '\KernelObjects\EventAggregation.dll'
[*] Mapped payload DLL to: '\KernelObjects\EventAggregation.dll'
[*] Started protected process, waiting...
(DLL) [*] DLL loaded.
(DLL) [*] KnownDll entry 'EventAggregation.dll' removed.
(DLL) [*] DumpProcessMemory: SUCCESS
[+] Dump successful! ;)
```
References

- https://itm4n.github.io/lsass-runasppl/
- https://docs.microsoft.com/en-us/sysinternals/resources/windows-internals