How We Micropatched a Publicly Dropped 0day in Task Scheduler (CVE-UNKNOWN)

Being Who You Are Can be a Bad Thing if You're a System Service

by Mitja Kolsek, the 0patch Team

Earlier this week security researcher SandboxEscaper published details and proof-of-concept (POC) for a "0day" local privilege escalation vulnerability in Windows Task Scheduler service, which allows a local unprivileged user to change permissions of any file on the system - and thus subsequently replace or modify that file.

As the researcher's POC demonstrates, one can use this vulnerability to replace a system executable file and wait for a privileged process to execute it. In particular, it was shown that a printing-related DLL could be replaced and then executed by triggering the Print Spooler Service to load it. (The latter being a legitimate system operation, only used for demonstrating how replacing a system executable leads to elevated privileges. One could alternatively replace one of a large number of other system executables, or perhaps even a configuration file that gets loaded by a privileged process.)

SandboxEscaper's documentation properly identifies the problem being in Task Scheduler's `SchRpcSetSecurity` method, which is externally accessible via Advanced Local Procedure Call (ALPC) facility. This method, which can be called by any local process, sets a desired security descriptor (`sddl`) on a task or folder, i.e., on a provided file path (`path`).

```
HRESULT SchRpcSetSecurity(
   [in, string] const wchar_t* path,
   [in, string] const wchar_t* sddl,
   [in] DWORD flags
 );
```

SandboxEscaper noticed that this method fails to *impersonate* the requesting client when setting the security descriptor, which results in Task Scheduler changing the access control list of the chosen file or folder as Local System user even if the user calling this method is a low-privileged user. Impersonation is a feature where, to put it simply, a process running as user A gets a request for some action from user B and performs this action disguised as user B, borrowing user B's permissions for that. Task Scheduler is such a process running as user Local System, and when some other user calls its `SchRpcSetSecurity` method, it should impersonate the caller to perform the file operation using their identity - but apparently it doesn't, and uses its own powerful permissions to do so.

What the POC does to demonstrate this issue is:

1. create an `UpdateTask.job` file in folder `%SystemRoot%\Tasks` where any user is allowed to create files (this is needed in the process of creating a new scheduled task, and non-admin users are allowed to do that); however, this file is not an ordinary file but rather a *hard link* pointing to a system file `PrintConfig.dll`. (which non-system user can't modify or replace);
2. call Task Scheduler's `SchRpcSetSecurity` method to change permissions on `UpdateTask.job` such that everyone will be able to modify it; this actually changes permissions of the linked-to `PrintConfig.dll` file, which thus becomes user-modifiable;
3. replace `PrintConfig.dll` with a "malicious" DLL that simply launched Notepad;
4. trigger the Print Spooler service to load and execute the modified `PrintConfig.dll` using its own Local System identity.

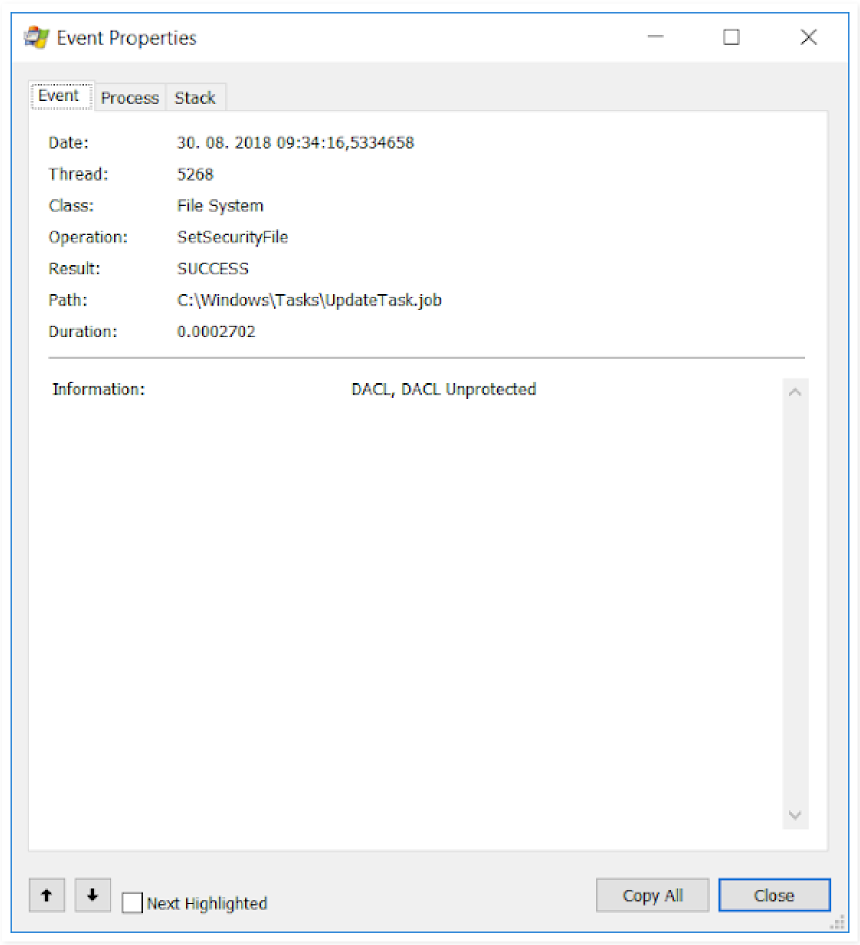
Vulnerability Analysis

The problem is clearly in step #2, which allows a non-admin user to change permissions on a system executable, and one can quickly assess the root cause of the problem to be a combination of two facts:

the `SetSecurityFile` file system operation, and

2. Task Scheduler being willing to perform `SchRpcSetSecurity` on a hard link.

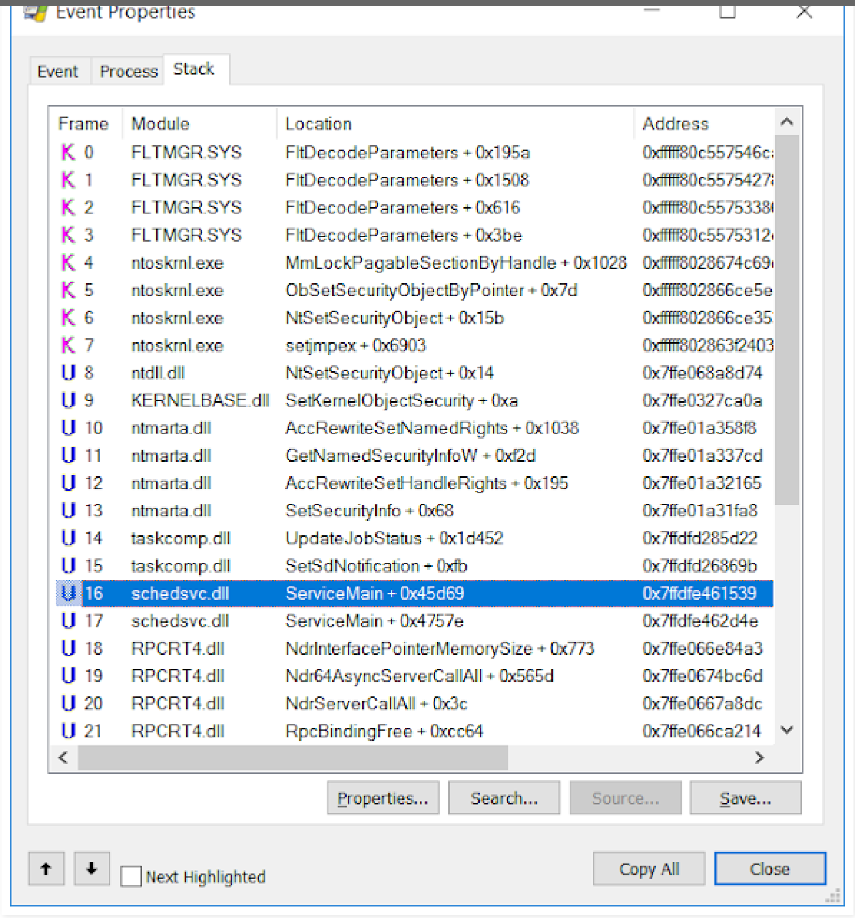
After running the POC, we took a look at operations performed on `UpdateTask.job` with Process Monitor, and found the one that changes permissions:



```
Date:        30. 08. 2018 09:34:16,5334658
Thread:      5268
Class:       File System
Operation:   SetSecurityFile
Result:      SUCCESS
Path:        C:\Windows\Tasks\UpdateTask.job
Duration:    0.0002702

Information:                 DACL, DACL Unprotected
```

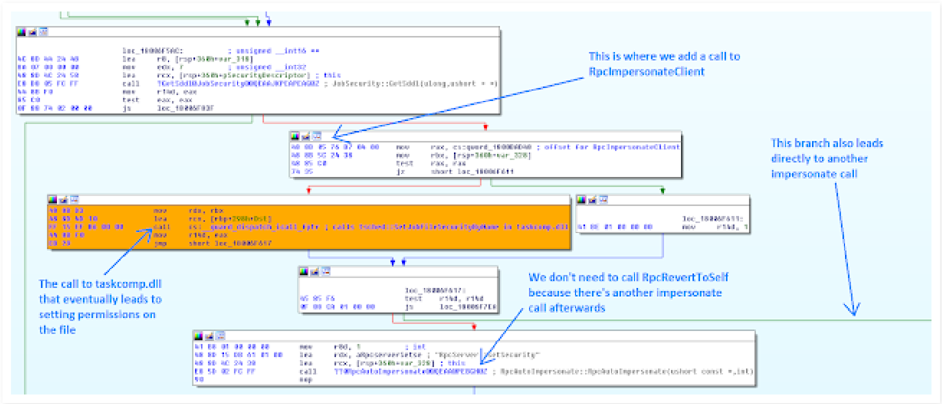
So we took a look at its call stack to see who invoked this action:



Frame		Module	Location	Address
K	0	FLTMGR.SYS	FltDecodeParameters + 0x195a	0xfffff80c557546c
K	1	FLTMGR.SYS	FltDecodeParameters + 0x1508	0xfffff80c5575427
K	2	FLTMGR.SYS	FltDecodeParameters + 0x616	0xfffff80c5575338
K	3	FLTMGR.SYS	FltDecodeParameters + 0x3be	0xfffff80c5575312
K	4	ntoskrnl.exe	MmLockPagableSectionByHandle + 0x1028	0xfffff8028674c69
K	5	ntoskrnl.exe	ObSetSecurityObjectByPointer + 0x7d	0xfffff802866ce5e
K	6	ntoskrnl.exe	NtSetSecurityObject + 0x15b	0xfffff802866ce35
K	7	ntoskrnl.exe	setjmpex + 0x6903	0xfffff802863f2403
U	8	ntdll.dll	NtSetSecurityObject + 0x14	0x7ffe068a8d74
U	9	KERNELBASE.dll	SetKernelObjectSecurity + 0xa	0x7ffe0327ca0a
U	10	ntmarta.dll	AccRewriteSetNamedRights + 0x1038	0x7ffe01a358f8
U	11	ntmarta.dll	GetNamedSecurityInfoW + 0xf2d	0x7ffe01a337cd
U	12	ntmarta.dll	AccRewriteSetHandleRights + 0x195	0x7ffe01a32165
U	13	ntmarta.dll	SetSecurityInfo + 0x68	0x7ffe01a31fa8
U	14	taskcomp.dll	UpdateJobStatus + 0x1d452	0x7ffdfd285d22
U	15	taskcomp.dll	SetSdNotification + 0xfb	0x7ffdfd26869b
U	16	schedsvc.dll	ServiceMain + 0x45d69	0x7ffdfe461539
U	17	schedsvc.dll	ServiceMain + 0x4757e	0x7ffdfe462d4e
U	18	RPCRT4.dll	NdrInterfacePointerMemorySize + 0x773	0x7ffe066e84a3
U	19	RPCRT4.dll	Ndr64AsyncServerCallAll + 0x565d	0x7ffe0674bc6d
U	20	RPCRT4.dll	NdrServerCallAll + 0x3c	0x7ffe0667a8dc
U	21	RPCRT4.dll	RpcBindingFree + 0xcc64	0x7ffe066ca214

Okay, there's `schedsvc.dll` (Task Scheduler's executable) making a call to `taskcomp.dll` (Task Scheduler's helper library), which ends up with a call to kernel's `NtSetSecurityObject`. So we disassembled `schedsvc.dll` and `taskcomp.dll` to see what's going on in there at the identified locations. What we found was interesting.

The call from `schedsvc.dll` to `taskcomp.dll` occurs in function `RpcServer::SetSecurity` (in the orange block):

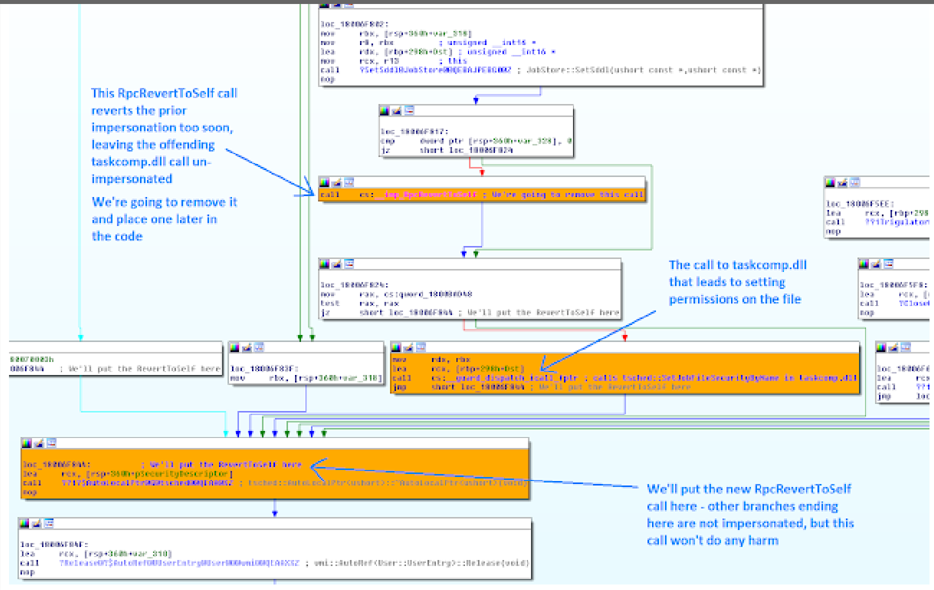

This is where we add a call to RpcImpersonateClient
This branch also leads directly to another impersonate call
The call to taskcomp.dll that eventually leads to setting permissions on the file
We don't need to call RpcRevertToSelf because there's another impersonate call afterwards

We were expecting to see code without any impersonation here, but actually found impersonation being used - just that the call that sets file permissions is done before the impersonation (in the lowest code block) begins.

The plan was clear: let's begin impersonation before the offending call to make sure that said call will be impersonated. So we created a micropatch with a single patchlet containing a call to `RpcImpersonateCient` and placed it at the beginning of the block preceding the orange block. How about reverting the impersonation? It turns out that wasn't needed because all code execution paths were leading directly to another impersonation call without making any other kernel calls that might be affected by our impersonation.

We tried this micropatch, but the exploit still worked !?! What was going on?

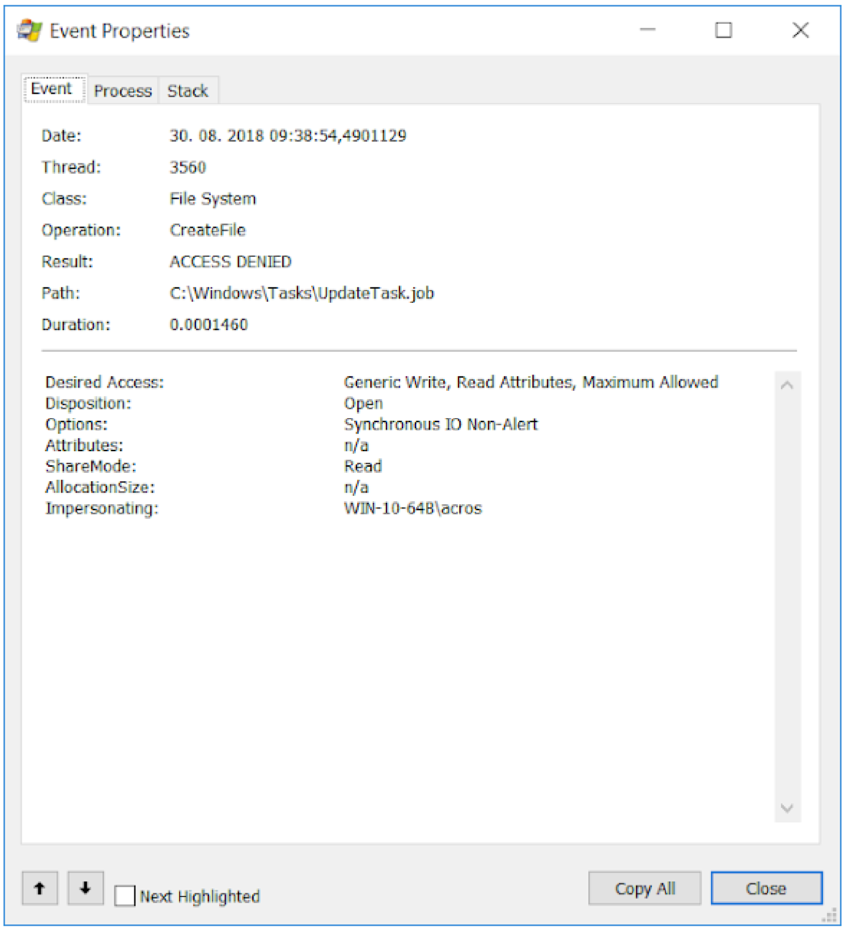
It turned out that there is another permissions-setting call in function `RpcServer::SetSecurity`, possibly a fallback mechanism in case the first one failed. So we made the first one fail, and the second one came to the rescue - again without impersonation (the middle orange block).


This RpcRevertToSelf call reverts the prior impersonation too soon, leaving the offending taskcomp.dll call un-impersonated
We're going to remove it and place one later in the code
The call to taskcomp.dll that leads to setting permissions on the file
We'll put the new RpcRevertToSelf call here - other branches ending here are not impersonated, but this call won't do any harm

In this case, we can see a call to `RpcRevertToSelf` right before the offending call, which means that previous impersonation was reverted too soon to include the said call.

What we did here was remove the premature `RpcRevertToSelf` call and insert a replacement `RpcRevertToSelf` call to the code block following the offending call. While this block has many other branches leading to it, we checked that these are not impersonated which means our inserted call won't erroneously prematurely revert some other impersonation.

So finally, our micropatch worked and Process Monitor showed this instead:


Event Properties

Event | Process | Stack

Date:	30. 08. 2018 09:38:54,4901129
Thread:	3560
Class:	File System
Operation:	CreateFile
Result:	ACCESS DENIED
Path:	C:\Windows\Tasks\UpdateTask.job
Duration:	0.0001460

Desired Access:	Generic Write, Read Attributes, Maximum Allowed
Disposition:	Open
Options:	Synchronous IO Non-Alert
Attributes:	n/a
ShareMode:	Read
AllocationSize:	n/a
Impersonating:	WIN-10-64B\acros

You can see the "Impersonating" line, proving that we have successfully forced Task Scheduler to impersonate the calling user when trying to set permissions on `UpdateTask.job`. Now, since this file was a hard link to another file which our user had insufficient permissions to modify ACL for, the result was ACCESS DENIED, as it should be.

This is the source code of our micropatch, with all of its 4 instructions in three patchlets:

```
; Patch for VULN-4051 in schedsvc.dll version 10.0.17134.1 64bit
MODULE_PATH "..\AffectedModules\schedsvc.dll_10.0.17134.1_64bit\schedsvc.dll"
PATCH_ID 328
```

```
VULN_ID 4051
PLATFORM win64


patchlet_start
 PATCHLET_ID 1
 PATCHLET_TYPE 2
 PATCHLET_OFFSET 0x6F5CB
 PIT rpcrt4.dll!RpcImpersonateClient
 code_start
  xor ecx, ecx ; Impersonating the client that made the request
  call PIT_RpcImpersonateClient
 code_end
patchlet_end


patchlet_start
 PATCHLET_ID 2
 PATCHLET_TYPE 2
 PATCHLET_OFFSET 0x6F81E
 JUMPOVERBYTES 6 ; We eliminate the 6-byte call to RevertToSelf
 code_start
  nop
 code_end
patchlet_end


patchlet_start
 PATCHLET_ID 3
 PATCHLET_TYPE 2
 PATCHLET_OFFSET 0x6F844
 PIT rpcrt4.dll!RpcRevertToSelf
 code_start
  call PIT_RpcRevertToSelf
 code_end
patchlet_end
```

Micropatch In Action

This video shows our micropatch in action.

Frequently Asked Questions

Q: Which Windows versions does this micropatch apply to?

Currently we have instances of this micropatch applying to:

1. fully updated 64bit Windows 7 [added on 9/6]
2. fully updated 64bit Windows Server 2008 [added on 9/6]
3. fully updated 64bit Windows 10 version 1607 [added on 9/4]
4. fully updated 64bit Windows 10 version 1709 [added on 9/5]
5. fully updated 64bit Windows 10 version 1803
6. fully updated 64bit Windows Server 2016
7. fully updated 64bit Windows Server 1803 [added on 9/6]

our micropatch on Windows Server 1803 in a real-time Twitter DM session!

We can quickly port the micropatch to other affected versions but we'll only do that on request (support@0patch.com). As far as we know at this point, the vulnerability was confirmed to also be present and exploitable on 32bit Windows 10 and 32bit Windows 7, so it's safe to assume that at least all Windows versions from Windows 7 and Windows Server 2008 are likely affected.

Q: Will modifying the exploit allow attackers to bypass this micropatch?

No, and that's one of the significant advantages of changing the code compared to signature- or behavior-based exploit prevention systems. For instance, while most antivirus products will detect the original POC by now, Will Dormann modified the POC and showed that it went undetected. Such modifications *always* allow for bypassing detection-based systems, while fixing the code actually removes the vulnerability. There is simply nothing there to bypass. There is *no more efficient and reliable way* to address a vulnerability than to actually remove it. (Although the entire industry built *around* vulnerabilities will try to convince you otherwise.)

Q: How do we apply this micropatch?

If you have 0patch Agent already installed, this micropatch is already downloaded and applied so you don't have to do anything. Otherwise, download and launch the 0patch Agent installer, create a free 0patch account and register the agent to that account. You will immediately receive all micropatches including this one, and it will automatically get applied to Task Scheduler.

Q: Is this patch functionally identical to how Microsoft will fix it?

Obviously we can't know that. As we always claim, the original vendor - with their internal knowledge of the product - is best-positioned to correct their own code. Nobody else knows all the possible side effects of a code change as well as they do (granted, with large products even they often don't see everything) and in an ideal world software vendors would be issuing micropatches like this to quickly and painlessly fix vulnerabilities. That said, Microsoft may do the same as we did, but they may also prevent Task Scheduler from changing permissions on hard links. Or they may find that they need to support hard links *and* not impersonating the user is essential for some other operation that Task Scheduler performs - and will make a substantial code change. We often create micropatches after the vendor has issued the official update, which allows us to see what they did and ideally replicate their logic in a micropatch. With a 0day, this is obviously not possible.

You should therefore consider our micropatch a temporary solution while waiting for the official fix.

Q: What will happen on Patch Tuesday?

When Microsoft makes their official fix available, you simply apply it as you would if you had never heard of 0patch. Applying it will automatically obsolete this micropatch on your computer as the update will replace a vulnerable executable with a fixed one, thereby changing its cryptographic hash. Since our micropatches are associated with specific hashes, this will make the micropatch inapplicable without intervention on either your end or ours.

Q: Can we keep using this micropatch instead of applying Microsoft's update?

We strongly recommend against that. Microsoft's update will not only fix this issue in a more informed way, but will also bring fixes for other vulnerabilities that we don't have micropatches for. Yes, we hate losing hours of our lives to updating our systems too, but wouldn't dream of outright replacing official updates with our micropatches ;)

Q: How can you provide a micropatch so quickly compared to original vendors?

While having a micropatch candidate ready 24 hours after a 0day was dropped is quick relative to today's standards of software patching, a couple of things must be considered:

1. Software vendors know their products much better than we do, and are likely to create a more comprehensive code fix than we can without their knowledge and source code. That takes more time than writing a micropatch.
2. Software vendors bundle numerous fixes together in a "fat update" that replaces entire executables, which requires a lot more testing across the board. We test our micropatches with focused tests targeting only the patched code.
3. "Fat updates" are a huge problem for users and vendors when something goes wrong, which is why software vendors are even more wary of issuing a defective update. Of course a micropatch also can be flawed, but it can be revoked remotely and instantly replaced with a corrected version without users ever noticing anything. That said, we will always have "fat updates" for substantial functional changes, we're just arguing that we may not need them this frequently because most vulnerabilities could be patched with micropatches.
4. Software vendors must issue patches for all supported versions, and extensively test all of them. We currently only have this micropatch for ~~two three four six~~ seven affected Windows versions. Nevertheless, porting to other versions, basic testing and deployment would take us about two hours of effort for each additional version, so that could still be done in one day.

All that said, comparing our speed with software vendors' must also account for the difference in our deliverables. A micropatch can be quickly created, deployed to all computers in a hour's time and applied without even the slightest disturbance to users. But it must be considered a temporary security measure until the official patch can be applied.

Q: What should we do if we encounter problems with Task Scheduler after applying this micropatch?

Obviously we can't guarantee that our micropatch won't cause some unwanted side effects, e.g., with non-admin users editing existing scheduled tasks under certain circumstances. (Then again, software vendors can't guarantee that either.) The rule of thumb for using 0patch (or any other 3rd party behavior-changing product like antivirus or malware blockers) should be to first disable the agent and see if the problem persists, before contacting the original software vendor for the affected product. If the problem persists, the culprit is unlikely to be the micropatch. If the problem goes away, it's probably us and we'd like to hear from you at support@0patch.com.

Fortunately, in contrast to standard "fat update" patching that software products employ today, 0patch allows you to *instantly revert a patch with a click of a button.*

Cheers!

@mkolsek
@0patch