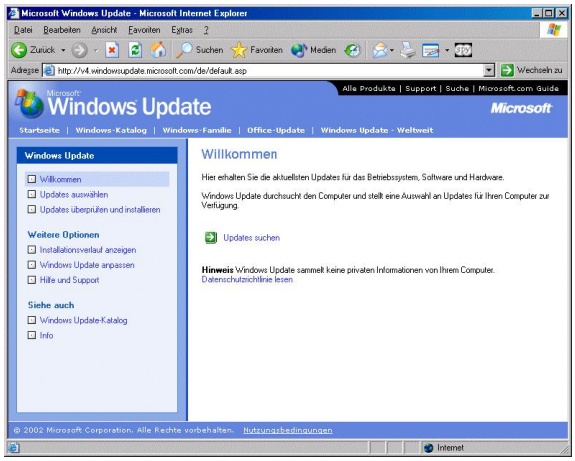
Inside Windows-Update

› When you connect your computer to Microsofts website windowsupdate.com, you reveal a lot of information about your computer to Microsoft. This article shows bit for bit, which data is transferred to Redmond and what Microsoft could learn from it.

› VON MIKE HARTMANN

The privacy policy of Windows Update used to be simple. A large banner stating that everything was done "without sending any information to Microsoft" and the lack of suspicious network activity assured us that Microsoft valued our privacy. However, times have changed.



Data collector or update vending machine? Microsoft learns alot about your computer, when you connect to Windows Update.

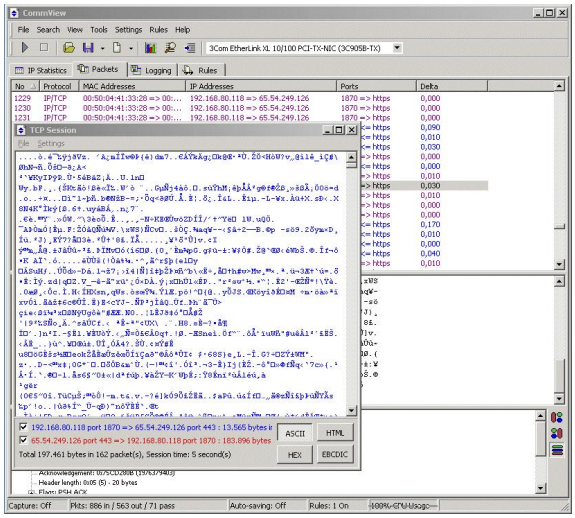
Newer versions of Windows Update send a significant amount of data to the Microsoft update server. Unfortunately, the exact content of the transmission and its relevance to the users' privacy have up to now not been disclosed. Fortunately, times are changing again. This article explains in detail which personal information Windows Update transmits to the Microsoft server.

› Basic observations

Windows Update consists of a few HTML pages with a large amount of embedded Java Script code and a COM component. These building blocks are downloaded when a user opens the Windows Update URL

http://v4.windowsupdate.microsoft.com/default.asp

in Internet Explorer. The main task of the Java Script code, which is easy to analyze because its source code can be examined, is to interact with the user. The more interesting functionality is unfortunately hidden inside the COM component.

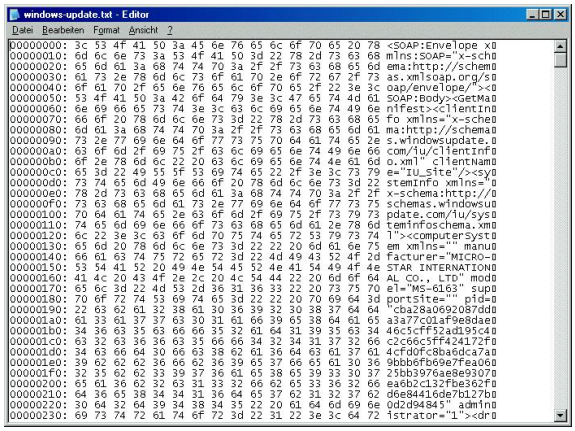


Encrypted: A network sniffer doesn't help finding out, what information is transmitted to Microsoft.

When the user selects to list the available updates, Windows Update does not only transfer data from the Microsoft server to the user's computer. A few kilobytes of data are also transfered in the opposite direction from the user's computer to the Microsoft server. This is what we are interested in, but unfortunately the data is transmitted through an encrypted SSL connection and therefore cannot be examined with a network packet analyzer.

› A black box approach

Our first approach exploits the fact that Windows Update uses the WinInet API to handle the SSL connection and data transmission. By hooking into the *HttpOpenRequest()* function and examining the passed arguments we learn that the data that is sent from the user's computer to the Microsoft server through the SSL connection consists of HTTP POST requests. Hooking also into the *InternetWriteFile()* function we are then able to peek at the data that is posted to the Microsoft server before it is encrypted.



Hooked into communication: With our dump-tool we are able to see exactly which data is transferred to Windows-Update.

This is what the tecDump utility does. It asks for the name of a log file, starts Internet Explorer, opens the Windows Update URL, and hooks into InternetWriteFile(). When Scan for updates is selected in Windows Update, the tecDump utility becomes active and writes a hex dump of the intercepted data to the chosen log file.

The utilities that we provide with this article are based on undocumented behavior of Windows Update. It is likely that an update, e.g. a new service pack or a hotfix, will change this behavior and therefore render the tools unusable.

› The communication protocol

The tecDump utility reveals that Windows Update uses the POST requests to transmit SOAP (Simple Object Access Protocol) messages to the Microsoft server. SOAP is the XML-based standard protocol for communicating with web-services, an elementary building block of Microsoft's .NET strategy. The intercepted messages have the following format.

```
» <SOAP:Envelope xmlns:SOAP="x-schema:http://schemas.xmlsoap.org/soap/envelope/">
» <SOAP:Body>
» <GetManifest>
» <clientInfo [...]> [...] </clientInfo>
» <systemInfo [...]>
» <computerSystem [...]>
» [...]
» </computerSystem>
» <platform [...]>
» [...]
» </platform>
» <locale [...]>
» [...]
» </locale>
» <devices [...]>
» [...]
» </devices>
» </systemInfo>
» <query [...]> [...] </query>
» </GetManifest>
» </SOAP:Body>
» </SOAP:Envelope>
```

The *» <SOAP:Envelope> «* and *» <SOAP:Body> «* tags are required by the SOAP standard to encapsulate the payload to be transmitted. Windows Update uses the payload to implement a RPC (Remote Procedure Call) mechanism. The SOAP messages posted to the Microsoft server by Windows Update contain the name of a function to be executed on the server and the arguments to be passed to the function. When receiving a message from Windows Update the server then runs the specified function with the specified arguments and encapsulates the result into another SOAP message, which it then transmits back to Windows Update in the HTTP response to the pending POST request.

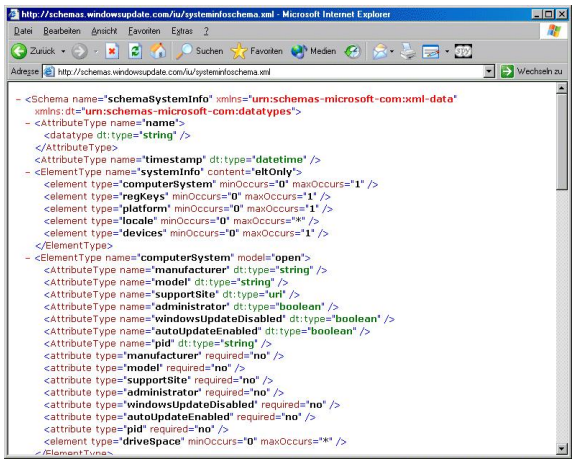
These tools do use undocumented functions in the Windows-API and of the Windows Update service. A new service pack or hotfix might render them unusable. Thus we cannot guarantee the ongoing functionality of the tools and we cannot support them.

› The interaction with the Microsoft server

In the observed messages the function to be called is *GetManifest*. *GetManifest* asks the Microsoft server to return information about available updates. The function requires three arguments - *clientInfo*, *systemInfo*, and *query*. The XML schemas for the first two arguments have the following URLs, the XML schema for the query argument is not known to us.

http://schemas.windowsupdate.com/iu/clientInfo.xml

http://schemas.windowsupdate.com/iu/systeminfoschema.xml



Web-Service: Windows-Update is a normal webservice. Here is the schema for the SOAP-request.

The purpose of the *clientInfo* parameter seems to be to identify and authenticate the user accessing the Microsoft server. Currently the server is accessible to everybody and the *clientInfo* argument only contains the generic user identification "IU_Site". However, being able to restrict access to updates based on the identity of a user, e.g. supplying certain updates only to users that have a support contract, seems to have been a requirement during the design of Windows Update, although this feature is not currently used.

› Parameter: query

The *query* argument specifies the level of detail of the information to be returned. In the observed messages four different levels are used - Provider, Products, Items, and DriverUpdates.

Provider - The Microsoft server returns all product categories for which it supplies updates, e.g. ie60x for Internet Explorer 6.x or winxp for the various versions of Windows XP, and rules for determining whether a product that belongs to a specific category is installed. The rules are boolean expressions written in a simple XML-based language. For instance, the rule for product category ie60x is that one of the products that belong to the product category ie60x is installed if the registry value HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ Internet Explorer\ Version is lower than 7.0 and higher than or equal to 6.0.

Products - The query argument contains a list of product categories and the Microsoft server returns the products that belong to each product category, e.g. winxp.windowsxpcore for the components of Windows XP that are shared by all versions, winxp.windowsxp for the components specific to the professional edition and the home edition of Windows XP, or winxp.windowsnetserverfamily for the components specific to the .NET server edition.

Items - The query argument contains a list of products and the Microsoft server returns detailed information about the updates that are available for each product.

DriverUpdates - The Microsoft server returns a list of driver updates that are relevant for the computer that is being updated. Note that the SOAP messages for calls to GetManifest at this level are several times as long as the other observed SOAP messages.

Provider, Products, and Items form a hierarchy. Windows Update first asks the Microsoft server at the Provider level for the supported product categories. It then determines which of the product categories are relevant for the computer that is being updated by evaluating the rules returned by the server. Based on this, Windows Update then asks the server at the Products level for the products contained in each category. It then determines which of the products are relevant for the computer that is being updated, again by evaluating the rules supplied by the server. Finally, Windows Update is able to ask the server at the Items level for details about the available updates for the relevant products.

› Parameter: systemInfo

The systemInfo argument consists of four parts - computerSystem, platform, locale, and devices.

computerSystem - This part is very short and contains only general information, e.g. the make and model of the computer that is being updated or the free space on its harddrives. However, there also is a mysterious PID attribute that consists of 76 seemingly random (and therefore probably encrypted) bytes that are represented by a string of 152 hex digits. We will have an in-depth look at this later.

platform - This part summarizes the basic facts about the operating system, e.g. the processor architecture on which it is running, the build number, or the installed service pack.

locale - Since updates are typically language-specific this part specifies the language of the operating system.

devices - This is by far the longest part. It is only present if GetManifest is run at the DriverUpdates level. In order to be able to determine the driver updates that are relevant for the computer that is being updated the Microsoft server needs to know which hardware components, e.g. which graphics adapter or which sound card, are installed in the computer. This section contains a complete list of all detected hardware devices. In contrast to the hardware hash transmitted to Microsoft during Windows Product Activation this list has to reveal the make and model of each device in order to be of any use. For mass storage devices the IDE ID or SCSI ID is given, which explicitly contains the make and model. This is illustrated by the following excerpt, which shows an IDE ID that represents the SD-C2002 DVD-ROM drive by Toshiba.

```
» <hwid rank="0">
» IDE\CdRomTOSHIBA_DVD-ROM_SD-C2002_____1120____
» </hwid>
```

For plug & play devices the numeric vendor ID and device ID are listed, which can easily be mapped to the make and model, respectively. The following excerpt represents a Cyber 9397 notebook graphics adapter (device ID 9397) by Trident (vendor ID 1023).

```
» <hwid rank="0">
» PCI\VEN_1023&DEV_9397&SUBSYS_00000000&REV_F3
» </hwid>
```

› The PID attribute

With XML being (more or less) easily readable by humans the analysis of the SOAP-based RPC protocol is easy, once it is possible to look inside the SSL connection. It is obvious without any complicated analysis what the transmitted information, e.g. the PNP IDs, is and from where it originates. The only exception is the PID attribute.

As we have suspected before, the 76 bytes are encrypted. The employed algorithm is the extended version of TEA (Tiny Encryption Algorithm). TEA is a symmetric algorithm and it is very popular in the copy protection community. The choice of this algorithm is not surprising as the PID attribute is generated by parts of Windows XP that are also elementary to Windows Product Activation. To decrypt the PID attribute for further analysis, we can use the following C function.

```
» void xtea_decrypt(unsigned char *data, unsigned int *key)
» {
» [...]
» }
»
» void pid_decrypt(unsigned char *pid)
» {
» unsigned int key[4] = { [...] };
» int i;
»
» for (i = 68; i >= 0; i--)
```

```
» xtea_decrypt(pid + i, key);
» }
```

The decrypted PID attribute consists of three parts. The first eight bytes are a timestamp in FILETIME format (see the Microsoft SDK) that captures the generation time of the attribute. Note that due to the way in which TEA is applied for encryption, two PID attributes that differ in their timestamps and that are otherwise identical still result in two completely different encrypted byte sequences. If we run the tecDump utility twice, we therefore observe two seemingly completely different PID attributes, although the attributes differ only in the timestamps.

› The Product-ID in the PID attribute

Then, what do the remaining 68 bytes contain? The last 48 bytes of the PID attribute contain the product ID of the Windows XP installation as a null-terminated Unicode string. Product IDs look like this.

12345-123-1234567-12345

The product ID of Windows XP is accessible in the "Properties" of "My Computer". The ten digits making up the second and third group of digits is extracted from the product key that is entered during the installation of Windows XP. Product keys look like this.

ABCDE-ABCDE-ABCDE-ABCDE-ABCDE

The value represented by the mentioned ten digits is combined with a digital signature. The resulting combination is encoded using 25 letters and digits, which directly leads to the product key. Digital signatures are based on public key cryptography. Microsoft generates the digital signatures using its secret key and Windows XP verifies the signature using the matching public key. If the signature is valid, the product key is genuine. If the algorithm and the parameters of the algorithm are carefully chosen, nobody but Microsoft can generate product keys that contain a valid signature. The threat posed by key generators that are written by software pirates to generate counterfeit product keys is reliably countered because the necessary secret key is only known to Microsoft.

The mentioned ten digits unambiguously identify the product key from which they have been derived. The reason why the PID attribute contains the product ID therefore probably is that it enables the Microsoft server to deny updates to product IDs that match product keys that are spread via the Internet by software pirates.

› Securing the Product-ID

If this were the only information in the PID attribute, software pirates using an illegitimate product key would simply write a tool similar to the tecDump utility, intercept the data transmitted to the Microsoft server, and overwrite the ten digits of the product ID with random digits to hide the fact that they are using a pirated product key. The problem is that it is easy to modify a product ID in a way that the modified product ID is still a valid product ID - but a product ID that identifies a different product key. Luckily, because of its digital signature, it is impossible to modify a product key in a way that the modified product key is still valid. Therefore it is a better idea to include the product key, and not the product ID, in the PID attribute.

But this would enable software pirates again to write a tool similar to the tecDump utility and intercept the transmitted data, i.e. the product key, when it is sent to the Microsoft server. If this tool was added to a Trojan horse program, it would be trivial for software pirates to collect lots of legitimate product keys from the computers of unsuspecting legitimate users.

Moreover using this idea without SSL encryption would also be a problem. It would be trivial for an attacker that eavesdrops between the users' computers and the Microsoft server to spy on the exchanged SOAP messages and extract legitimate product keys of unsuspecting users.

Therefore we need a way to transmit our product ID and prove at the same time that the product ID matches the product key that we know and use. This is achieved by the

remaining 20 bytes of the PID attribute, i.e. the bytes from offset 8 through offset 27. They contain an SHA-1 hash of the 60-byte null-terminated Unicode string representing the product key from which the product ID is derived.

Verification of the PID attribute on the Microsoft server probably resembles these ten steps.

1. decrypt the PID attribute
2. extract the product ID from the PID attribute
3. use the secret key to generate a signature for the value represented by the ten relevant digits of the product ID
4. combine the signature and the value and generate the product key
5. if the product key is blacklisted, deny updating and abort
6. calculate the SHA-1 hash of the generated product key
7. extract the SHA-1 hash from the PID attribute
8. compare the SHA-1 hash from step 6 and the SHA-1 hash of step 7
9. if the hashes do not match, deny updating and abort
10. allow updating

› The COM component

To further explore the capabilities of Windows Update we now have a look at the COM component. The component is implemented by the *iuctl.dll* dynamic link library, which can be found in the System32 folder. The *iuctl.dll* library is a COM wrapper around *iuengine.dll*, which is stored in the same folder.

Using the COM component takes a bit more work than usual, because it expects to be used from within Internet Explorer. Moreover the component can only be used in web-pages that have a URL that begins with *http://v4.windowsupdate.microsoft.com*. To be precise the COM component examines the [IUServerURLs] section in the file *Program Files\ WindowsUpdate\ V4\ iuident.txt*. If the URL of a web-page begins with one of the entries in this section, the component agrees to be used from within this web-page. By default only *http://v4.windowsupdate.microsoft.com* is contained in the section.

We therefore create a bogus WebBrowserApp object to emulate a running instance of Internet Explorer and implement a special LocationUrl() function that always returns *http://v4.windowsupdate.microsoft.com*. Then we make the COM component believe that it is run from within our fake Internet Explorer by calling the SetSite() function of the IObjectWithSite interface of the COM component to set its site to our bogus WebBrowserApp object. When determining the URL of the web-page that it is being used in, the component will call our special LocationUrl() function and believe that it is being used in a web-page with a valid URL.

› Automatic Update for Update

The component DLL also contains a type library that describes the IUpdate interface that we can now work with. Two functions of this interface are important for us:

» *long Initialize(long flag, IUnknown *unk)*
» *BSTR GetSystemSpec(BSTR classes)*

Initialize() must be the first function that we call when using the COM component. The semantics of the flag parameter are not known to us and we set this parameter to zero to imitate the behavior of the Windows Update Java Script code. Initialize() contacts the Microsoft server, checks for a newer version of the component, and automatically updates the component if necessary. The interface parameter must point to the IUnknown interface of an UpdateCompleteListener object, which is also described in the type library. This object seems to implement a callback function that is called when the component has been successfully updated.

If updating iuctl.dll is necessary but fails, Initialize() returns 1. If updating iuengine.dll is necessary but fails, Initialize() returns 2. If both DLLs require an update but updating fails, 3 is returned.

For the sake of simplicity we set the interface parameter to NULL. This seems to have the effect that updating either DLL always fails.

› GetSystemSpec

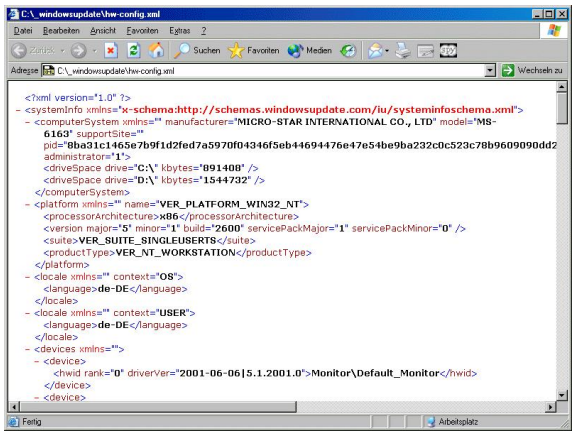
When the component is initialized we can use GetSystemSpec() to obtain an XML string with the information that we already know from the systemInfo argument to GetManifest. The only argument to GetSystemSpec() determines which parts of the systemInfo argument we want to be returned. It is an XML string that contains a combination of any of these five tags - » <*computerSystem* /> « , » <*platform* /> « , » <*locale* /> « , » <*devices* /> « , and » <*regKeys* /> « . We are already familiar with the first four tags, which correspond to the four parts of the systemInfo argument. The » <*regKeys* /> « tag is new. It is supported by the COM component, although it is not currently used by Windows Update.

The described tags must be enclosed by » <*classes*> « and » </*classes*> « , e.g. like

» <*classes*><*computerSystem* /><*platform* /><*locale* /><*devices* /></*classes*>

› The utility control.exe

The tecControl utility can be used to experiment with GetSystemSpec(). Just tick the tags to be passed to it, select an output file for the XML result returned by the function, and open the saved XML file with Internet Explorer to view it. The tecControl utility does not support updates of iuctl.dll or iuengine.dll, because it passes a NULL interface parameter to Initialize(). If an update is required, the utility will display an error message. In this case run Windows Update once to perform the update and run the tecControl utility again.



Your hardware: This is what Windows Update sees about your computer.

As can easily be seen the » <*regKeys* /> « tag causes a list of registry subkeys of HKEY_LOCAL_MACHINE\ SOFTWARE, i.e. a list of the vendors of all software packages installed on the user's computer, to be included in the result.

The tecControl utility is complemented by the tecDecode tool, which extracts the PID attribute from a saved XML file, decrypts it and displays the product ID and the timestamp. If a product key is specified, then its SHA-1 hash is compared to the SHA-1 hash stored in the PID attribute and the tool indicates a match or a mismatch.

› Conclusions

The details that we have documented in this article match the vague information provided by Microsoft. We believe that the biggest privacy issue with Windows Update is the list of hardware components that is transferred to the Microsoft server, which reveals the make and model of all installed PCI cards, mass storage devices, and other hardware

components to Microsoft. The approach that older versions of Windows Update took was to download a complete list of updates and then filter out the relevant ones on the user's computer - without transferring any sensitive information to Microsoft. Why does the current version implement an approach that transfers the information required for the filtering from the user's computer to the Microsoft server, which then does the filtering and returns a list of updates that is tailored to the configuration of the user's computer? Bandwidth is hardly a limiting factor today and downloading a complete list of updates would probably take only a few seconds. This question therefore remains unanswered.

This does not only apply to driver updates. The server-side filtering could also be abused to determine which software is installed. Imagine that Microsoft would like to know whether you use Mozilla 1.0. It would then simply create a product category for Mozilla 1.0, e.g. mo10, add a rule for determining whether Mozilla 1.0 is installed, e.g. Mozilla 1.0 is installed if *» HKEY_LOCAL_MACHINE\SOFTWARE\Mozilla\Mozilla 1.0 «* exists, and return this product category when Windows Update sends a Provider-level request to the Microsoft server. If you were using Mozilla, Windows Update would then by evaluating this rule determine that the product category mo10 applies to your computer, ask the Microsoft server to list the products by sending a Product-level request for mo10, and reveal in this way that you use Mozilla 1.0.

New product categories could also be used for more benign reasons. They make it technically very easy to open Windows Update to other software vendors. As Microsoft is trying to shift to making money with services instead of software, it might try to use the fact that most people who have Windows also have Windows Update as a lever and become the world's premier update service.

The ability of the GetSystemSpec() function of the COM component to list the software vendors of all installed software packages (*» <regKeys /> «* tag) is currently unused by Windows Update, but it might become a privacy issue in the future. Microsoft might be planning to open the Windows Update service to other software vendors, which could be the moment in which Windows Update starts using this feature of GetSystemSpec(). (mha)

› Weitere Themen zu diesem Artikel:

Inside Windows Product Activation (http://www.tecchannel.de/betriebssysteme/742/index.html)