Tracking Malware with Import Hashing | Mandiant

By Mandiant

Published: 2014-01-23 · Archived: 2026-04-05 18:23:02 UTC

Tracking threat groups over time is an important tool to help defenders hunt for evil on networks and conduct effective incident response. Knowing how certain groups operate makes for an efficient investigation and assists in easily identifying threat actor activity.

At Mandiant, we utilize several methods to help identify and correlate threat group activity. A critical piece of our work involves tracking various operational items such as attacker infrastructure and email addresses. In addition, we track the specific backdoors each threat group utilizes - one of the key ways to follow a group's activities over time. For example, some groups may favor the SOGU backdoor, while others use HOMEUNIX.

One unique way that Mandiant tracks specific threat groups' backdoors is to track portable executable (PE) imports. Imports are the functions that a piece of software (in this case, the backdoor) calls from other files (typically various DLLs that provide functionality to the Windows operating system). To track these imports, Mandiant creates a hash based on library/API names and their specific order within the executable. We refer to this convention as an "imphash" (for "import hash"). Because of the way a PE's import table is generated (and therefore how its imphash is calculated), we can use the imphash value to identify related malware samples. We can also use it to search for new, similar samples that the same threat group may have created and used.

Though Mandiant has been leveraging this technique for well over a year internally, we aren't the first to publicly discuss this. An imphash is a powerful way to identify related malware because the value itself should be relatively unique. This is because the compiler's linker generates and builds the Import Address Table (IAT) based on the specific order of functions within the source file. Take the following example source code:

```
#include
                    #include
                    #include
                    #include
                    #pragma comment(lib, "ws2_32.lib")
                    #pragma comment(lib, "wininet.lib")
                    int makeMutexA()
                    {
                        CreateMutexA(NULL, FALSE, "TestMutex");
                        return 0;
                    }
                    int makeMutexW()
                    {
                        CreateMutexW(NULL, FALSE, L"TestMutex");
                        return 0;
                    }
```

```
int makeUserAgent()
{
    HANDLE hInet=0, hConn=0;
    char buf[sizeof(struct hostent)] = {0};
    hInet = InternetOpenA("User-Agent: (Windows; 5.1)", INTERNET_OPEN_TYPE_DIRECT, NULL, NUL
    hConn = InternetConnectA(hInet, "www.google.com", 443, NULL, NULL, INTERNET_SERVICE_HTTP
    WSAAsyncGetHostByName(NULL, 3, "www.yahoo.com", buf, sizeof(struct hostent));
    return 0;
}
int main(int argc, char *argv[])
{
    makeMutexA();
    makeMutexW();
    makeUserAgent();
    return 0;
}
```

When that source file is compiled, the resulting import table looks as follows:

```
 ws2_32.dll

ws2_32.dll.WSAAsyncGetHostByName

wininet.dll

wininet.dll.InternetOpenAwininet.dll.InternetConnectA

kernel32.dll
                kernel32.dll.InterlockedIncrement
                kernel32.dll.IsProcessorFeaturePresent
                kernel32.dll.GetStringTypeW
                kernel32.dll.MultiByteToWideChar
                kernel32.dll.LCMapStringW

kernel32.dll.CreateMutexA kernel32.dll.CreateMutexW

kernel32.dll.GetCommandLineA
            kernel32.dll.HeapSetInformation
            kernel32.dll.TerminateProcess

Imphash: 0c6803c4e922103c4dca5963aad36ddf
```

We abbreviated the table to save space, but the bolded APIs are the ones referenced in the source code. Note the order in which they appear in the table, and compare that to the order in which they appear in the source file.

If an author were to change the order of the functions and/or the order of the API calls in the source code, this would in turn affect the compiled import table. Take the previous example, modified:

```
#include
                    #include
                    #include
                    #include
                    #pragma comment(lib, "ws2_32.lib")
                    #pragma comment(lib, "wininet.lib")
                    int makeMutexW()
                    {
                        CreateMutexW(NULL, FALSE, L"TestMutex");
                        return 0;
                    }
                    int makeMutexA()
                    {
                        CreateMutexA(NULL, FALSE, "TestMutex");
                        return 0;
                    }
                    int makeUserAgent()
                    {
                        HANDLE hInet=0, hConn=0;
                        char buf[sizeof(struct hostent)] = {0};
                        hConn = InternetConnectA(hInet, "www.google.com", 443, NULL, NULL, INTERNET_SERVICE_HTTP
                      hInet = InternetOpenA("User-Agent: (Windows; 5.1)", INTERNET_OPEN_TYPE_DIRECT, NULL, NULL,
                        WSAAsyncGetHostByName(NULL, 3, "www.yahoo.com", buf, sizeof(struct hostent));
                      return 0;
                    }
                    int main(int argc, char *argv[])
                    {
                        makeMutexA();
                      makeMutexW();
                      makeUserAgent();
                      return 0;
                    }
```

In this example, we have reversed the order of makeMutexW and makeMutexA, and of InternetConnectA and InternetOpenA. (Note that this would be an invalid sequence of API calls, but we use it here to illustrate the point.) Below is the import table generated from this modified source code (again abbreviated); note the changes when compared to the original IAT, above, as well as the different imphash value:

```
 ws2_32.dll

ws2_32.dll.WSAAsyncGetHostByName
```

```
wininet.dll

wininet.dll.InternetConnectAwininet.dll.InternetOpenA

kernel32.dll
                    kernel32.dll.InterlockedIncrement
                    kernel32.dll.IsProcessorFeaturePresent
                    kernel32.dll.GetStringTypeW
                    kernel32.dll.MultiByteToWideChar
                    kernel32.dll.LCMapStringW

kernel32.dll.CreateMutexWkernel32.dll.CreateMutexA

kernel32.dll.GetCommandLineA
                    kernel32.dll.HeapSetInformation
                    kernel32.dll.TerminateProcess

Imphash: b8bb385806b89680e13fc0cf24f4431e
```

The final example shows how the ordering of included files at compile time will affect the resulting IAT (and thus the resulting imphash value). We'll expand on our original example by adding files `imphash1.c` and `imphash2.c`, to be included with our original source file `imphash.c`:

```
 -- imphash1.c --
                    int makeNamedPipeA()
                    {
                    HANDLE ph = CreateNamedPipeA("\\.\pipe    est_pipe", PIPE_ACCESS_DUPLEX,
                        PIPE_TYPE_MESSAGE, 1, 128,
                        64, 200, NULL);
                        return 0;
                    }
-- imphash2.c --
                    int makeNamedPipeW()
                    {
                        HANDLE ph2 = CreateNamedPipeW(L"\\.\pipe    est_pipeW", PIPE_ACCESS_DUPLEX,
                            PIPE_TYPE_MESSAGE, 1, 128,
                            64, 200, NULL);
                        return 0;
                    }
-- imphash.c --
                    #include
                    #include
                    #include
                    #include
                    #include "imphash1.h"
                    #include "imphash2.h"
```

```
#pragma comment(lib, "ws2_32.lib")
#pragma comment(lib, "wininet.lib")
int makeMutexW()
{
    CreateMutexW(NULL, FALSE, L"TestMutex");
    return 0;
}
int makeMutexA()
{
    CreateMutexA(NULL, FALSE, "TestMutex");
    return 0;
}
int makeUserAgent()
{
    HANDLE hInet = 0, hConn = 0;
    char buf[sizeof(struct hostent)] = {0};
    hConn = InternetConnectA(hInet, "www.google.com", 443, NULL, NULL, INTERNET_SERVICE_HTTP
    hInet = InternetOpenA("User-Agent: (Windows; 5.1)", INTERNET_OPEN_TYPE_DIRECT, NULL, NUL
    WSAAsyncGetHostByName(NULL, 3, "www.yahoo.com", buf, sizeof(struct hostent));
    return 0;
}
int main(int argc, char *argv[])
{
    makeMutexA();
    makeMutexW();
    makeUserAgent();
    makeNamedPipeA();
    makeNamedPipeW();
    return 0;
}
```

Using the following command to build the EXE:

```
cl imphash.c imphash1.c imphash2.c /W3 /WX /link
```

The resulting IAT is:

```
ws2_32.dll
                ws2_32.dll.WSAAsyncGetHostByName
                wininet.dll
                wininet.dll.InternetConnectA
                wininet.dll.InternetOpenA
                kernel32.dll
                kernel32.dll.TlsFree
                kernel32.dll.IsProcessorFeaturePresent
                kernel32.dll.GetStringTypeW
```

```
                    kernel32.dll.MultiByteToWideChar
                    kernel32.dll.LCMapStringW
                    kernel32.dll.CreateMutexW
                    kernel32.dll.CreateMutexA

kernel32.dll.CreateNamedPipeAkernel32.dll.CreateNamedPipeW



                    kernel32.dll.GetCommandLineA
                    kernel32.dll.HeapSetInformation
                    kernel32.dll.TerminateProcess

Imphash: 9129bdbc18cfd1aba498c94e809567d5
```

Changing the order of includes for `imphash1.h` and `imphash2.h` within the source file `imphash.c` will have no effect on the ordering of the IAT. However, changing the order of the files on the command line and recompiling will affect the IAT; note the re-ordering of `CreateNamedPipeW` and `CreateNamedPipeA`:

```
 cl imphash.c imphash2.c imphash1.c /W3 /WX /link
                    ws2_32.dll
                        ws2_32.dll.WSAAsyncGetHostByName
                    wininet.dll
                        wininet.dll.InternetConnectA
                        wininet.dll.InternetOpenA
                    kernel32.dll
                        kernel32.dll.TlsFree
                        kernel32.dll.IsProcessorFeaturePresent
                        kernel32.dll.GetStringTypeW
                        kernel32.dll.MultiByteToWideChar
                        kernel32.dll.LCMapStringW
                        kernel32.dll.CreateMutexW
                        kernel32.dll.CreateMutexA

kernel32.dll.CreateNamedPipeWkernel32.dll.CreateNamedPipeA



                    kernel32.dll.GetCommandLineA
                        kernel32.dll.HeapSetInformation
                        kernel32.dll.TerminateProcess

Imphash: c259e28326b63577c31ee2c01b25d3fa
```

These examples show that both the ordering of functions within the original source code - as well as the ordering of source files at compile time - will affect the resulting IAT, and therefore the resulting imphash value. Because the source code is not organized the same way, two different binaries with exactly the same imports are highly

likely to have different import hashes. Conversely, if two files have the same imphash value, they have the same IAT, which implies that the files were compiled from the same source code, and in the same manner.

For packed samples, simple tools or utilities (with few imports and, based on their simplicity, likely compiled in the same way), the imphash value may not be unique enough to be useful for attribution. In other words, it may be possible for two different threat actors to independently generate tools with the same imphash based on those factors.

However, for more complex and/or custom tools (like backdoors), where there are a sufficient number of imports present, the imphash should be relatively unique, and can therefore be used to identify code families that are structurally similar. While files with the same imphash are not guaranteed to originate from the same threat group (it's possible, for example, for the files were generated by a common builder that is shared among groups) the files can at least be reasonably assumed to have a common origin and may eventually be attributable to a single threat group with additional corroborating information.

Employing this method has given us great success for verifying attacker backdoors over a period of time and demonstrating relationships between backdoors and their associated threat groups.

Mandiant has submitted a patch that enables the calculation of the imphash value for a given PE to Ero Carrera's pefile (http://code.google.com/p/pefile/).

Example code:

```
import pefile
                    pe = pefile.PE(sys.argv[1])
                    print "Import Hash: %s" % pe.get_imphash()
```

Mandiant uses an imphash convention that requires that the ordinals for a given import be mapped to a specific function. We've added a lookup for a couple of DLLs that export functions commonly looked up by ordinal to pefile.

Mandiant's imphash convention requires the following:

- Resolving ordinals to function names when they appear
- Converting both DLL names and function names to all lowercase
- Removing the file extensions from imported module names
- Building and storing the lowercased string . in an ordered list
- Generating the MD5 hash of the ordered list

This convention is implemented in pefile.py version 1.2.10-139 starting at line 3618.

If imphash values serve as relatively unique identifiers for malware families (and potentially for specific threat groups), won't discussing this technique alert attackers and cause them to change their methods? Attackers would need to modify source code (in a way that did not affect the functionality of the malware itself) or change the file order at compile time (assuming the source code is spread across multiple files). While attackers could write tools to modify the imphash, we don't expect many attackers to care enough to do this.

We believe it is important to add imphash to the lexicon as a way to discuss malware samples at a higher level and to exchange information about attackers and threat groups. For example, incident responders can use imphash values to discuss malware without specifically disclosing which exact sample (specific MD5) is being discussed.

Consider a scenario where an attacker compiles 30 variants of its backdoor with different C2 locations and campaign IDs and deploys them to various companies. If a blog post comes out stating that a specific MD5 was identified as part of a campaign, then based on that MD5 the attacker immediately knows what infrastructure (such as C2 domains or associated IP addresses) is at stake and which campaign may be in jeopardy. However, if the malware was identified just by its imphash value, it is possible that the imphash is shared across all 30 of the attacker's variants. The malware is still identifiable by and can be discussed within the security community, but the attacker doesn't know which specific samples have been identified or which parts of their infrastructure are in jeopardy.

To demonstrate the effectiveness of this analysis method, we've decided to share the imphash values of a few malware families from the Mandiant APT1 report:

Family Name	**Import Hash**	**Total Imports**	**Number of Matched Samples**
GREENCAT	2c26ec4a570a502ed3e8484295581989	74	23
GREENCAT	b722c33458882a1ab65a13e99efe357e	74	18
GREENCAT	2d24325daea16e770eb82fa6774d70f1	113	13
GREENCAT	0d72b49ed68430225595cc1efb43ced9	100	13
STARSYPOUND	959711e93a68941639fd8b7fba3ca28f	62	31
COOKIEBAG	4cec0085b43f40b4743dc218c585f2ec	79	10
NEWSREELS	3b10d6b16f135c366fc8e88cba49bc6c	77	41
NEWSREELS	4f0aca83dfe82b02bbecce448ce8be00	80	10
TABMSGSQL	ee22b62aa3a63b7c17316d219d555891	102	9
WEBC2	a1a42f57ff30983efda08b68fedd3cfc	63	25
WEBC2	7276a74b59de5761801b35c672c9ccb4	52	13

We calculated the above malware families and corresponding imphash values over the set of malware from the Mandiant APT1 report released in February 2013. Using the imphash method described above, we calculated imphash values over all the samples, and then counted the total number of samples that matched on each imphash. Using 356 total samples from the report, we were able to identify 11 imphash values that provided significant coverage of their respective families. Pivoting from these imphash values, we were able to identify additional

malware samples that further analysis showed were part of the same malware families and attributable to the same threat group.

Imphash analysis, like any other method, has its limitations and should not be considered a single point of success. Just because two binaries have the same imphash value does not mean they belong to the same threat group, or even that they are part of the same malware family (though there is an increased likelihood that this is the case). Imphash analysis is a low-cost, efficient and valuable way to triage potential malware samples and expand discovery by identifying "interesting" samples that merit further analysis. The imphash value gives analysts another pivot point when conducting discovery on threat groups and their tools. Employing this method can also yield results in tracking and verifying attacker backdoors over time, and it can assist in exposing relationships between backdoors and threat groups. Happy Hunting!

Posted in

- Threat Intelligence
- Security & Identity
