Host-based Threat Modeling & Indicator Design

By Jared Atkinson

Published: 2017-07-18 · Archived: 2026-04-05 21:31:45 UTC

Introduction and Background

Last week, my colleague Brian Reitz (@brian_psu) wrote a brilliant post about leveraging PSReflect to model malware techniques. This post builds upon his thought process and explicitly lays out SpecterOps' methodology surrounding threat modeling and design of defensive indicators.

Ultimately, this process is designed to facilitate researching a technique from the underlying technology all the way to specific implementations. We start by identifying the offensive technique we want to focus on. Once we have a technique, we identify the actual technology that the technique targets. This step forces us to look at the technique from the macro level, while still maintaining a technical perspective. For the next portion of the process we will build on the knowledge we gained to identify relevant data sets, understand how to collect it, and write a proof-of-concept (POC) for implementing the specific attack technique. With these tools we can then begin to understand the technology's normal behavior and identify how the attack technique deviates from normal as well as validate any assumptions made along the way. Lastly, we will check how the scale of an enterprise environment challenges our assumption and adapt our detection based on this reality.

In this post I will leverage the MITRE ATT&CK Framework to identify post-exploitation techniques that we may want to focus on.

Methodology

Describing the methodology seems to make the most sense if I work through it using an actual example technique. For the remainder of the post, I will explain my thought process on detecting malicious use of NTFS Extended Attributes (ATT&CK: T1096 — Defense Evasion/NTFS Extended Attributes), a technique used by the Zeroaccess Trojan since at least 2012.

Research Underlying Technology

Developing a lasting behavioral detection requires a solid understanding not only of the possible attacker implementations of the technique, but of the underlying technology as well. In this case, how can we truly detect the malicious use of NTFS Extended Attributes (EA) if we don't actually understand them?! A good place to start delving into this technology is the ATT&CK Framework's wiki entry for our chosen technique. If we refer to the technique's wiki we can see the description of the attack and how adversaries leverage this technique to evade in-place defenses:

"Data or executables may be stored in New Technology File System (NTFS) partition metadata instead of directly in files. This may be done to evade some defenses, such as static indicator scanning tools and anti-virus.

The NTFS format has a feature called Extended Attributes (EA), which allows data to be stored as an attribute of a file or folder."

After reading this, I walk away with some level of understanding, but I also have some questions. Let's take a look at what we know after reading these sentences:

- Attackers are hiding data in NTFS metadata structures (namely Extended Attributes).
- Data is being hidden so the attacker can bypass AV and other automated scanning solutions.

Let's take a peek at the questions I've come up with:

- What are Extended Attributes?
- Where are Extended Attributes stored?
- Why are attackers using Extended Attributes?
- Is there any particular advantage to this technique?
- How are the attackers interacting with the Extended Attributes (storing and retrieving data)?

Ok! Lets see what we can do to answer these questions!

First things first, we know that EAs are part of NTFS, so we should start by checking out some NTFS documentation. Technically speaking, NTFS is a proprietary file system, so we cannot simply find information about Extended Attributes on MSDN. Luckily, Richard Russon provides excellent documentation derived from the Linux NTFS driver project. While working on PowerForensics development, this is my first stop when I have an NTFS related question. The page is a gold mine of information about NTFS, including the most niche details about the Master File Table (MFT) and its attributes.

Upon arriving on the page, I searched for "Extended Attribute" and was directed to the $EA page. The "Overview" section describes the $EA attribute as:

"*Used to implement the HPFS extended attribute under NTFS. This file attribute may be non-resident because its stream is likely to grow.*

As defined in $AttrDef, this attribute has a no minimum size but a maximum of 65536 bytes."

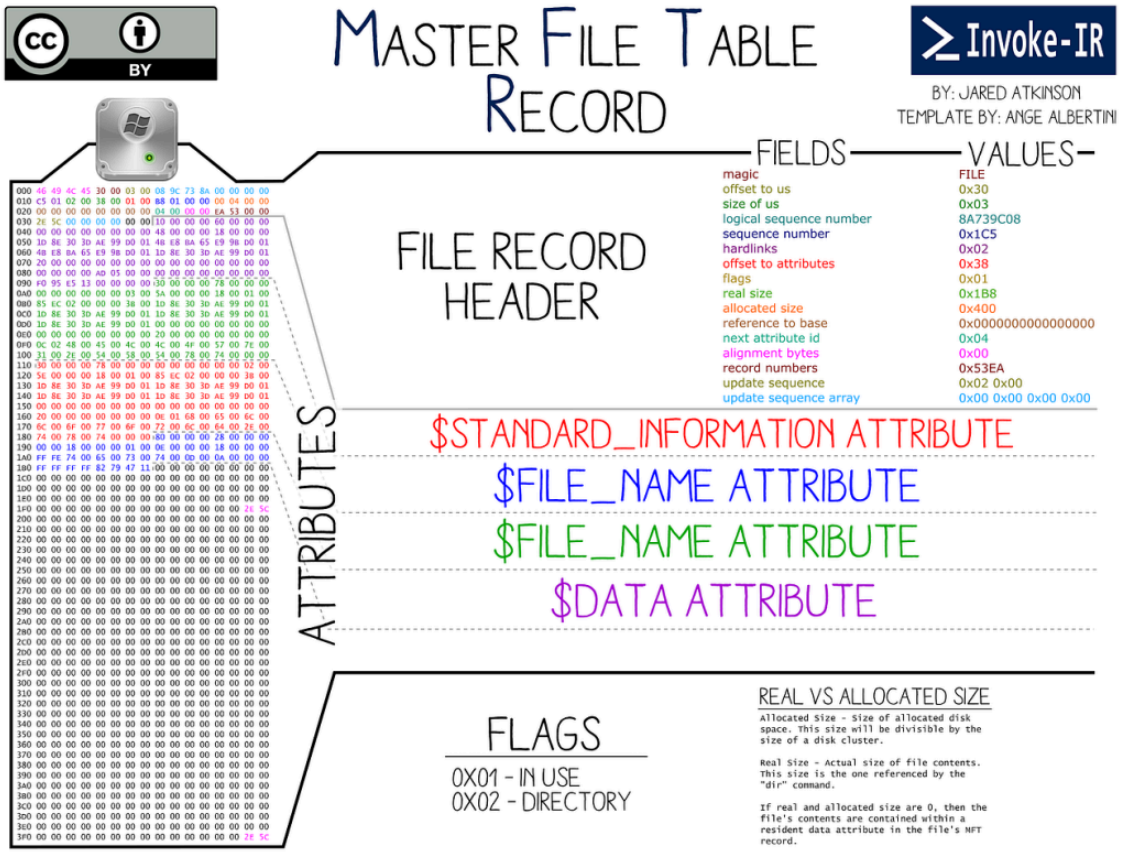
The documentation also provides a the $EA attribute data structure as shown below:

Offset	Size	Description
~	~	Standard Attribute Header
0x00	4	Offset to next Extended Attribute
0x04	1	Flags
0x05	1	Name Length (N)
0x06	2	Value Length (V)
0x08	N	Name
N+0x08	V	Value

$EA attribute structure retrieved from https://flatcap.org/linux-ntfs/ntfs/attributes/ea.html

Turns out Extended Attributes are Master File Table attributes, meaning it serves to provide some sort of metadata about a file. For some context, a few of the more well known attributes are the $STANDARD_INFORMATION attribute (stores the timestamps we see in File Explorer), the $FILE_NAME attribute (keeps track of the file's name), and the $DATA attribute (which contains or points to the file's contents). It also looks like the $EA is used for compatibility with the High Performance File System (HPFS), which is a legacy file system that is unlikely to be used in conjunction with modern operating systems. I also see that a $EA is basically a name/value pair where the Value field can store arbitrary data (possibly code) with a maximum size of ~65536 bytes.

With a generic concept of EA, we can look at the overarching concept of the Master File Table. The MFT is a metadata structure present on every NTFS formatted partition. The MFT maintains a record for every "file" (data files and directories) on the partition and each record contains file data and metadata. For a bit more understanding of how the MFT works, I've included a poster I made to break down the structure of a Master File Table record below:



Notice that an MFT Record is composed of a header that describes the record's place in the MFT itself and an array of "Attributes." As mentioned earlier, there are numerous attribute types and some are more common than others. For instance, every MFT Record has a $STANDARD_INFORMATION attribute and at least one $FILE_NAME attribute, while a data file will also have at least one $DATA attribute (any additional $DATA attributes are what we all know as "Alternate Data Streams"). Other attributes are used in certain situations, including $EA attributes (Extended Attributes), which seem to be used rather infrequently.

With this understanding of Extended Attributes and how they fit into the MFT picture, we can begin looking at how attackers leverage this technique to achieve their objectives.

Understand and Model Technique

In the last section, we learned about NTFS, specifically the Master File Table and its $EA attribute. During our research, one thing stood out to me. You may have noticed that I mentioned Alternate Data Streams (ADS). ADS is a popular place for attackers to hide files, but the proliferation of this technique has led to the popularization of ADS detections. When we looked at the structure of the $EA attribute, it appears to be a good place to store arbitrary data that matches up with the description on the technique's wiki page. It appears we are dealing with a creative alternative to hiding data in ADS and banking on the idea that less tools/people are looking for Extended Attributes being used to store attacker-supplied data.

Our background of the Master File Table, and more specifically Extended Attributes, allows us to address detecting this technique behaviorally. This portion of the process focuses on modeling the technique. This includes

identifying data relating to this technique, collecting said data, and writing a simple POC to confirm your understanding and to test your data collection.

Identify Related Data Set

First things first, we need to identify what data we need to collect. Some situations call for multiple data sets that must be compared to identify the malicious behavior, but in this case it appears that we only need to look for Extended Attributes throughout the file system.

Implement Collection

Before we begin analyzing behavior for detections, we need to understand how to collect relevant data (in this case NTFS Extended Attributes).

As the developer of PowerForensics, my initial thought was to add support for $EA attributes (something I need to do anyway) and then use my MFT parser to identify all $EA attributes on the file system. After considering this method for a bit, I decided I wanted to also provide a more lightweight option that leverages the Windows API to check a file for an Extended Attribute.

Initially, I had no idea how to enumerate Extended Attributes through the Windows API, but the ATT&CK wiki pointed me to a blog post by Symantec researcher Mircea Ciubotariu. Mircea finds that Zeroaccess Trojan used the ZwSetEaFile and ZwQueryEaFile functions to interact with the Extended Attributes. Awesome! We now know the API calls we need to implement to create and enumerate Extended Attributes.

NOTE: Since this is a post about methodology, I am not going to get into details about the code itself as I believe that will distract from the point of this post. I plan on writing a follow up post that will cover the specific implementation of the “Zw*EaFile” family of functions. For now, there will be a bit of hand waving just to get the point across.

Since this section is focusing on collection, let’s start with ZwQueryEaFile. I wrote an abstraction function for ZwQueryEaFile in PowerShell using PSReflect. Below you will find the MSDN C++ function definition for ZwQueryEaFile:

The **ZwQueryEaFile** routine returns information about extended-attribute (EA) values for a file.

Syntax

C++

```
NTSTATUS ZwQueryEaFile(
  _In_     HANDLE           FileHandle,
  _Out_    PIO_STATUS_BLOCK IoStatusBlock,
  _Out_    PVOID            Buffer,
  _In_     ULONG            Length,
  _In_     BOOLEAN          ReturnSingleEntry,
  _In_opt_ PVOID            EaList,
  _In_     ULONG            EaListLength,
  _In_opt_ PULONG           EaIndex,
  _In_     BOOLEAN          RestartScan
);
```

A couple of high level observations: the function appears to require multiple calls to enumerate ALL Extended Attributes (a file can have multiple $EA attributes). Also, ZwQueryEaFile requires a handle to the file you are querying, so we need to find a way to get that file handle. After a bit of Googling, I found that NtOpenFile did the trick; you can see the C++ function definition for NtOpenFile below:

NtOpenFile function

Opens an existing file, device, directory, or volume, and returns a handle for the file object.

This function is equivalent to the **ZwOpenFile** function documented in the Windows Driver Kit (WDK).

Syntax

C++

```
NTSTATUS NtOpenFile(
  _Out_ PHANDLE            FileHandle,
  _In_  ACCESS_MASK        DesiredAccess,
  _In_  POBJECT_ATTRIBUTES ObjectAttributes,
  _Out_ PIO_STATUS_BLOCK   IoStatusBlock,
  _In_  ULONG              ShareAccess,
  _In_  ULONG              OpenOptions
);
```

Now that we presumably have all the code we need to enumerate a single file for Extended Attributes, we can wrap that functionality and recursively check every file on the file system. Our collection mechanism is now complete.

Implement Technique POC or Malware Sample

After implementing ZwQueryEaFile, ZwSetEaFile should be pretty straightforward. Again we write an abstracted function using PSReflect, so we can add custom Extended Attributes to any file of our choosing. This function is also simpler because you are choosing the file to write to instead of looking for every instance of a file that has Extended Attributes. Below is the C++ function definition for ZwSetEaFile:

The **ZwSetEaFile** routine sets extended-attribute (EA) values for a file.

Syntax

C++

```
NTSTATUS ZwSetEaFile(
  _In_  HANDLE           FileHandle,
  _Out_ PIO_STATUS_BLOCK IoStatusBlock,
  _In_  PVOID            Buffer,
  _In_  ULONG            Length
);
```

In some cases, writing a POC could be a too big of an investment. In these situations, I like to find a malware sample that implements this technique and use that sample for testing purposes. Each technique's wiki entry includes references to blog posts or white papers from which the technique's knowledge was derived. These posts are often a great starting point for understanding the technique itself, as well as the way malware used it in the wild.

Each technique wiki page includes examples of malware that leverage it. An example from the NTFS Extended Attribute entry is below:

Examples

- The Regin malware platform uses Extended Attributes to store encrypted executables.[3]
- Some variants of the Zeroaccess Trojan have been known to store data in Extended Attributes.[4]

What if we just get a copy of Regin or Zeroaccess Trojan and test it in a lab environment? If we refer back to the articles analyzing Zeroaccess and its use of Extended Attributes, we can look for a way to identify the specific version of Zeroaccess that implements this technique. When we review Corey Harrell's post, we find the hash for this version of the malware (ee14dcd20b2ee118618e3a28db72cce92ccd9e85dd8410e02479d9d624934b13). We can now use whatever means are available to us to download a copy of this file to use for testing (I found the sample on VirusShare).

Always consider reimplementing the basic functionality involved in the attack technique, as a first priority. This allows threats to be modeled on a generic level without focusing on a specific tool's implementation. For example, in his post Corey Harrell provides the following description of Zeroaccess Trojan's implementation of this technique:

"*A large amount of binary data is also written to the NTFS Extended Attributes of services.exe. … ZeroAccess uses this feature to hide a whole PE file as well as shellcode that loads the PE file. The overwritten subroutine in services.exe reads in all the data from the Extended Attributes and executes it.*"

If we only analyze ZeroAccess Trojan, we may mistakenly conclude that we only need to check services.exe for Extended Attributes which would obviously not solve the problem in the way we'd like.

Detection

With the ability to collect Extended Attributes and implement a POC "malicious EA," we can work on identifying ways to detect malicious use specifically.

The first step in building a detection is identifying normal. How normal are Extended Attributes on your average enterprise computer? Is there a different "normal" depending on the Operating System version? Do these normal instances follow an identifiable pattern? If we can answer these questions, then we can begin to wrap our heads around identifying the malicious use or more generically, deviations from normal.

One thing that is important to remember in this case is that attackers aren't using Extended Attributes for their original purpose. Attackers are using them as data storage (often PE files) which is not consistent with the "key/value pair" concept we saw in documentation. This information can help us look for patterns when considering normal use versus malicious use.

I put together a couple PowerShell functions to recursively enumerate all files for Extended Attributes (details will be discussed in a future post). You can find my code at the end of the post.

First, I ran the function on a stock Windows 7 VM which came back with exactly 0 EAs. This lines up with our original assumption that Extended Attributes are a legacy compatibility feature that is no longer used in modern operating systems. As I mentioned earlier, just because a stock Windows 7 image has no EAs does not mean that this is the default behavior on all OS's. For this reason I decided to test a Windows 10 VM as well.

Before running my Windows 10 test, I wrote an Extended Attribute named **TEST** to a file located at C:\demo\ea.exe. I want to make sure the function detects my custom EA, as well as, any built in EAs.

Next, I ran the function on my Windows 10 VM and found 35751 EAs! Either EAs made a come back, or I'm pwned! To get an idea of what is going on, I used PowerShell's Group-Object cmdlet to group all EAs based on name to see if there is an identifiable pattern.

```
PS C:\WINDOWS\system32> $x.Length
35751

PS C:\WINDOWS\system32> $x | Group-Object Name

Count Name                      Group
----- ----                      -----
    1 TEST                      {@{Name=TEST; Value= ??h e l l o w o r l d ...
21621 $CI.CATALOGHINT           {@{Name=$CI.CATALOGHINT; Value=  K Microsof...
14128 $KERNEL.PURGE.ESBCACHE    {@{Name=$KERNEL.PURGE.ESBCACHE; Value= T   ...
    1 $KERNEL.PURGE.CATALOGHINT {@{Name=$KERNEL.PURGE.CATALOGHINT; Value=   ...
```

35749 out of 35751 EAs are named either **$CI.CATALOGHINT** or **$KERNEL.PURGE.ESBCACHE**. The remaining two EAs are my "TEST" EA and one named **$KERNEL.PURGE.CATALOGHINT** which appears related to the other built in EAs.

Now we want to see if we can understand the purpose of these EAs. I'll start with the most prevalent EA **$CI.CATALOGHINT**. Let's take a look at one:

```
PS C:\WINDOWS\system32> $x[0] | fl


Name      : $CI.CATALOGHINT
Value     :   ? Package_962_for_KB4025342~31bf3856ad364e35~amd64~~10.0.1.13.ca
ValueData : {0, 1, 0, 63...}
FileName  : C:\Windows\bfsvc.exe
```

Upon further review, it appears that all **$CI.CATALOGHINT** EAs have a value that is related to a .cat file or a "digitally-signed catalog file". These are files that are used to verify the digital signature of other files as a collection and alternative to Authenticode signing each individual file.

We can assume that the catalog referenced in the EA is the catalog file that would be used to validate our file's signature (C:\Windows\bfsvc.exe), but it is best not to assume. We use PowerShell's Get-AuthenticodeSignature cmdlet to check the digital signature of C:\Windows\bfsvc.exe and it tells us that the file is indeed "Catalog" signed.

```
PS C:\> Get-AuthenticodeSignature -FilePath C:\Windows\bfsvc.exe | select *


SignerCertificate      : [Subject]
                           CN=Microsoft Windows, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                         [Issuer]
                           CN=Microsoft Windows Production PCA 2011, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                         [Serial Number]
                           33000001066EC325C431C9180E000000000106

                         [Not Before]
                           10/11/2016 10:39:31 PM

                         [Not After]
                           1/11/2018 9:39:31 PM

                         [Thumbprint]
                           AFDD80C4EBF2F61D3943F18BB566D6AA6F6E5033

TimeStamperCertificate :
Status                 : Valid
StatusMessage          : Signature verified.
Path                   : C:\Windows\bfsvc.exe
SignatureType          : Catalog
IsOSBinary             : True
```

Next we check the catalog file's digital signature and it turns out that the catalog file is Authenticode signed (the file itself contains an embedded signature).

One thing to notice is that both files have an identical Thumbprint. This proves that the catalog file referenced in the **$CI.CATALOGHINT** EA is the catalog used to validate the base file's digital signature. Another way to resolve what catalog to use for validating a file is to use sigcheck sysinternals with the **-i** parameter.

When I thought about why this EA is unique to Windows 10, it struck me that $CI likely stands for Code Integrity which is related to Device Guard. This is a security feature included in Windows 10, but unfortunately there is literally no documentation how Device Guard interacts with Extended Attributes. We can now make an educated guess as to the purpose of the **$CI.CATALOGHINT**.

I will leave the other two EAs as an exercise for the reader, but I'd recommend investigating the relationship between **$KERNEL.PURGE.ESBCACHE** and the Enterprise Service Bus. I also stumbled upon this page which describes the **$KERNEL.PURGE** prefix with the statement below:

Auto-Deletion of Kernel Extended Attributes

Simply rescanning a file because the USN ID of the file changed can be expensive as there are many benign reasons a USN update may be posted to the file. To simplify this, an auto delete of Kernel EA's feature was added to NTFS.

Because not all Kernel EA's may want to be deleted in this scenario, an extended EA prefix name is used. If a Kernel EA begins with: `"$Kernel.Purge."` then if any of the following USN reasons are written to the USN journal, NTFS will delete all kernel EAs that exist on that file that conforms to the given naming syntax:

- USN_REASON_DATA_OVERWRITE
- USN_REASON_DATA_EXTEND
- USN_REASON_DATA_TRUNCATION
- USN_REASON_REPARSE_POINT_CHANGE

This delete of Kernel EA's will be successful even in low memory situations.

I'm interested to hear what EAs you come across and how you think they are being used!

Hopefully, this example demonstrated the process I like to follow to understand "normal" behavior on a system.

Test Detection at Scale

It's one thing to get a working detection on one machine in your test lab, but things can surprise you when you attempt to apply that detection at scale. Once we have identified a detection with fairly low false positive rates in the lab, we need to see how that detection works at scale.

In my testing, Extended Attributes seem to remain consistent in large environments. In order to not skip this section, I want to describe how to approach this portion of the methodology from the perspective of the Golden Ticket technique. When I think of detections that work in labs, but needed a little more thought after testing at scale, I think of some of the problems I faced with detections I use for Golden Tickets.

The original assumption was Kerberos tickets (both Ticket Granting Tickets and Service Tickets), in modern environments, should not use RC4 encryption. In our lab environment, this was true and a 100% detection. While an adversary could upgrade to AES encryption, any ticket with RC4 was in fact malicious. We decided to test this at scale and collect tickets across a fairly large environment, and were surprised when we saw TONS of RC4 and therefore, appeared to be TONS of Golden Tickets in the environment (fortunately this was not the case!). After some research and consulting with Sean Metcalf (@PyroTek3) over at Trimarc Security, it turns out that cross-domain and cross-forest tickets use RC4 by default. I assume this is to maintain backward compatibility in cases where the domain being requested is not functional level 2008. Testing this detection at scale can prevent a large amount of effort on trying to address false positives in the future.

This step of the process should also be catered to your environment. For example, many organizations have very few WMI Permanent Event Subscriptions, but maybe someone in your organization has found a useful way to leverage this technology. In most companies, you are unlikely to learn about that just by asking around. Many organizations are too complex for any individual to know everything. The only true way to know how each technology/technique is used in your environment is to look yourself. Every network has its own unique quirks that are sure to surprise you.

Conclusion

So why go through this lengthy process to build detections? A mature threat hunting program continuously maps documented attack techniques (those found in the MITRE ATT&CK Framework) to durable, behavior based detections. Target techniques should be identified based on your resources and you should cover as much of the attack chain as possible. An attacker must evade all detections throughout the chain to be successful. There is an implicit need to fully understand a technology in order to identify detections that are neither too tactical in nature (something an adversary can easily change) nor too broad (with a high false positive rate). Hopefully this methodology can help your organization work toward a more mature threat hunting program.
