Snort My Memory

Snort My Memory is a talk about the concept of “snorting memory.” Snorting memory is a novel concept with application in the incident response realm**.** It involves applying a snort signature to a process’ enumerated strings using Memoryze™[1] or Audit Viewer[2]. Snorting memory combines the signatures provided with Snort™[3] or other communities such as Emerging Threats[4], with MindSniffer™, a new tool being released at Blackhat DC, and the memory analysis/acquisition capabilities of Memoryze. MindSniffer will translate the snort signatures into a format usable by Memoryze or Audit Viewer. The translated Snort signature applied to a process’ enumerated strings, results in the ability to identify suspicious processes in the same way a user would identify suspicious packets on their network.

To understand how snorting memory works there are two concepts that need to be understood: i) accessing physical memory and ii) Snort itself.

Accessing Physical Memory

Snorting memory will not work if access to physical memory is not available. Access to physical memory is necessary for two purposes: i) either to do live analysis or ii) to create a memory image for offline analysis. Windows made available a section object allowing applications to read from physical memory. The section object is \Device\PhysicalMemory[5]. By opening this device and reading from the returned handle, a program can read physical memory. Access to physical memory means Memoryze has access to each process’ virtual address space. Memoryze will translate memory from virtual to physical on its own. The translation is done within each process’ virtual address space, using the process’ *DirectoryTableBase* value in the EPROCESS block. Furthermore, if during the memory translation the memory page being requested is paged to the paging file, Memoryze will parse the paging file to find it and use it.

Accessing physical memory to enumerate process attributes and obtain information about the process has the benefit of bypassing anti-debugger routines since Memoryze is accessing the physical memory, ignoring the virtual address and not attaching to any process. Additionally, accessing physical memory provides the ability to defeat packed files and the debug register rootkits[6]. Most packers will unpack all or some of their sections in memory, giving Memoryze access to more strings to use when snorting memory. Rootkits that trap certain virtual address ranges using the debug registers will fail to prevent Memoryze from reading the memory they are trying to protect.

This type of access to memory allows Memoryze to enumerate strings in a given executable, loaded dlls, heap and stack. The breadth of string coverage is much greater than running strings against a file on disk or doing a process dump using a call to ReadProcessMemory[7].

[1] http://www.mandiant.com/software/memoryze.htm
[2] http://fred.mandiant.com/auditviewer.zip
[3] http://www.snort.org
[4] http://www.emergingthreats.net
[5] http://technet.microsoft.com/en-us/library/cc787565.aspx
[6] http://lists.immunitysec.com/pipermail/dailydave/2008-September/005323.html
[7] http://msdn.microsoft.com/en-us/library/ms680553(VS.85).aspx

One major limitation in accessing physical memory was introduced by Microsoft in Windows 2003 SP1 and beyond. The added restriction requires kernel permissions to open a handle to \Device\PhysicalMemory. To get around this Memoryze installs a driver that opens a handle to \Device\PhysicalMemory and passes that handle back to userland for use.

Understanding Snort

Released in 1998, Snort was one of the first intrusion detection systems (IDS) to come to market; and today is considered one of the most widely deployed IDS. The Snort community is very large and active, responding quickly with signatures for new threats ensuring that a user "snorting memory" can detect the latest threats.

Snort works by comparing all the packets going across a network against a rule set and, if necessary, further inspecting the packets' payload to see if it matches a given signature.

One drawback to using Snort is poorly written signatures. A user must be careful which signatures they select. If the user uses a poorly written signature it can miss a lot of data, or mark too much data as a threat.

Snort identifies network indicators by inspecting network packets in transmission. A process on a host's machine usually generates these network indicators. The strings sent over the network are in memory prior to going out on the network. This means whatever the snort signature matches the packet, that same signature must be in memory for some period (possibly micro seconds) of time. By enumerating all the strings in a process' memory space and applying a given snort signature to the enumerated strings, "snorting memory" can act much like Snort acts on traffic, except in memory.

There is a range of reasons why it is possible to apply snort signatures to enumerated strings. Most executables contain some strings in the .data section of the PE file. These strings can be compiler specific or hopefully malware or sample specific which will potentially lead to a hit with a Snort signature. Another major reason why snorting memory works well is that strings that are dynamically allocated and then freed stay around in memory after being freed. The only way to get rid of a dynamically allocated string is to zero out the buffer prior to freeing it[8]. Even then, if the dynamically allocated string was passed to a Windows API call it is possible an uncontrolled copy of the buffer was created.

There are quite a few benefits to snorting memory. Using the snort signatures the user gets a continually updated and tweaked set of signatures that may in some cases be better or more advanced than anti-virus signatures. Additionally, there are companies that sell signatures.   These signatures, if they follow the snort semantics, could also be applied to memory using MindSniffer™. Memory signatures can identify malware that remain dormant. These samples may be waiting for a specific time to communicate or may be listening for a connection. In this case memory signatures have the potential to identify the samples prior to their using the network.

[8] Tests in the lab showed that after a free the first four bytes of a string were eliminated but the string but the rest of the string remained intact in memory. This may or may not affect the signature.

Malware authors continue to find new ways to hide malcode on victim systems. However, the majority of their techniques have not evolved sufficiently to be robust in the face of many detection techniques – in particular, memory forensic techniques can be quite effective in detection.

Common malware authors think about problems like:

- How can I hide on disk or in registry better?
- How can I obscure my binary on disk better?

An advanced malware author may ask:

- How can I hide my process from process listings?

For most malware the first two questions are the common objectives they try to accomplish. There is a small trend toward malware becoming more sophisticated but this trend is just starting. Even the sophisticated attackers may not be asking questions like:

- How can I obscure strings and clean up in memory better?

Since these authors for the most part are not asking the question about obscuring strings in memory they are doing nothing to hide in memory. There is no need to worry how complex a sample's persistence mechanism is, if it is leaving easily identifiable strings in memory.

Some malware authors are not professional developers and make mistakes like leaving unencrypted strings in the .data section of the binaries. Or, they may be conscious of freeing strings after use[9] but may not zero out the buffer prior to freeing them. In some cases, these left over strings can match snort signatures even after they are freed.

A common attribute of a malware sample is that it is usually packed. Most malware is packed to obscure itself on disk, to decrease it size and to make it harder to statically analyze. Packers used by malware authors will either completely unpack themselves in memory or may unpack most of the binary. Only a few packers make the claim of function level unpacking and even these packers will decrypt more than a function a time. Since these samples are mostly unpacked in memory, and their strings are freely available to enumerate, these mechanisms used to protect malware from detection are useless against snorting memory.

Challenges with Snorting Memory

There are some challenges with snorting memory. The first issue with applying strings to memory is that dynamic strings in the heap or allocated on the stack may not stay around long enough to be identified by a signature. This increases the importance of fast response times. It is possible to see part of a string that would have matched a signature if an incident responder is timely enough in the face of a potential attack.

The second challenge with snorting memory is that MindSniffer discards any signature that is not ASCII. Any binary signatures that cannot be translated to ASCII will have to be discarded. MindSniffer discards

[9] In tests freeing a string after using it resulted in most (in some cases 4 bytes were missing) or all of the string staying intact in memory.

these signatures in part because Memoryze only enumerates alphanumeric characters at this time and if it didn't the result would be examining all of the process address space instead of just strings.

The third problem with snorting memory is the snort signatures themselves. If the signature is too broad it may mark many different processes as suspicious.   If the signature is too narrow it may miss a suspicious process. The signature issue is due to the small range of data malware may send out on the network. If a new variant comes out and just changes the packets it is sending out, the snort signature will miss it.

MindSniffer

MindSniffer is a python utility designed for use in conjunction with Memoryze and/or Audit Viewer. MindSniffer parses a snort signature file and generates either a XML file or a python file. The XML file can be given to Memoryze to run an "audit" that has the snort signature appended to it as an XPath filter. The python file is a plug-in to the Audit Viewer that, if placed in the Signatures directory, will be auto loaded giving the user will have the option to apply the signature. MindSniffer can handle snort "content", and "pcre" keywords. Below are some examples of MindSniffer translating snort signatures:

-
  ```
                      alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS
  (msg:"ET TROJAN Gimmiv.A.dll Infection"; flow: to_server,established;
  uricontent:"/test"; uricontent:".php"; uricontent:"?abc="; uricontent:"?def=";
  reference:url,www.microsoft.com/security/portal/Entry.aspx?name=TrojanSpy%3aWin
  32%2fGimmiv.A; classtype:trojan-activity; sid:2008689; rev:2;)
  ```

-
  ```
                      <value xsi:type="xsd:string">//*[(contains(StringList,
  '/test') and contains(StringList, '.php') and contains(StringList, '?abc=') and
  contains(StringList, '?def='))]</value>
  ```

-
  ```
                      alert tcp $HOME_NET any -> $EXTERNAL_NET 1024: (msg:"ET
  TROJAN Perfect Keylogger FTP Initial Install Log Upload (Null obfuscated)";
  flow:established,to_server;
  content:"C|00|o|00|n|00|g|00|r|00|a|00|t|00|u|00|l|00|a|00|t|00|i|00|o|00|n|00|
  s|00|!|00| |00|P|00|e|00|r|00|f|00|e|00|c|00|t|00|
  |00|K|00|e|00|l|00|o|00|g|00|g|00|e|00|r|00| |00|w|00|a|00|s|00|
  |00|s|00|u|00|c|00|c|00|e|00|s|00|s|00|f|00|u|00|l|00|l|00|y|00|
  |00|i|00|n|00|s|00|t|00|a|00|l|00|l|00|e|00|d|00|"; classtype:trojan-activity;
  sid:2008327; rev:1;)
  ```

-
  ```
                      <value xsi:type="xsd:string">//*[(contains(StringList,
  'Congratulations! Perfect Kelogger was successfully installed'))]
  ```

Memoryze discards any Unicode characters that don't map to ASCII character sets. MindSniffer™ does the same with snort signatures.

Conclusion

As with any newly formed concept or tool, snorting memory and MindSniffer will only be improved as they mature. Such improvements include better parsing of Snort signatures and better generation of those signatures. Research into how long signatures typically stay in memory on an active system, and how long freed strings stay around in memory are going to be vital components to understand how far snorting memory can go.

The future of memory specific signatures is extraordinarily bright. Malware authors must use strings for API calls such as spawning processes, creating and writing files and remote control. Identifying these strings and creating strong memory signatures will be an effective tool in continuing fight against malware.