COUNTERCEPT

MEMORY ANALYSIS

Advanced malware detection
in the enterprise

Contents

Abstract

Traditional malware detection and forensic investigation techniques typically focus on detecting malicious native executables on disk, and performing disk forensics to uncover evidence of historical actions on a system.

In response, many threat actors have shifted their offensive techniques to avoid writing to disk, staying resident only in memory. In particular, Metasploit's Meterpreter payload has adopted this philosophy for more than a decade. Consequently, the ability to effectively analyze live memory for evidence of compromise and to gather additional forensic evidence has become increasingly important.

In this paper, we will look at some of the memory resident techniques used by common malware families and how open-source memory analysis frameworks, such as Volatility, can be used to detect evidence of these techniques on compromised systems. Finally, we will look at how we have adopted similar ideas at Countercept, developing capabilities for performing targeted live memory analysis at scale. This enables us to detect unknown malware, making use of these techniques on isolated systems within large enterprise networks.

Introduction

The first question to ask is why memory analysis is important.

To answer the question, first consider how much forensically interesting data is never written to disk. For example, on a default Microsoft Windows build you are unlikely to see much evidence of historical network connections on disk, aside from a few IP addresses associated with certain Windows event logs. You will also miss evidence of data contained in unsaved documents, any runtime code modifications, malware that has been dynamically loaded and much more besides. On a Microsoft Windows system, you will find that the operating system loads executables from disk into memory and, from that point on, most activity occurs purely in memory, unless explicit actions or logging functionality cause evidence to be written to disk. Put simply, disk forensics only tells you half the story and memory analysis will help to fill in the blanks.

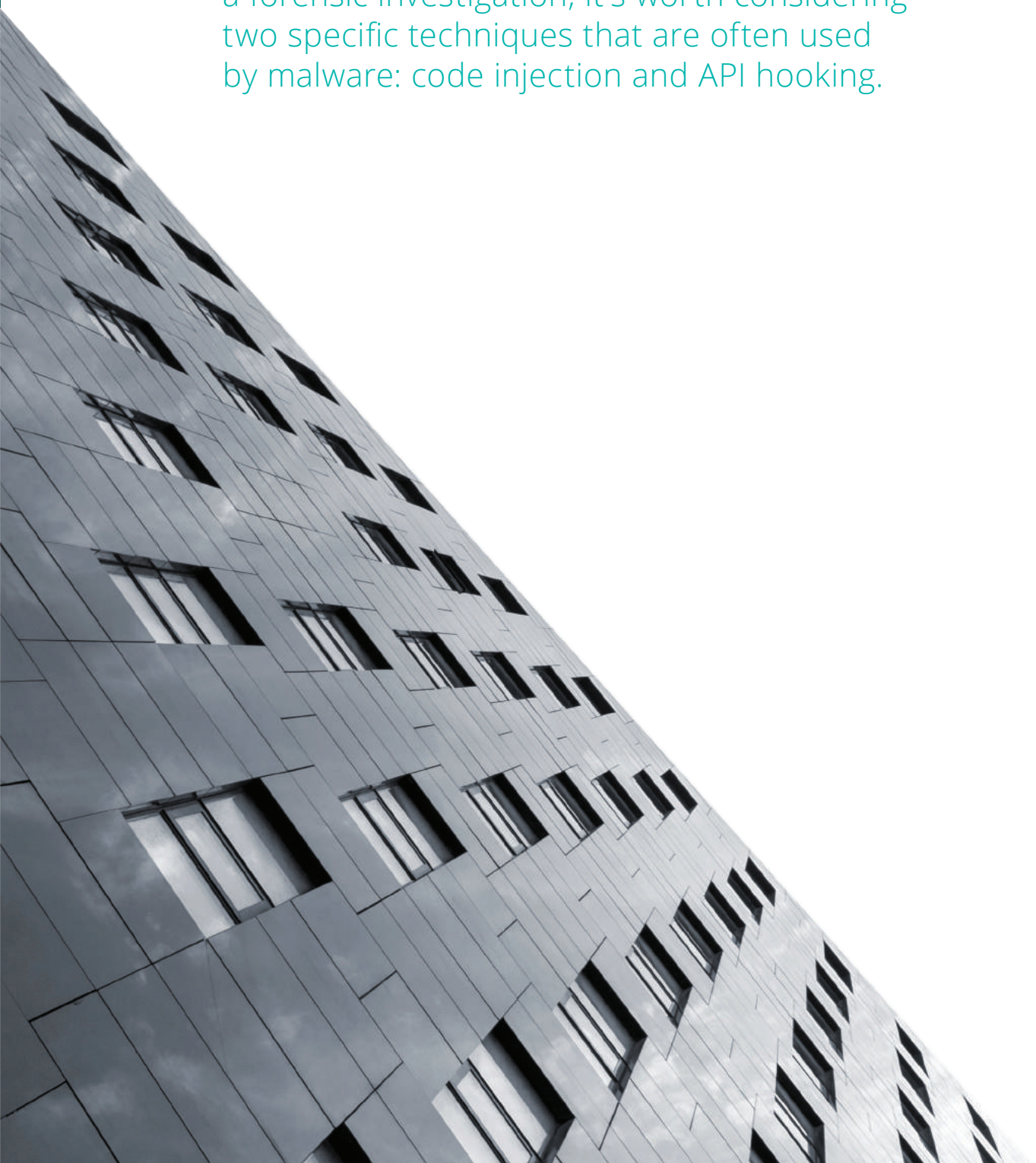
Another reason is that many malware families have moved to using such techniques as code injection and API hooking in a stealthy attempt to avoid traditional security solutions and achieve certain end goals, such as spying on passwords before they are encrypted using TLS. The 2015 Verizon Data Breach Report found that "malware is part of the event chain in virtually every security incident", that "70-90% of malware samples are unique to an organization" and that "organizations would need access to all threat intelligence indicators in order for the information to be helpful – a herculean task". Given these findings, it is obvious that having an effective technique for discovering previously unseen malware on your network is extremely important – and memory analysis can be used to uncover some of the stealth techniques used by modern malware families.

70-90%

of malware samples are unique to an organization.

Concepts

While there are many types of data recoverable through memory analysis that could prove useful in detecting compromises and performing a forensic investigation, it's worth considering two specific techniques that are often used by malware: code injection and API hooking.



Code Injection

Code injection refers to the technique of placing executable code into the address space of another process and causing it to run.

This can be done purely to use the other process as a hosting container to modify the behavior of the process in ways the developers did not intend or to gain access to sensitive data used by that process, such as passwords. There are both legitimate and malicious uses for these techniques.

In this instance, we are not considering exploitation-related code injection vulnerabilities, such as buffer overflows or SQL injection, but rather the post-exploitation activity of placing malicious executable code in the address space of a legitimate process on a compromised system. One reason for this is that it's considered 'stealthier' to ensure no additional processes are running on a compromised system, plus it avoids writing code to disk in order to start a new process. For example, attackers that have compromised a Microsoft Windows system might choose to inject their code into the legitimate 'c:\windows\explorer. exe' process that is running a user's login session, rather than write their code to an executable file on disk and start a new process running that executable file. While a new process is sometimes required to initiate the code injection, this is generally short-lived and can often be achieved using legitimate scripting technologies already installed, such as PowerShell or the Windows Script Host.

Technically, code injection can be implemented purely by the injection of shellcode into another process, which is often the case in traditional memory corruption exploits. However, in practice it is far more common to see a small shellcode stub used as a stager to load a more fully featured implant that has been implemented as a DLL. It is much easier to develop complicated code in a high-level language than to write or compile it entirely as position-independent shellcode. Hence this is a common solution we see used for both legitimate and malicious purposes.

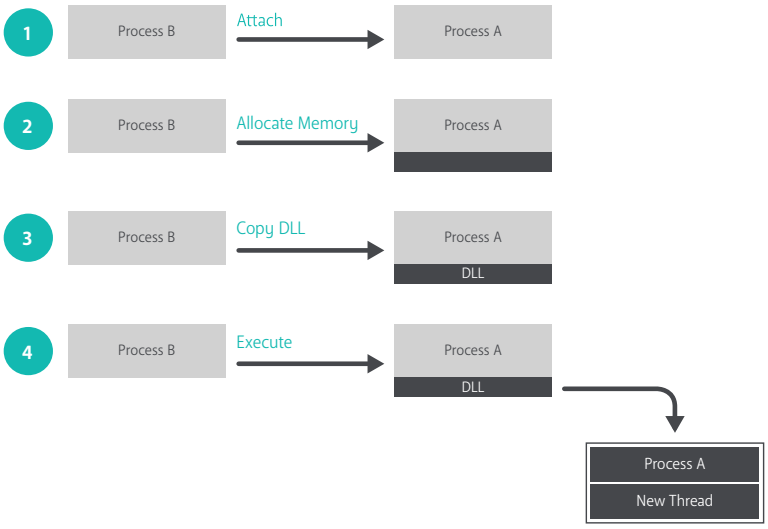
Code injection on Microsoft Windows is often achieved using DLL injection techniques, which generally start by writing some executable code into the address space of another process. Although there are several methods employed that achieve the same end result, probably the most common is using the API function CreateRemoteThread() to start a new thread in the target process to run the malicious code. This code is often no more than a small execution stub that causes the loading of a full DLL with much richer functionality.

Standard DLL Injection

This technique uses functionality offered by the Windows LoadLibrary() API function to map an injected DLL into the target process memory space, and often uses thread injection via CreateRemoteThread() to force the DLL loading. Occasionally, other techniques are used to help hide the fact that the DLL has been loaded, such as unlinking its entry from the list of loaded modules in the process's Process Environment Block (PEB).

Reflective Loading DLL Injection

More stealthy methods utilize reflective loading techniques that employ a custom DLL loader to avoid writing any code to disk, such as the technique used by Metasploit's Meterpreter based on Stephen Fewer's technique. This loader would have to replicate the functionality offered natively by Windows, such as loading any dependencies, resolving indirect function call addresses based on the relocation table, and other procedures necessary to allow correct operation of the DLL. This also helps bypass security products monitoring module loads, as the loader is custom and hence the Windows loader process is not invoked.



Process Hollowing

Another related form of code injection is known as process hollowing. Here, instead of injecting an additional DLL and thread into an existing process to run side-by-side with the legitimate code, a legitimate executable is launched in a suspended state, with the core code section for the process entirely replaced by malicious code before execution is resumed. This approach differs slightly in that it cannot be used to infect existing processes that need to continue operating normally to avoid instability. Consequently, it is not usually a suitable technique for implementing API hooking, which often requires the use of code injection as a first step. However, despite this, Countercept often sees process hollowing and reflective DLL injection used in combination by malware in the field.

A legitimate executable is launched in a suspended state, with the core code section for the process entirely replaced by malicious code before execution is resumed.

API Hooking

API hooking is the process of intercepting function calls in order to monitor and/or change the information passing back and forth between them.

There are many reasons – both legitimate and malicious – why this might be desirable. For example, in the case of data loss prevention (DLP) software, functions related to writing files and sending network traffic might be intercepted to monitor for protected information being written to external storage, or uploaded to personal webmail accounts. However, in the case of malware, this is commonly considered to be 'rootkit' functionality and the most likely reasons are to hide evidence of its presence on the system from other processes and to spy on sensitive data. Some common examples are listed below:

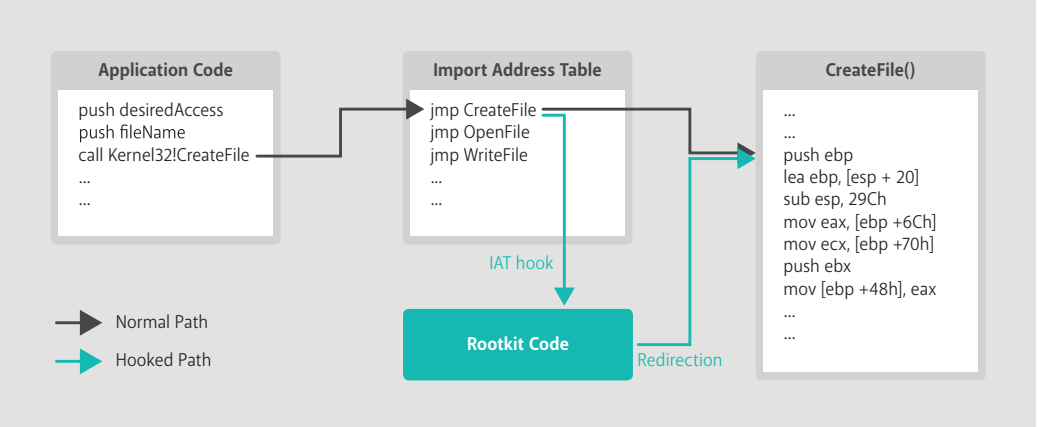
- Intercepting communications with the keyboard in order to log keystrokes, including passwords that are typed by a user;
- Modifying the results returned from file-system or registry-related API calls to remove entries used by the malware to hide its presence from other processes;
- Intercepting data sent to HTTPS-related communication APIs to capture passwords and other sensitive information, prior to them being encrypted using TLS.

There are several hooking techniques, broadly divided into userland and kernel methods. Of the two, kernel hooking is often considered the more stealthy and powerful, but it's arguably more difficult to perform, requires system-level privileges, and can only be used to inspect API calls that result in system calls to the kernel, hence not all malicious use cases can be achieved with kernel hooking in isolation. Kernel hooking is usually implemented using a driver and will often either modify the System Service Descriptor Table (SSDT) to reroute specific system calls to malicious handling code or will use inline hooking to directly modify the kernel code responsible for implementing the system call itself.

Userland hooking is most commonly achieved using either import address table (IAT) hooking, export address table (EAT) hooking, or inline hooking. All these techniques require modification of either code or data structures within the address space of the target process.

IAT/EAT Hooking

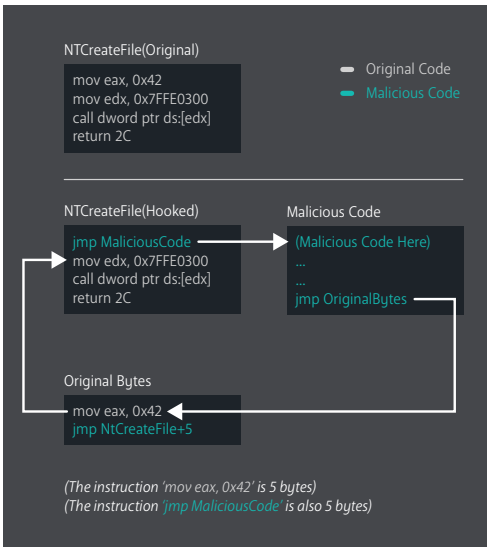
The IAT and EAT are lookup tables holding the start addresses of the functions that implement the APIs used by various processes. Which table is used depends on how the function is called. Modification of the addresses stored in these tables can be used to reroute API calls elsewhere, in a similar manner to SSDT hooking in the kernel.



Inline Hooking

Meanwhile, inline hooking in userland is the direct modification of the code responsible for implementing an API call. Often this will involve the first few instructions being modified to redirect the execution flow, with the hooking code typically returning execution to the legitimate function once the relevant function arguments have been inspected or the return values modified.

Userland hooking can generally be used to monitor a wider set of APIs, plus it can be implemented more robustly, without concerns over changing internals of the Windows kernel, and it doesn't necessarily require administrative privileges on the system. For example, if a low-privilege domain user is compromised, the malware can hook that user's own processes without requiring a privilege escalation exploit. However, if the actor wants to hook processes running as different user accounts on the system, they would first need to find a way to escalate privileges.



Using Volatility to Perform Memory Analysis

Volatility is an open-source framework and the de facto standard toolset for performing memory analysis techniques against raw system memory images, useful in forensic investigations and malware analysis.

Implemented in Python, it has a plugin-based architecture that offers a large number of standard plugins, and is used to acquire a wide variety of forensic information from raw memory images. We make regular use of the Volatility Framework at Countercept to perform in-depth investigation of systems on which we've detected day-to-day compromises. Additionally, our incident response teams make use of it when conducting investigations.

As a short case study, we will look at using two different plugins specifically designed to identify evidence of code injection and API hooking, in this case in a system infected with a variant of the well-known Zeus Trojan. Zeus is a longstanding malware family with powerful backdoor capabilities, including password-stealing techniques, and it makes heavy use of code injection and API hooking.

Zeus Malware Double Threat

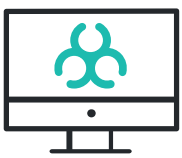
What Zeus does when it infects your PC

Steals logins and passwords saved in your web browser

Records everything you type on your keyboard

Takes regular screenshots

Modifies web pages to steal your personal data



Installs other malicious programs

Tries to infect your Android smartphone

APIhooks Plugin

The first step is to consider the APIhooks plugin. This analyzes all processes on the system, as well as the kernel, to detect evidence of a variety of hooking techniques by spotting abnormal entries in lookup tables such as the SSDT, IAT, EAT, and so on. In addition, it emulates the x86 instruction set to spot function prologues that could have been modified to divert the execution flow. Given below is an example of one userland inline hook detected on the infected system:

```
Hook mode:      Usermode
Hook type:      Inline/Trampoline
Process:        676 (services.exe)
Victim module:   ntdll.dll (0x7c900000 - 0x7c9b0000)
Function:       ntdll.dll!NtCreateThread at 0x7c90d7d2
Hook address:   0x7e3b47
Hooking module: <unknown>

Disassembly(0):
0x7c90d7d2 e97063ed83   JMP 0x7e3b47
0x7c90d7d7 ba0003fe7f   MOV EDX, 0x7ffe0300
0x7c90d7dc ff12         CALL DWORD [EDX]
0x7c90d7de c22000       RET 0x20
0x7c90d7e1 90           NOP
0x7c90d7e2 90           NOP
0x7c90d7e3 90           NOP
0x7c90d7e4 90           NOP
0x7c90d7e5 90           NOP
0x7c90d7e6 90           NOP
0x7c90d7e7 b8           DB 0xb8
0x7c90d7e8 36           DB 0x36
0x7c90d7e9 00           DB 0x0

Disassembly(1):
0x7e3b47 55             PUSH EBP
0x7e3b48 8bec           MOV EBP, ESP
0x7e3b4a 83ec18         SUB ESP, 0x18
0x7e3b4d 53             PUSH EBX
0x7e3b4e 56             PUSH ESI
0x7e3b4f 57             PUSH EDI
0x7e3b50 8b7d14         MOV EDI, [EBP+0x14]
0x7e3b53 8d4514         LEA EAX, [EBP+0x14]
0x7e3b56 50             PUSH EAX
0x7e3b57 6a18           PUSH 0x18
0x7e3b59 8d45e8         LEA EAX, [EBP-0x18]
0x7e3b5c 50             PUSH EAX
0x7e3b5d 33f6           XOR ESI, ESI
```

Detect evidence of a variety of hooking techniques by spotting abnormal entries in lookup tables or modification of function prologues.

From this output we can see the plugin has detected an inline hook of the function 'NtCreateThread()' implemented by 'ntdll.dll' within the 'services.exe' process, with a PID of 676. Looking at the disassembly, it's obvious that this is a classic inline hook, with the very first instruction of 'NtCreateThread()' replaced by a 5-byte JMP instruction to reroute the execution flow to another address (JMP 0x7e3b47). A further disassembly listing gives some of the code found at the location to which the JMP instruction diverts.

In this instance, Volatility has listed the hooking module as '<unknown>'. This is because Volatility has not found the address range to which the hook diverts to be associated with any legitimately loaded module in the process. This is often a suspicious sign as, under most normal circumstances, code should not be executing from memory regions not associated with executable regions of known loaded modules.

Here, there are actually 439 individual hooks identified on the system. Through a little scripting and statistics, we can process this information to give us a slightly higher level view of the hooking to see which types of hooks are present, which types of functions are being hooked, and in which processes.

The screenshot right shows us that a fairly niche set of functions are being hooked across the system, with around 19 instances of each, and with many of the functions being related to HTTP(S) communications. This is a reason for particular concern, as these functions are often used for secured HTTPS connections but the data passed to and from them is unencrypted at this stage. Consequently, these are exactly the types of functions you might expect to see targeted by malware with man-in-the-browser style hooking attacks, with the aim of capturing web passwords.

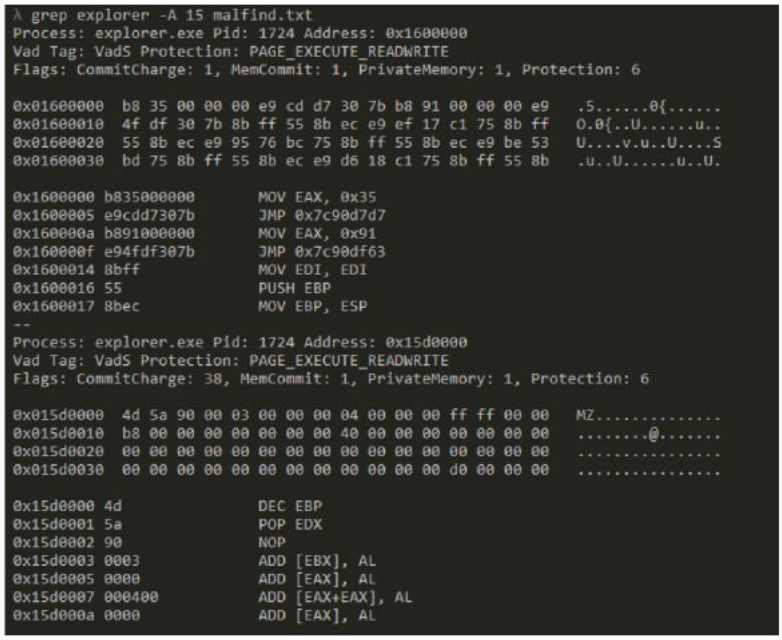
```
λ grep Function apihooks.txt | sort | uniq -c
     19 Function: CRYPT32.dll!PFXImportCertStore at 0x77aef748
     19 Function: USER32.dll!BeginPaint at 0x77d4b4b1
     19 Function: USER32.dll!DefWindowProcA at 0x77d4df6b
     19 Function: USER32.dll!DefWindowProcW at 0x77d4b1e5
     19 Function: USER32.dll!EndPaint at 0x77d4b4c5
     19 Function: USER32.dll!GetClipboardData at 0x77d6fcb2
     19 Function: USER32.dll!TranslateMessage at 0x77d48bce
     19 Function: WININET.dll!HttpQueryInfoA at 0x771c8c6a
     19 Function: WININET.dll!HttpSendRequestA at 0x771c76b8
     19 Function: WININET.dll!HttpSendRequestExA at 0x7721190d
     19 Function: WININET.dll!HttpSendRequestExW at 0x771d53eb
     19 Function: WININET.dll!HttpSendRequestW at 0x77211808
     19 Function: WININET.dll!InternetCloseHandle at 0x771c61dc
     19 Function: WININET.dll!InternetQueryDataAvailable at 0x771d325f
     19 Function: WININET.dll!InternetReadFile at 0x771c9555
     19 Function: WININET.dll!InternetReadFileExA at 0x771f7e9a
     19 Function: WS2_32.dll!WSASend at 0x71ab6233
     19 Function: WS2_32.dll!closesocket at 0x71ab9639
     19 Function: WS2_32.dll!send at 0x71ab428a
     20 Function: ntdll.dll!NtCreateThread at 0x7c90d7d2
     19 Function: ntdll.dll!NtQueryDirectoryFile at 0x7c90df5e
     20 Function: ntdll.dll!ZwCreateThread at 0x7c90d7d2
     19 Function: ntdll.dll!ZwQueryDirectoryFile at 0x7c90df5e
```

By taking one of these functions as an example, in this case 'HttpSendRequestW()', we can see that this particular hook has been implemented across a wide range of processes on the system, including core system services such as 'lsass.exe'. This suggests relatively indiscriminate hooking as it appears to be system-wide, rather than targeted at just one or two processes. Additionally, the targeting of key system services suggests that whatever is responsible for the hooking has system-level privileges. Performing a similar process with the other functions being hooked, it appears that the same set of processes with the same set of functions are being targeted by the hooking right across the system.

```
λ grep -B2 -i HttpSendRequestW apihooks.txt | grep Process | sort | uniq -c
      1 Process: 1028 (svchost.exe)
      1 Process: 1084 (TPAutoConnect.e)
      1 Process: 1088 (svchost.exe)
      1 Process: 1148 (svchost.exe)
      1 Process: 1432 (spoolsv.exe)
      1 Process: 1668 (vmtoolsd.exe)
      1 Process: 1724 (explorer.exe)
      1 Process: 1732 (wuauclt.exe)
      1 Process: 1788 (VMUpgradeHelper)
      1 Process: 1968 (TPAutoConnSvc.e)
      1 Process: 216 (alg.exe)
      1 Process: 432 (VMwareTray.exe)
      1 Process: 452 (VMwareUser.exe)
      1 Process: 468 (wuauclt.exe)
      1 Process: 676 (services.exe)
      1 Process: 688 (lsass.exe)
      1 Process: 844 (vmacthlp.exe)
      1 Process: 888 (wscntfy.exe)
      1 Process: 936 (svchost.exe)
```

Malfind Plugin

Also of use in these circumstances is the 'malfind' plugin. This fairly generic module used for detecting code injection can, in this instance, help to identify the source of the hooking. The plugin scans the memory space for evidence of processes with executable regions that are not associated with known loaded modules, and might be related to code that has been injected into the process at runtime. With this plugin, we find 51 different results across the system. However, given below is one example from the 'explorer. exe' process with PID 1724 that we also saw exhibiting the hooking behavior.



The plugin scans the memory space for evidence of processes with executable regions that are not associated with known loaded modules.

A cursory glance shows the second entry as being of particular interest, as we can immediately see both the classic 'MZ' header associated with a Windows executable and a memory protection state of 'PAGE_EXECUTE_ READWRITE'. When combined, these give a strong indication that an entire Windows executable has been injected into process memory at runtime. If we were then to look back at our hooking results for this process, we could see that there is a relationship between the hooking and this executable module location:

```
**************************************************************************
Hook mode: Usermode
Hook type: Inline/Trampoline
Process: 1724 (explorer.exe)
Victim module: ntdll.dll (0x7c900000 - 0x7c9b0000)
Function: ntdll.dll!NtCreateThread at 0x7c90d7d2
Hook address: 0x15d3b47
Hooking module: <unknown>

Disassembly(0):
0x7c90d7d2 e97063cc84       JMP 0x15d3b47
0x7c90d7d7 ba0003fe7f       MOV EDX, 0x7ffe0300
0x7c90d7dc ff12             CALL DWORD [EDX]
0x7c90d7de c22000           RET 0x20
0x7c90d7e1 90               NOP
0x7c90d7e2 90               NOP
0x7c90d7e3 90               NOP
0x7c90d7e4 90               NOP
0x7c90d7e5 90               NOP
0x7c90d7e6 90               NOP
0x7c90d7e7 b8               DB 0xb8
0x7c90d7e8 36               DB 0x36
0x7c90d7e9 00               DB 0x0

Disassembly(1):
0x15d3b47 55              PUSH EBP
0x15d3b48 8bec            MOV EBP, ESP
0x15d3b4a 83ec18          SUB ESP, 0x18
0x15d3b4d 53              PUSH EBX
0x15d3b4e 56              PUSH ESI
0x15d3b4f 57              PUSH EDI
0x15d3b50 8b7d14          MOV EDI, [EBP+0x14]
0x15d3b53 8d4514          LEA EAX, [EBP+0x14]
0x15d3b56 50              PUSH EAX
0x15d3b57 6a18            PUSH 0x18
0x15d3b59 8d45e8          LEA EAX, [EBP-0x18]
0x15d3b5c 50              PUSH EAX
0x15d3b5d 33f6            XOR ESI, ESI

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

> The potentially injected executable region we saw was located at 0x15d0000, while the code implementing one of the hooks within the same explorer process above is at 0x15d3b47.
> If we were to further analyze the size of memory regions in the process, we'd see that this second location is well within the region size of the memory allocated at 0x15d0000.
> This suggests that whatever executable module is located there, it's also responsible for the hooking. Furthermore, if we looked at all the other results from the Malfind and APIhooks Plugins, we'd start to spot a common relationship among the hooking across all processes we earlier identified. If we were to extract any of these memory regions and perform further analysis, we'd find that we had identified our instance of injected Zeus.

Problems With Memory Analysis

While memory analysis can be an incredibly powerful and useful technique, it's not without its own set of challenges. A few key hurdles are considered below:

Memory Acquisition

Obtaining an intact raw memory image from a live system is not normally standard functionality offered by an operating system. Since technology has progressed and memory grown ever larger, acquiring a complete image without corruption – due to changes while it is being copied – is not trivial.

Labor Intensive

Memory analysis is a highly skilled and time-intensive technique typically performed on one image at a time. When performing a dedicated investigation of a serious compromise, where the systems involved are known and relatively small in number, this is workable. However, using these techniques at scale to detect compromises on a large enterprise network in the absence of other evidence is challenging, to say the least.

Legitimate 'Bad' Behavior

We referred earlier to code injection and hooking techniques being used by malware for malicious purposes. However, in the real world there are plenty of examples of these techniques used for legitimate purposes. In particular, technologies such as data loss prevention and antivirus often target the same functions for hooking as does malware – and ever since the advent of application sandboxes, even the average internet browser hooks itself in order to implement its sandbox. Without the techniques and experience to quickly separate legitimate injection and hooking from malicious behavior, a great deal of time can be wasted.

Using Memory Analysis at Scale

At Countercept, we focus on three core data sources in order to perform effective attack detection and response at scale. These are network monitoring, log aggregation, and endpoint detection and response (EDR). When we started out, there were various excellent technologies available to build network monitoring and log aggregation capabilities, but there were no endpoint solutions that provided the functionality we wanted – so we set out to develop our own EDR software.

Many of the features we have implemented over the years, while powerful, are fairly simple. Yet we loved the idea of developing certain specific memory-analysis capabilities that could be performed continuously, in parallel, across an enterprise network. This would realistically allow us to detect an unknown, purely memory-resident compromise without any prior knowledge or signatures – however, we realized it would be a major challenge.

In order to achieve our aims, we kept three core goals in mind:

- **Scale and Automation**
  With Volatility, we might spend hours or days analyzing one system. We wanted a way to perform a targeted review of an entire network in minutes or hours.
- **Fast Analysis**
  Extracting and analyzing a memory image can be CPU-intensive. We needed a way for our analysis to be performed very quickly on a system, with minimal impact. We also required results to be reported centrally, rather than attempting to stream thousands of memory images to a central location for highly intensive batch processing.
- **Reliability and Safety**
  Raw memory image acquisition is not 100% reliable: in some circumstances it can produce corrupt data – or even cause instability. As the process would be operating across an entire network, we needed something that would be more reliable and completely safe.

In many cases, much of the information extracted using Volatility was data that we could acquire via various analysis and tracing capabilities in our EDR software. Additionally, we could assign a lower priority to any capabilities that would only be required once we were reasonably sure a system had been compromised. In those cases, we would be performing a full incident response exercise anyway – and probably acquiring a full memory image via our EDR software – so in such instances we might as well continue simply to utilize Volatility in the usual way.

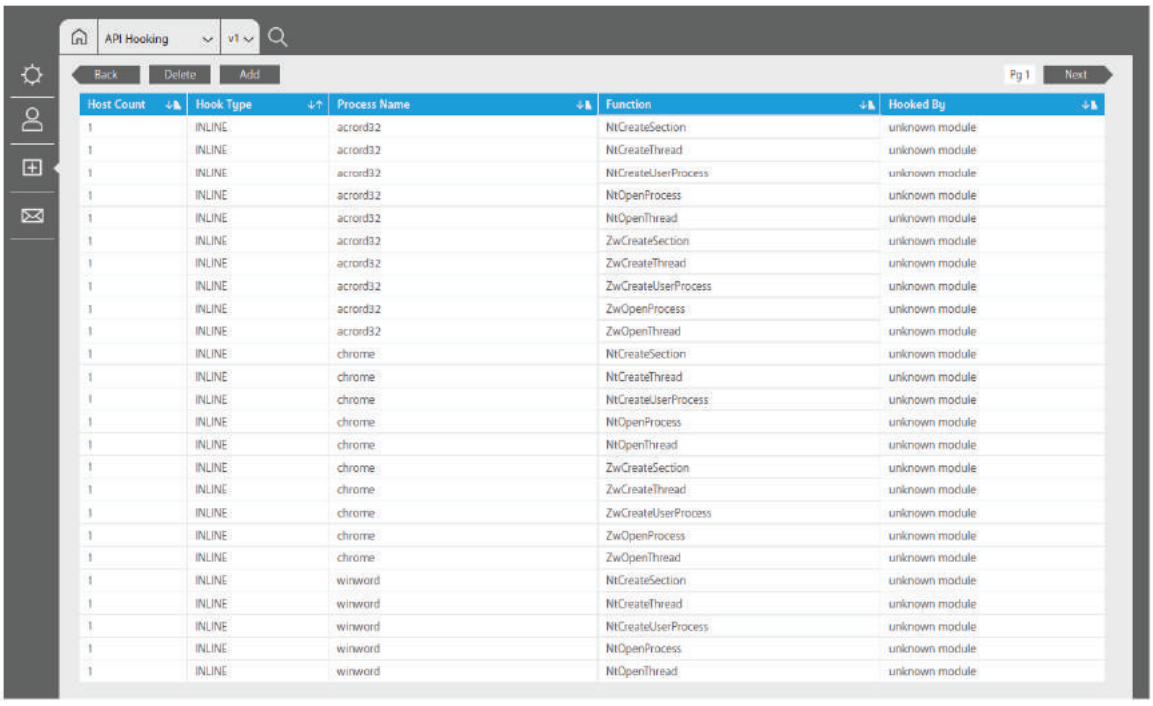
The key capabilities that remained on our priority list were methods to detect code injection and API hooking. These were the offensive techniques for which we lacked good methods for detecting at scale – and we believed it was vital to be able to do so. With this in mind, we set out to develop our own memory analysis capabilities to complement other attack detection methodologies. To achieve this, we took a slightly different approach to memory analysis:

- **Per-Process Analysis**
  Our experience in the field suggested that most of the code injection and API hooking evidence of interest was process-specific. This enabled us to perform per-process address space memory analysis in a far more reliable manner and still get the desired results.
- **Hybrid Disk/Memory Analysis**
  There are some techniques that might not be obvious from looking at memory alone. Often, it's the discrepancy between the code on disk and the code in memory that highlights suspicious activity, and we aimed to exploit the fact that we had access to both data sources simultaneously.
- **WOW64 Emulation**
  WOW64 is the subsystem on Microsoft Windows that allows 32-bit applications to run on 64-bit systems. In the past, we had encountered difficulties in that Volatility was unable to analyze WOW64 processes effectively enough. As we increasingly saw our clients adopt a 64-bit OS as standard, while malware was more commonly 32-bit, we needed to ensure this use case was appropriately covered.
- **Data Correlation**
  One of the bigger time sinks we historically found with memory analysis was in correlating findings between multiple data sources, and attempting to understand whether a particular finding was 'normal' or 'anomalous' with regard to the rest of the network. We needed a solution that allowed very rapid auditing of the findings, while applying anomaly detection techniques to swiftly rule out false positives caused by existing software – and more easily identify the anomalous findings related to genuine compromises.

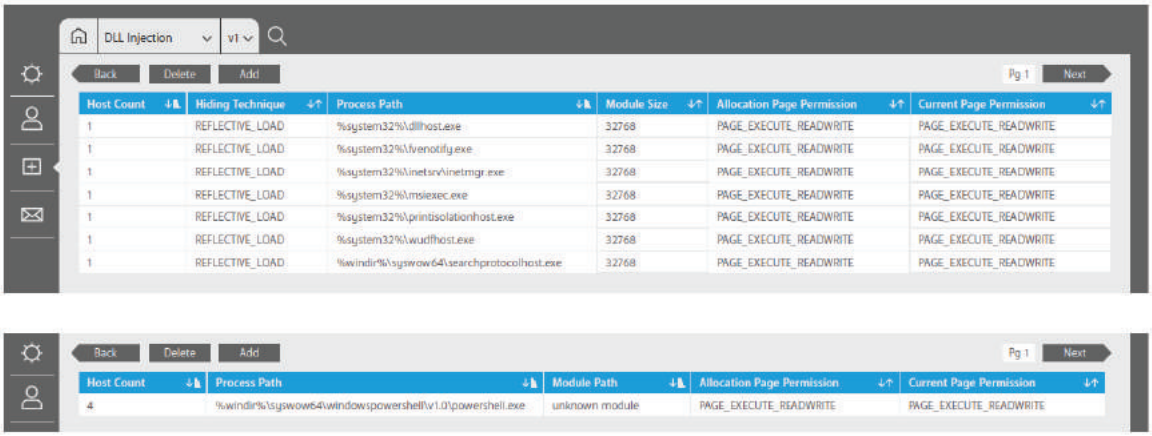
We can see some results in the examples below:

Our EDR software continuously reports results centrally, and here we see one particular version of Chrome hooking itself across 412 systems. The hooking was tracked back to the Chrome main executable module, despite being a WOW64 process, and that module has then been automatically enriched with data to show it was both legitimately signed and well known on VirusTotal. Seeing this volume of aggregated data makes it very quick and easy to rule out probable false positives across a large network. Without aggregation or other supplementary information, it would be a time-intensive matter of individually assessing hundreds of thousands of results from the various systems.

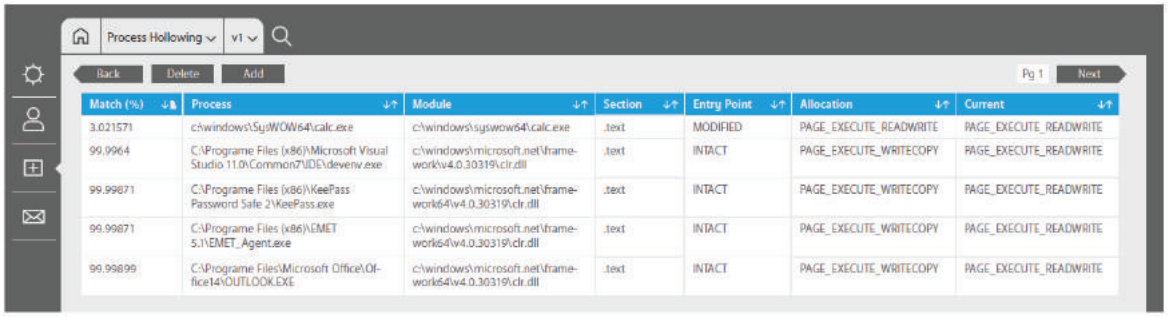
In contrast, consider the following hooking results. Here we see a large number of hooks of sensitive functions by an unknown module across several processes – and all affecting only one host on a particular network. This highly suspicious and anomalous finding would certainly warrant a closer look, and might ultimately lead to the extraction of a full raw memory image for detailed offline investigation.



Similarly, the anomalous results below suggest the possibility of reflectively loaded DLLs in some key processes, and would again be worthy of further analysis. In addition, the four systems showing potentially injected threads within 32-bit instances of PowerShell were caused by security testing exercises run by Metasploit-wielding penetration testers:



Furthermore, good correlation between disk and memory can help to show direct evidence of techniques such as process hollowing. In this instance we can see that the 'calc.exe' in memory bears little resemblance to what is on disk, while its entry point has also changed and the memory permissions associated with the module don't seem to reflect what would normally be expected. Meanwhile, the other results show only very small changes with normal allocation memory permissions but modified current memory permissions. Further investigation would be required to ascertain whether these findings identify malicious activity, but this behavior is often the result of small changes due to inline hooking that show up for auditing in other result sets.



Time-Efficient Analysis

One of the biggest wins for this approach is the time efficiency of the resulting human analysis of the data.

Picking one medium-sized network that we monitor at random, with several thousand endpoints, and then selecting one of the higher-load memory analysis result sets (namely all combinations of userland hooking), we'd see the following number of results over a seven-day window:

Raw number of hooks observed: 1,766,744

Even if we managed to obtain these raw results via Volatility across the entire network, analyzing the results would take an inordinate amount of time – and it's worth remembering that this raw number has already been reduced by targeting the types of hooks and processes that most interest us. However, when the data is continuously aggregated with more filters applied, it can be reduced to the following:

Aggregated: 383 unique hook types

By the time this information reaches our threat hunting team, it has been further reduced, complete with anomaly data, fully resolved paths for the modules, supplementary memory permission data and digital signatures, hash data and VirusTotal information for any linked modules. It then becomes a quick and simple process to identify any interesting results – and we're talking about an entire seven-day rolling window of data for a network with thousands of endpoints.

We find these data sets and analysis techniques extremely valuable as another tool in our scalable attack-detection capability set. When combined with other sets, it also becomes easier to triage systems to determine those that require further investigation. Ultimately, we often end up remotely acquiring raw disk and memory images for deeper analysis, but it's these techniques that allow us to get to that point. In some instances, these memory analysis capabilities have been the primary indicators that first tipped us off about a compromise. This includes compromises detected during our regular benchmarking exercises performed collaboratively with Countercept's parent company MWR's targeted attack simulation (TAS) teams, which often make use of code injection techniques and legitimate Windows functionality to bypass traditional security solutions.

Conclusion

Memory analysis is an extremely powerful technique for forensic investigation and analyzing malware:

Many malware families and attack groups have made increasing use of legitimate scripting technologies and code-injection techniques to stay memory-resident, with the aim of evading traditional security solutions and disk forensics. Consequently, memory analysis has become an increasingly important capability for threat hunting teams.

There are some highly effective open-source toolsets available, such as Volatility, for performing detailed analysis of raw memory images acquired from compromised systems. However, to help our own threat hunting team achieve its goal of detecting advanced attacks against enterprise networks without prior knowledge or signatures, we have additionally developed our own custom capabilities.

We look forward to seeing the security community further progress this important area of attack detection, to help raise the bar for threat actors seeking to compromise large organizations.

COUNTERCEPT

info@countercept.com
www.countercept.com

Connect with us