© SANS Institute 2024
https://linktr.ee/offsecexam

670.3

Operational Actions

Tamarisk OffsecExam Exam solution

© SANS Institute 2024
© 2024 Jonathan Reiter. All rights reserved to Jonathan Reiter and/or SANS Institute.
https://linktr.ee/offsecexam

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

© SANS Institute 2024

https://linktr.ee/offsecexam

SEC670.3 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control

SANS

Operational Actions

© 2024 Jonathan Reiter | All Rights Reserved | Version J01_03

Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control: 670.3
Welcome to Section 3 of SEC670. This section is all about what you could do locally against the target. Let's get started.

Tamarisk OffsecExam Exam Solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Table of Contents (1)	Page
PE Format	4
Lab 3.1: GetFunctionAddress	26
Threads	33
Injections	49
Lab 3.2: ClassicDLLInjection	55
Lab 3.3: APC Inception	60
Lab 3.4: ThreadHijacker	66
Escalations	81
Lab 3.5: TokenThief	100
Bootcamp	138
Lab 3.6: So, You Think You Can Type	140

This page intentionally left blank.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Table of Contents (2)	Page
Lab 3.7: UACBypass-Research	141
Lab 3.8: ShadowCraft	142

Tamarisk OffSecExam Exam solution

This page intentionally left blank.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- **Operational Actions**
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 3

PE Format
Lab 3.1: GetFunctionAddress
Threads
Injections
Lab 3.2: ClassicDLLInjection
Lab 3.3: APCInjection
Lab 3.4: ThreadHijacker
Escalations
Lab 3.5: TokenThief
Bootcamp
Lab 3.6: So, You Think You Can Type
Lab 3.7: UACBypass-Research
Lab 3.8: ShadowCraft

PE Format
In this module, we will discuss in detail the format of PE files. Knowing the structure of various headers is vital for carrying out a few injection methods, as well as retrieving other information to leverage like walking exported functions.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Objectives

Our objectives for this module are:

- Tear into the PE format
- Use winnt.h as a guide
- Build a lightweight PE parser

Objectives
The objectives for this module are to tear into the anatomy of a PE file using the winnt header file as a guide. At the end, you will be armed with enough knowledge that you should be able to build a lightweight PE file parser.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

PE Format

Portable executable: the format is architecture agnostic

PE files, or executables, are complex in design due to the many structures that are involved under the hood.

Executable images (PE)

Object files (COFF)

PE Format

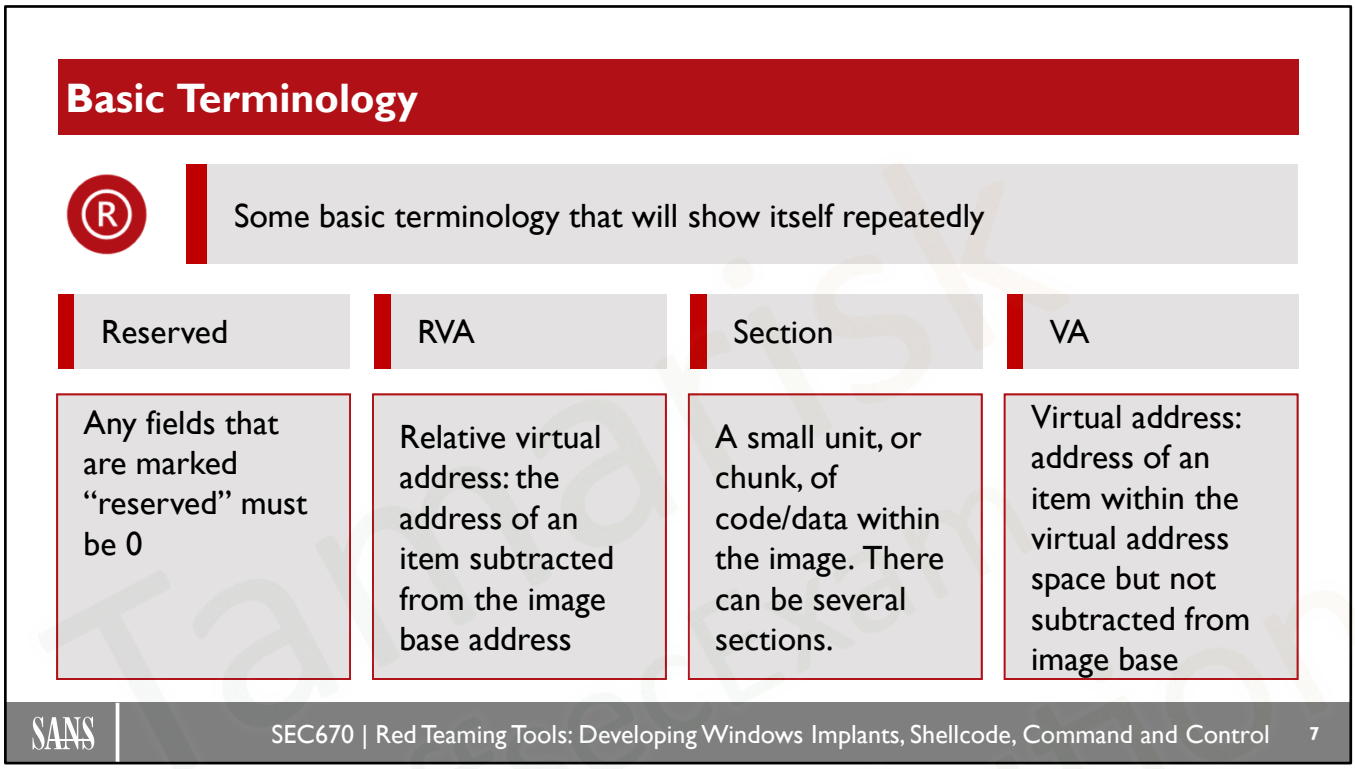
The internal structure of a PE file is very complex, even if you are using the MSDN documentation as a guide to go through it. There are several structures that you will need to understand and most, if not all of them are defined in the winnt.h header file. There is some very important information that is held in some of the struct fields, like what type of application it is (GUI or console), what architecture is it supposed to execute on, what functions are being imported, if any, and what functions are being exported, if any. When an image is in the process of being loaded into memory by the system loader, the system loader will parse every inch of the PE format to ensure the image is properly loaded into memory.

According to the MSDN page for PE Format, it will take you 127 minutes to read everything they have documented. To truly grasp everything documented and implement a highly detailed parser, it could perhaps take a week if it were your full-time job. For this reason, we will only be creating a lightweight parser for the lab at the end of this module.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



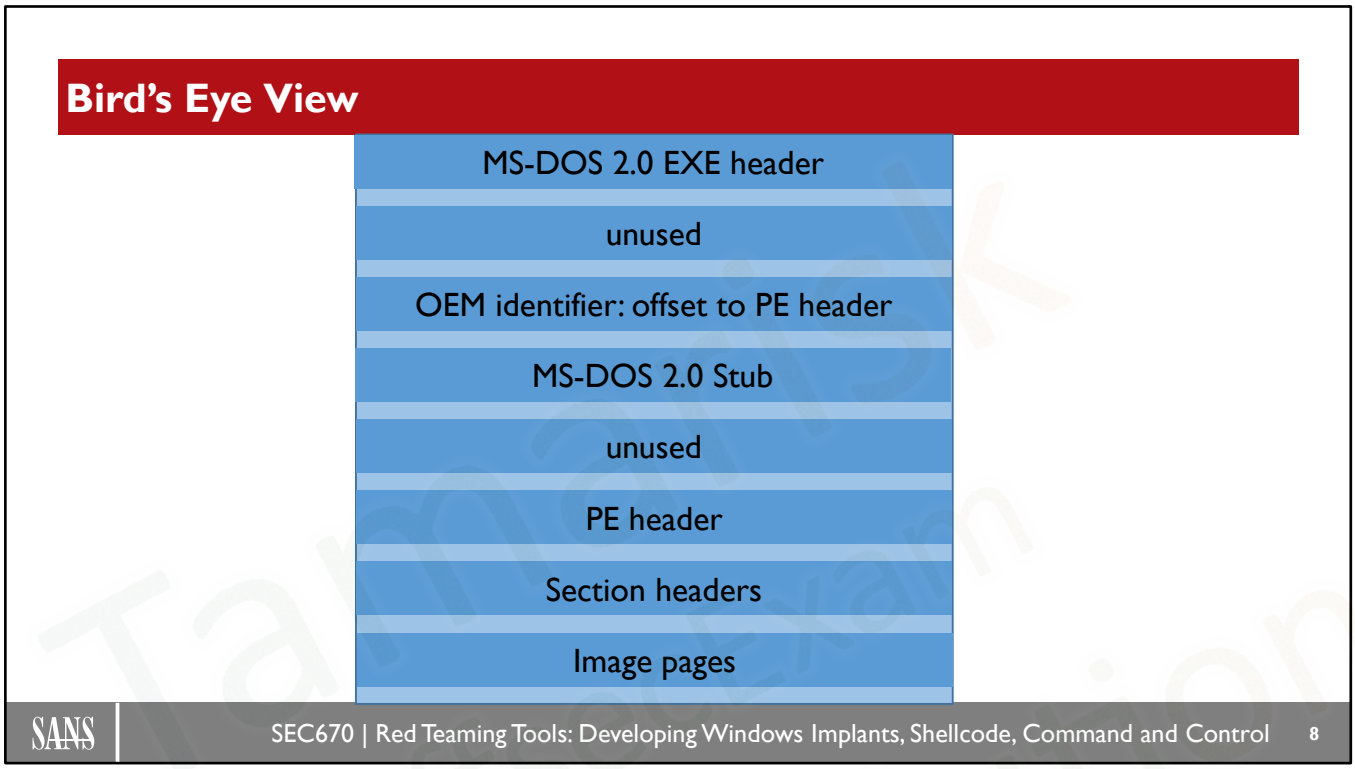
Basic Terminology

Whether you are browsing MSDN pages or blog posts related to the PE format, there are several terms that will show up repeatedly, so it is good to know them by heart. First up is Reserved. There are many structures that Windows will refer to as opaque, meaning they will not provide the full definition of a structure. Because of this, there might be several fields in a structure that are labeled Reserved. It is best to leave those alone unless you are absolutely certain you know what you are doing. Next, is RVA, which is the acronym for relative virtual address. This one should not be new to you because it was discussed in detail during Section 1 when we covered DLLs. However, it is relevant here as well, so it is provided for completeness. Next is Section, or a small segment of code or data within the image. Some of you might already be familiar with some of the section names like TEXT, DATA, BSS, PDATA, etc. In theory, there is no limit to the number of sections that a PE can have. The sections can be named anything a developer would like as the system loader could care less about section names. If you are in the field for long enough, you will come across this at some point. The last one worth mentioning is VA, or virtual address. The Virtual address is the address as it looks in memory, contained within the confines of the image's (now process) virtual address space.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Bird's Eye View

If you were to look at the PE format from a bird's eye view, this is what it might look like. This is straight from the MSDN page for the PE format but represented in a graphic to make it more digestible. The first item is the infamous EXE header that is marked with "MZ", which are the initials for the person who was a primary developer for MS-DOS, Mark Zbikowski. The more important item in that header is how to get to the PE header. The other major players here are the DOS stub, and PE header, and the next several slides will go into each one of these sections and break them down in more detail.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

MS-DOS 2.0 EXE Header

```
typedef struct _IMAGE_DOS_HEADER {
WORD e_magic;           // 00: MZ Header signature
WORD e_cblp;            // 02: Bytes on last page of file
WORD e_cp;              // 04: Pages in file
WORD e_crlc;            // 06: Relocations
WORD e_cparhdr;         // 08: Size of header in paragraphs
WORD e_minalloc;        // 0a: Minimum extra paragraphs needed
WORD e_maxalloc;        // 0c: Maximum extra paragraphs needed
WORD e_ss;              // 0e: Initial (relative) Stack Segment value
WORD e_sp;              // 10: Initial Stack Pointer value
WORD e_csum;            // 12: Checksum
WORD e_ip;              // 14: Initial Instruction Pointer value
WORD e_cs;              // 16: Initial (relative) Code Segment value
WORD e_lfarlc;          // 18: File address for the relocation table
WORD e_ovno;            // 1a: Overlay number
WORD e_res[4];          // 1c: Reserved
WORD e_oemid;           // 24: OEM identifier
WORD e_oeminfo;         // 26: OEM information
WORD e_res2[10];        // 28: Reserved
DWORD e_lfanew;         // 3c: Offset to extended header AKA PE Signature
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```


MS-DOS 2.0 EXE Header

This is the structure of the DOS header as defined in the winnt.h header file. The comments include the field offsets for the various members and a short description of what each field is for. The main areas of interest for our purposes are the first and last fields: e_magic and e_lfanew. The e_magic is a 2-byte field that holds the magic signature of PE files: MZ. Typically, when you would see this in memory, you could assume that it is part of an executable. The other field of importance found at offset 0x3C is e_lfanew, which is rumored to stand for the Long File Address for the New Executable header. This new executable header is what we are after because it sends us to the NT headers.

The winnt.h header file has a nice definition for the DOS signature: *#define IMAGE_DOS_SIGNATURE 0x5A4D /* MZ */.*

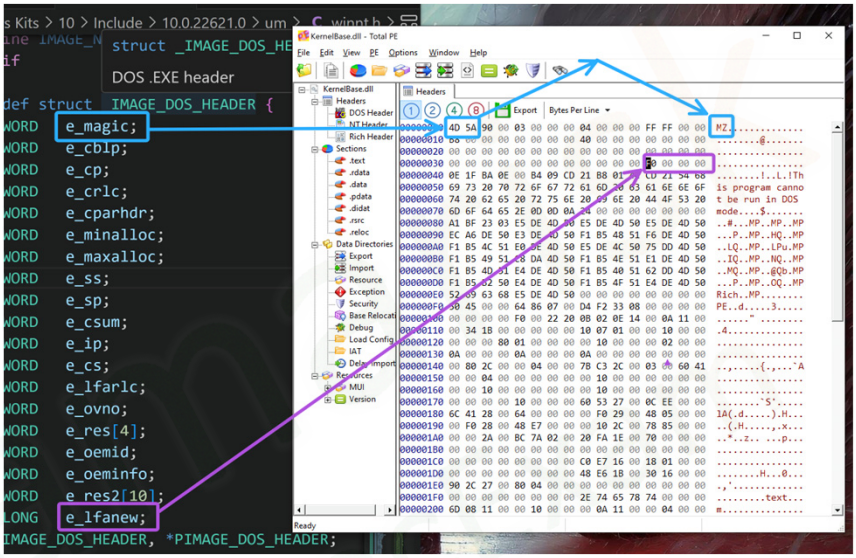
If you were developing a PE parsing tool, the other fields could be shown to the user just to give them a detailed view of the file.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

MS-DOS 2.0 EXE Header: kernelbase.dll



MS-DOS 2.0 EXE Header: kernelbase.dll

Now that you know the structure of the DOS header, we can start to make sense of the hexdump of kernelbase.dll. The screenshot is from Visual Studio Code to see the structures and Total PE, a tool written by Pavel Yosifovich. A side-by-side layout like this can help make your way through the various PE headers and the fields inside each structure. For this IMAGE_DOS_HEADER structure, almost each field is a WORD size. Now that you know this, it is easier to identify each field and move on to the next field, you just jump two (2) bytes at a time to the end. The main hexdump inside Total PE has offsets to the left of the hexdump to easily locate the various fields.

Starting with the first field, *e_magic*, we can see the famous magic value of 4D 5A for MZ. One thing to note here is the NOP (0x90) immediately following the magic header. You will see several tools that do signature scanning while only looking for a match for MZ, but this can generate some false positives. To reduce a number of false positives, check to make sure that the 3rd byte is 0x90 and the 4th byte is 00 since they rarely change. There is no 100% guarantee the bytes following MZ will always be \x90\x00, so just keep that in the back of your mind. The second field that is important to locate and understand is the *e_lfanew* field, which is traditionally found at offset 0x3C. Whatever value found there will lead you to the next header.

© 2024 Jonathan Reiter

© SANS Institute 2024

NT Headers

```
typedef struct _IMAGE_NT_HEADERS {
 DWORD                    Signature;        // 0x00 "PE"\0\0
 IMAGE_FILE_HEADER        FileHeader;       // 0x04
 IMAGE_OPTIONAL_HEADER32  OptionalHeader;                  // 0x18
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;

typedef struct _IMAGE_NT_HEADERS64 {
 DWORD            Signature;
 IMAGE_FILE_HEADER        FileHeader;
 IMAGE_OPTIONAL_HEADER64  OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;

#ifdef _WIN64
typedef IMAGE_NT_HEADERS64  IMAGE_NT_HEADERS;
typedef PIMAGE_NT_HEADERS64 PIMAGE_NT_HEADERS;
#else
typedef IMAGE_NT_HEADERS32  IMAGE_NT_HEADERS;
typedef PIMAGE_NT_HEADERS32 PIMAGE_NT_HEADERS;
#endif
```


NT Headers

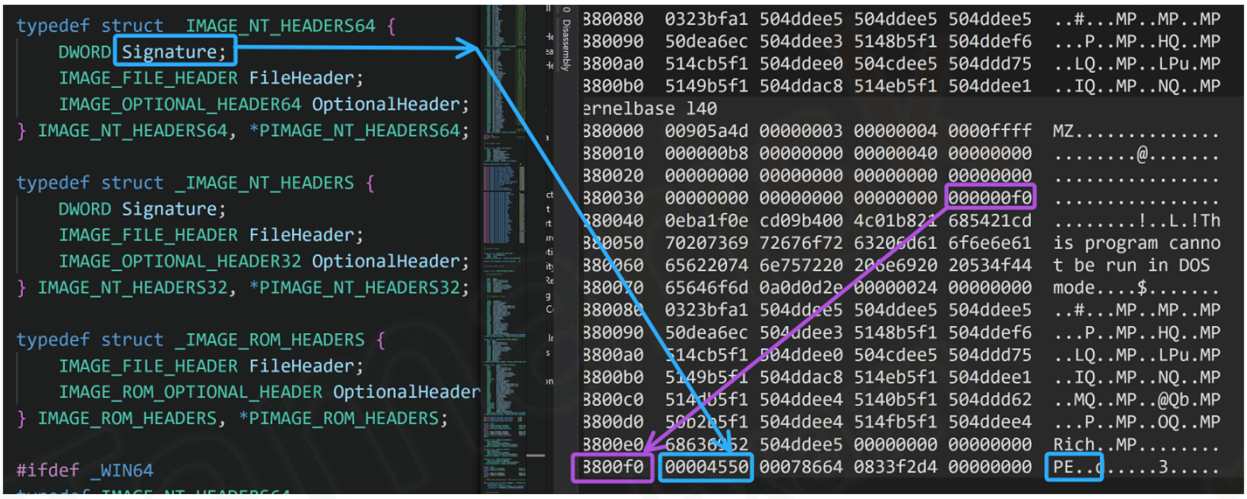
The next important structure to know is the IMAGE_NT_HEADERS struct. You will note that there is one for 32-bit applications and one for 64-bit applications. You do not need to specify one struct over the other because there is a check that happens to see if WIN64 is defined. If it is, then the structure names for 64-bit applications are typdef'd as IMAGE_NT_HEADERS. The same goes for 32-bit applications. For the moment, we will disregard that and just focus on the contents. The first field is the 4-byte signature, which is PE\0\0. The next field is the *FileHeader*, which is of struct type IMAGE_FILE_HEADER. The next slide will break down that structure, but while we are on it here there are some important fields, like the *SizeOfOptionalHeader* and *NumberOfSections*. The *NumberOfSections* is literally just that, the number of sections the executable holds: .text, .data, .bss, etc.

Following the *FileHeader* field is the *OptionalHeader* field, which is of struct type IMAGE_OPTIONAL_HEADER. This struct has a decent number of fields and several are important to us, like the DataDirectory. More on that one later.

© 2024 Jonathan Reiter

© SANS Institute 2024

NT Headers: kernelbase.dll



NT Headers: kernelbase.dll

For this side-by-side screenshot, the purple is coming from the IMAGE_DOS_HEADER->e_lfanew. That value there is used as an RVA that is then added to the base address of kernelbase.dll to give us the location of the first field in the IMAGE_NT_HEADERS struct, the *Signature*. The ASCII on the right hand side of the screenshot shows the signature being PE. Also, take note the size of the *Signature* is not two (2) bytes like it is for the IMAGE_DOS_HEADER->e_magic. This one is a DWORD, or 4 (4) bytes, so the next two (2) NULL bytes are part of it. Do not forget your endianness! Now, let us take a look at the next field, *FileHeader*.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

File Header

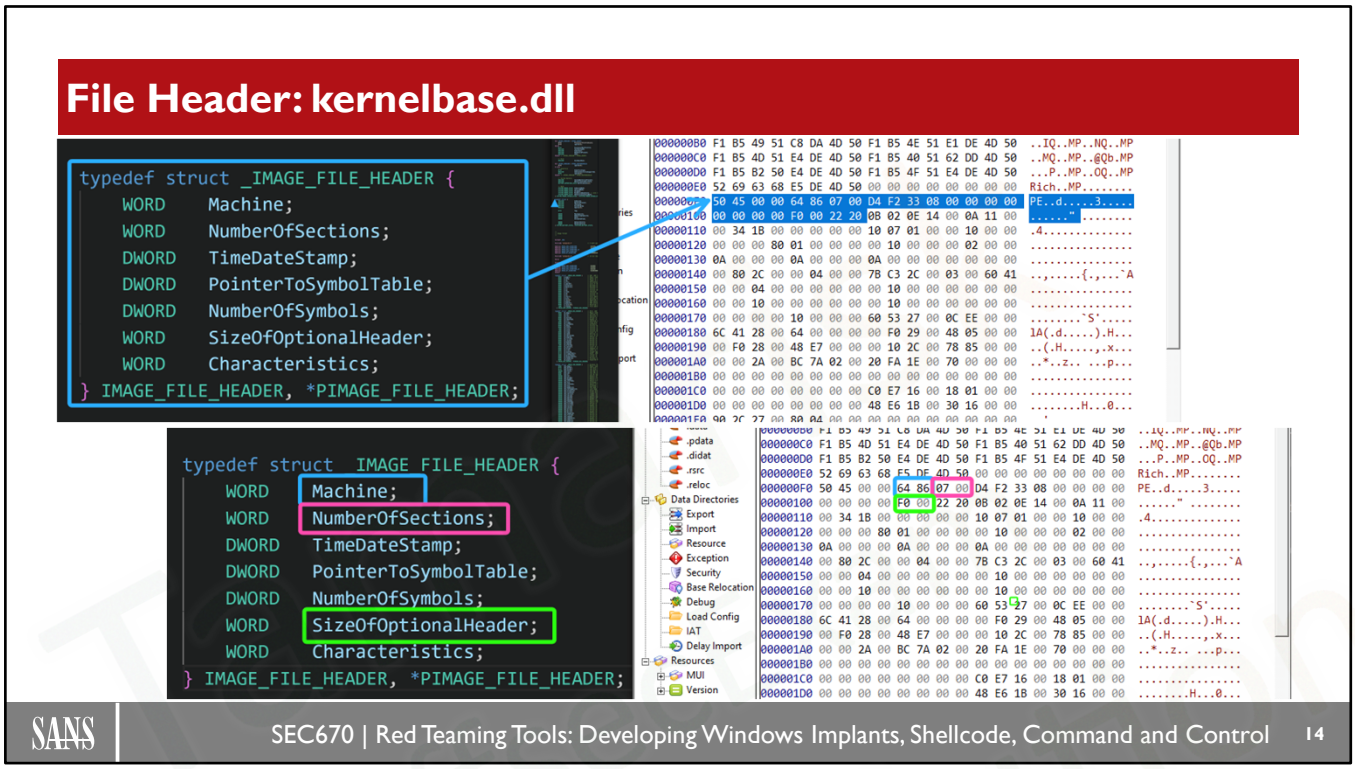
```
typedef struct _IMAGE_FILE_HEADER {
 WORD  Machine;
 WORD  NumberOfSections;
 DWORD TimeDateStamp;
 DWORD PointerToSymbolTable;
 DWORD NumberOfSymbols;
 WORD  SizeOfOptionalHeader;
 WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```


File Header

The file header holds just a few fields, but the ones that are of interest when building a PE parsing tool would be the *NumberOfSections* and the *SizeOfOptionalHeader*. The number of sections can be used in a loop to iterate over the sections, grabbing data from each one. These values also come into play when creating your own loader since each section will have to be parsed and loaded into an allocated region of memory. The header size is important because there is no direct pointer to the section headers. Therefore, once you determine the size of the optional header, you can jump directly to the section headers and start parsing those. There will be a SECTION_HEADER struct for every section in the file. The *Characteristics* field is simply some flags that are used to indicate if the file is a system file, user program, or DLL.

Tamarisk OffsecExam Exam Solution

© SANS Institute 2024



File Header: kernelbase.dll

The file header is one of the two structs inside of the NT headers struct. The upper screenshot is another side-by-side of the structure itself and the raw hexdump with Total PE. The entire IMAGE_FILE_HEADER struct is highlight in blue in Total PE so you can better see visually what it looks like. The lower side-by-side screenshot is simply highlighting the more important fields to know in the IMAGE_FILE_HEADER struct. The three (3) highlighted fields are important for us because they can be further used with our PE parsing efforts. Let us take the *NumberOfSections* field, this one indicates how many sections a binary has in it and there can be one (1) or more sections in a binary. This value can then be used to make a loop that iterates over each section until you have enumerated them all. The *SizeOfOptionalHeader* is useful because with it, we can effectively jump over the optional header and land at the first section of the binary. Typically, this first section is the .TEXT section, but that is not guaranteed. Once we have those 2 (values) we can continue with PE parsing.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Optional Header

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
 WORD      Magic;   // 0x20b
 BYTE      MajorLinkerVersion;
 BYTE      MinorLinkerVersion;
 DWORD     SizeOfCode; SizeOfInitializedData; SizeOfUninitializedData;
 DWORD     AddressOfEntryPoint;
 ULONGLONG ImageBase;
 WORD      DllCharacteristics;
 ULONGLONG SizeOfStackReserve; SizeOfStackCommit;
 IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;

typedef struct _IMAGE_DATA_DIRECTORY {
 DWORD VirtualAddress;
 DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```


Optional Header
First off, there are several fields that are not being shown here simply because they would not all fit on the slide. Let us break down the optional header.

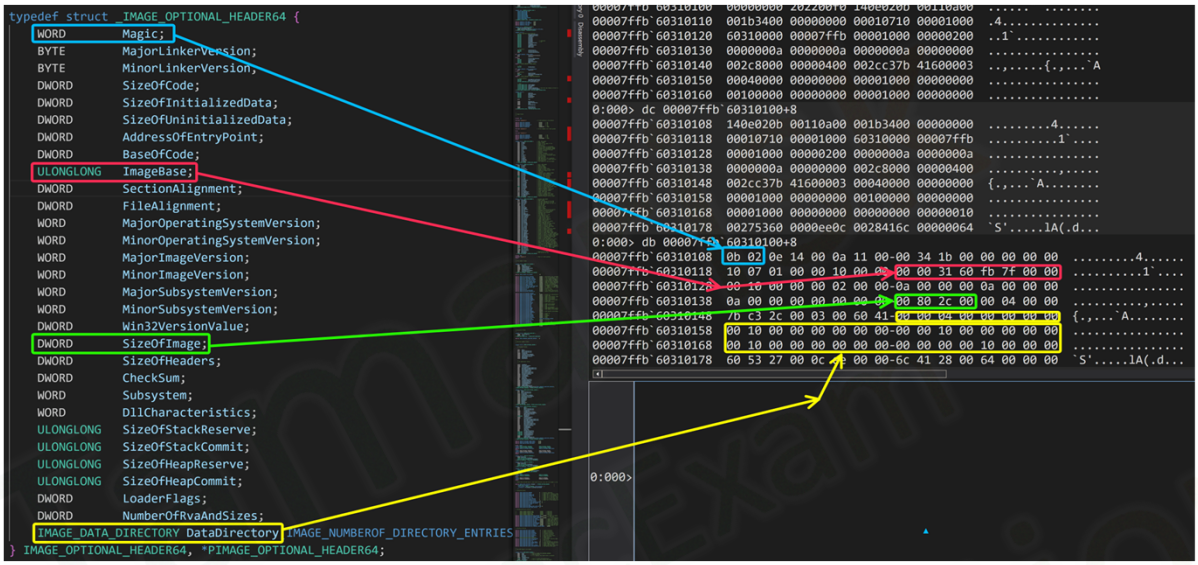
The optional header, despite its name, is not optional at all for executables; it is required. First up is the *Magic* field, which is used to indicate the state of the image file. The comment annotated on the slide (0x20b) indicates a 64-bit executable image (PE32+). On the other hand, if this were a 32-bit executable image (PE32), you would see 0x10b. The next several fields on the slide (*SizeOfCode*, *SizeOfInitializedData*, *SizeOfUnitializedData*) were placed on a single line because they are all DWORD types. These indicate the sizes for the various sections, like the .text (code), .data (initialized), and .bss (uninitialized) sections. There is also the entry point address for the program followed by the *ImageBase* field, which is the preferred address. For the *ImageBase* address, you would typically see 0x400000 for EXEs and 0x10000 for DLLs. For *DllCharacteristics*, these indicate various attributes for the image like if the image can be relocated when it is loaded, if it will be compatible with DEP, or if it will use SEH. The next two fields, *SizeOfStackReserve* and *SizeOfStackCommit*, deal with the stack. The reserve size will be set aside and only made available one page at a time when it is needed. Not all of the memory set aside for a thread's stack is committed right up front as this would just be wasteful. Rather, as stack consumption increases, more pages of reserved memory would then be committed.

The last field in the struct, and perhaps the most important one to us, is the *DataDirectory* field, which is of type IMAGE_DATA_DIRECTORY. This is a pointer to an array of data directory entries and as the comment on the slide indicates, there are 16 entries. The array entries that we care about the most are the first two, which are for the import and export entries. Inside the IMAGE_DATA_DIRECTORY struct, the field we care most about is *VirtualAdddress* as it will be key for parsing the table.

© 2024 Jonathan Reiter

© SANS Institute 2024

Optional Header: kernelbase.dll



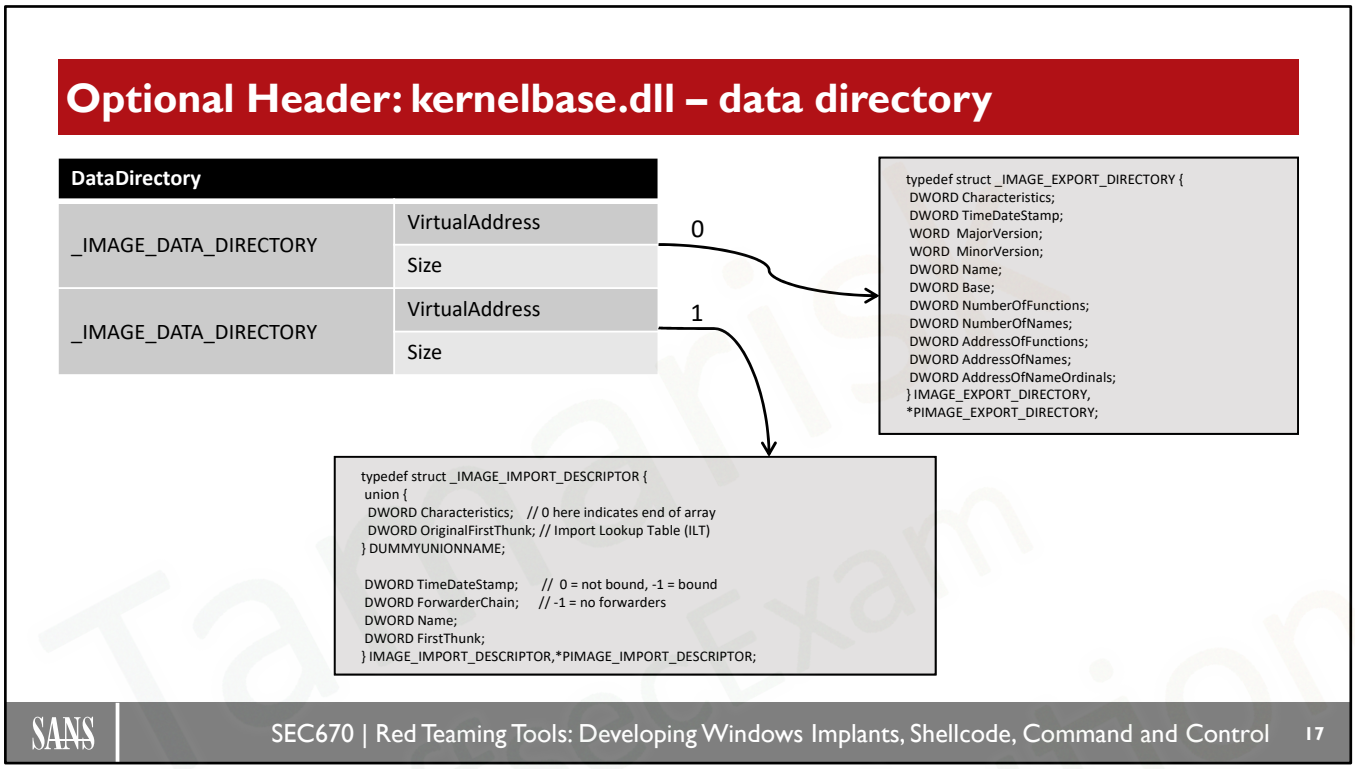
Optional Header: kernelbase.dll

This is the optional header from kernelbase.dll. The optional header is not as easy to parse because not all fields are the same size. As mentioned on the previous slide, there are a few places you can check to make sure you might be in the right place. The magic field can hold several values, but typically it will either be 0x10B or 0x20B for 32-bit (PE32) or 64-bit (PE32+), respectively. It would be very uncommon these days to see a different magic value for Windows binaries, but it could happen. The section and file alignment fields are interesting because they dictate how the sections will be aligned on disk or in memory. The section alignment value of 0x1000 indicates to the loader that the sections should be aligned on page boundaries. This is important because not every section will have the same permissions and page permissions can only be applied at single page granularity. For example, the .TEXT section of a PE32+ binary should have the characteristics of R/X. Naturally it would make sense for those same permissions be applied when mapped into pages of memory. The file alignment would be much closer together and seeing sections right next to other sections is common. Let us jump into the *DataDirectory* with a bit more detail.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Optional Header: kernelbase.dll – data directory

The last field of the optional header is arguably the most important because this is where you find the imports and exports for the image. For the most EXE binaries, there are no exports, so the value might be 0. For almost every single Windows DLL, there should be something that is being exported. You can absolutely make a DLL that does not export anything at all. Remember, the *DataDirectory* field can hold 16 arrays, or tables, which must be indexed before enumerating over them. Each entry in this table is another table of information. Inside the *DataDirectory*, you can pick apart each entry and dive into those tables to access what is need. At index 0 in the *DataDirectory* table is the entry for all exports. Each entry in the export table must be treated with the type of _IMAGE_EXPORT_DIRECTORY. Then, at index 1 in the *DataDirectory* table is the entry for all imports where each entry here must be treated with the type of _IMAGE_IMPORT_DESCRIPTOR.

Also, as you are using winnt.h as a guide for your PE parsing, you may have noticed several *#define* entries that tie into specific indexes for the *DataDirectory* table. The *#define* for exports is *IMAGE_DIRECTORY_ENTRY_EXPORT* with a value of 0. Then, there is *#define IMAGE_DIRECTORY_ENTRY_IMPORT* with a value of 1. This is a nice way to avoid the usage of what is typically called magic numbers. Magic numbers in code are just random values that can be confusing as to their meaning. Simply making a #define for them would make the code so much easier to read. Pro tip: avoid using magic numbers in your own code; just make a few #defines for them.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Exports: kernelbase.dll

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD  VirtualAddress;
  DWORD  Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

0:000> dx -r1 (*((combase!_IMAGE_DATA_DIRECTORY (*)[16])0x7ffb60310178))
    [0]          [Type: _IMAGE_DATA_DIRECTORY]  // exports
    [1]          [Type: _IMAGE_DATA_DIRECTORY]
    [2]          [Type: _IMAGE_DATA_DIRECTORY]
    [3]          [Type: _IMAGE_DATA_DIRECTORY]
    [4]          [Type: _IMAGE_DATA_DIRECTORY]
    [5]          [Type: _IMAGE_DATA_DIRECTORY]
    [6]          [Type: _IMAGE_DATA_DIRECTORY]
    [7]          [Type: _IMAGE_DATA_DIRECTORY]
    [8]          [Type: _IMAGE_DATA_DIRECTORY]
    [9]          [Type: _IMAGE_DATA_DIRECTORY]
    [10]         [Type: _IMAGE_DATA_DIRECTORY]
    [11]         [Type: _IMAGE_DATA_DIRECTORY]
    [12]         [Type: _IMAGE_DATA_DIRECTORY]
    [13]         [Type: _IMAGE_DATA_DIRECTORY]
    [14]         [Type: _IMAGE_DATA_DIRECTORY]
    [15]         [Type: _IMAGE_DATA_DIRECTORY]
```


Exports: kernelbase.dll

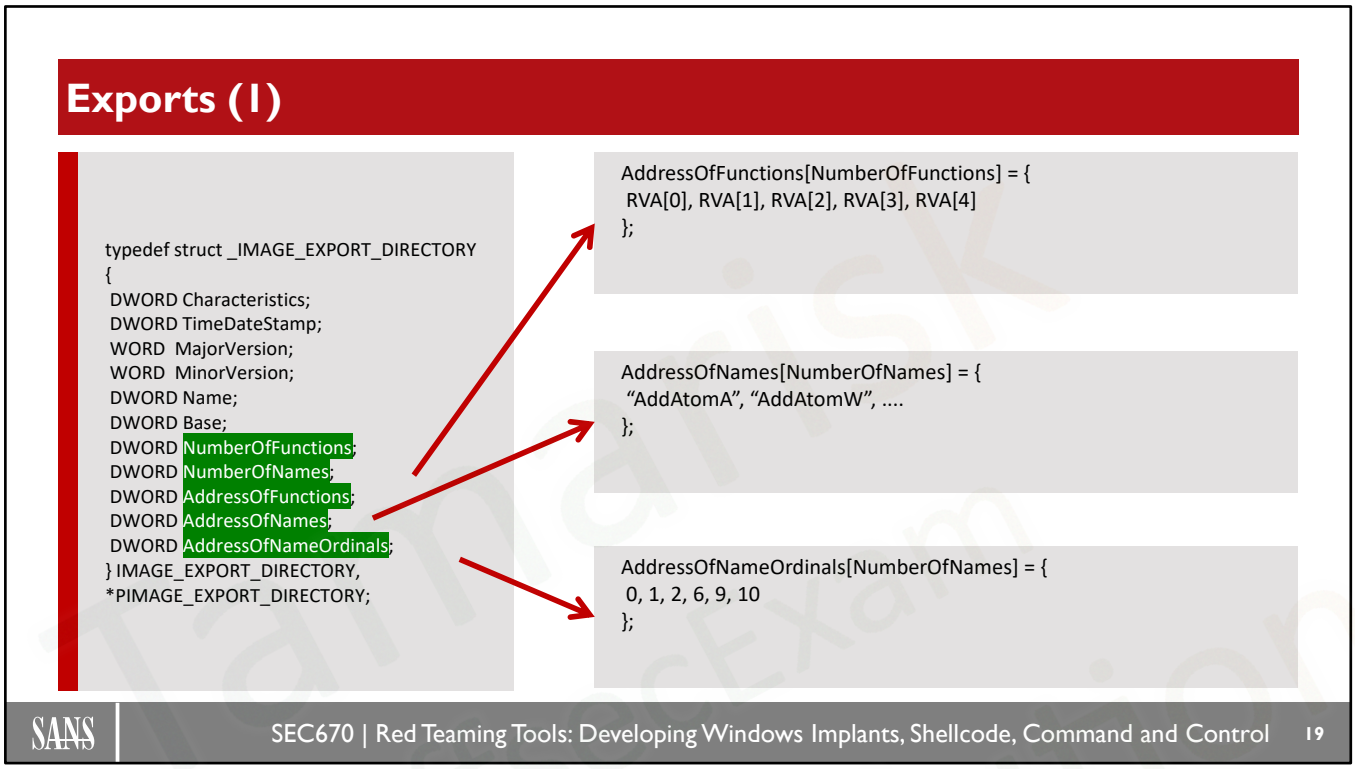
The *DataDirectory* field was located at offset F8 and since we know that it holds entries in the format of IMAGE_DATA_DIRECTORY structs, we can parse it pretty easily without using a tool.

The *VirtualAddress* of the imports entry is highlighted green, and the size of that same entry is highlighted black. When building out parsing logic, you need to remember what the data types are because they indicate how far to advance.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Exports (1)

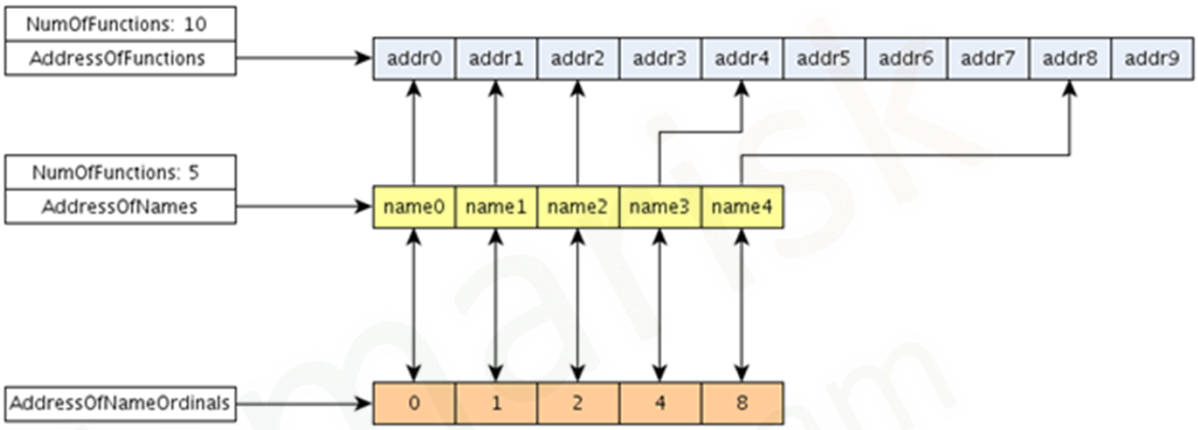
Following the exports is probably the most difficult one of all because of the added layers of complexity. The IMAGE_EXPORT_DIRECTORY struct has several important fields to us, which have been highlighted for you on the slide. The first field is the *NumberOfFunctions* and as you might have guessed, this is the number of functions that are being exported by that module (DLL). The next field, *NumberOfNames*, is the number of entries in the *AddressOfNames* and *AddressOfNameOrdinals* arrays. The last three fields are all pointers to arrays. The *AddressOfFunctions* points to an array of addresses, the *AddressOfNames* points to an array of addresses to function names, and the *AddressOfNameOrdinals* points to an array of ordinal numbers that are used as indexes into the *AddressOfFunctions* array. Multiple tools and APIs rely on these arrays for populating lists of modules and their exports, and one such API is ***GetProcAddress***.

The ***GetProcAddress***, at a high level, works by looping over the *AddressOfNames* and the *AddressOfNameOrdinals* arrays at the same time looking for a match for whatever function name you specified. Once it finds a match it can use the ordinal value as an index into the *AddressOfFunctions* array and grab the address. Later, you will be implementing your own version of ***GetProcAddress***, so be sure to know the overall logic.

© SANS Institute 2024

https://linktr.ee/offsecexam

Exports (2)



Exports (2)
This slide shows another way to visualize the flow and structure of the exports. At the top is the array holding the addresses for all of the imported functions. There happens to be 10 of them for this example, so the *NumberOfFunctions* reflects 10. If you were to only traverse the address of functions array, you would have no idea what function did what. For example, you would not know if the first entry is the address for ***GetProcAddress*** or ***TerminateProcess***. Not much can be done with that array just yet since we have not tied it to anything yet. In the middle of the graphic is the *AddressOfNames* array, which holds the addresses to the names of the imported functions. This is what is needed when creating your own version of ***GetProcAddress***. Lastly, there is the *AddressOfNameOrdinals*, which is the ordinal number that aligns with an entry in the address of names array.

The logic here is to loop over the address of names array while at the same time keeping track of your index in the array. Once you find a string match, the index you happen to be on at that time is used as an index into the address of ordinals array to obtain the ordinal number for that function name. Armed with the ordinal number, you can go right into the address of functions array and pull out that index and assign it to your function. At this point, you now have a function name, its address, and its ordinal value.

If a function is only exported by ordinal, you can simply take that ordinal and index directly into the address of functions array to grab the address. For OPSEC considerations, it is not common for a DLL to only export functions by ordinal, so keep that in mind when crafting a malicious DLL.

Reference:
resources.infosecinstitute.com/topic/the-export-directory

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Exports (3)

```
for (DWORD x = 0UL; x < NumberOfNames; x++) {
  // magical code goes here
        x=0          x=1          x=2        x=n

  strcmp(   AccessCheck | AccessCheckAndAuditAlarmW | AccessCheckByType | ...

  // found a match? Take x as an index into AddressOfNameOrdinals
        x=0          x=1          x=2        x=n

  ord =     0 | 1 | 5 | ...

  // take the ordinal value and use it as an index into AddressOfFunctions
        ord=0        ord=1        ord=2      ord=n

  rva =     1420 | 1430 | 1440 | ...
}
```


Exports (3)

For the programmatic part of it all, here is what you will have to keep in mind. Let us say you make a for loop like the pseudo code above. As you loop over the names of each function, you will be looking for a match and once you find the function of interest, the current index value of the for loop will be used as a lookup into the *AddressOfNameOrdinals* table. The ordinal value that is found at that current index is what is needed to serve as a lookup into the *AddressOfFunctions* table. Each entry in that table is an RVA so once you use the ordinal value as an index into this table, the RVA must be added to the module's base address. It might look like there is a lot going on with this slide, but there really is nothing too complicated about this. Once you start diving into the code of it all, you will see how relatively straightforward it is.

© 2024 Jonathan Reiter

© SANS Institute 2024

Exports: kernel32.dll

```
   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
8B310 00 00 00 00 DF 43 FA F5 00 00 00 00 58 00 09 00
8B320 01 00 00 00 50 06 00 00 50 06 00 00 38 C1 08 00
8B330 78 DA 08 00 B8 F3 08 00 7D 00 09 00 B3 00 09 00

typedef struct _IMAGE_EXPORT_DIRECTORY {/*sizeof: 40 bytes*/
 DWORD Characteristics;
 DWORD TimeDateStamp;
 WORD  MajorVersion;
 WORD  MinorVersion;
 DWORD Name;                    // 090058 -> KERNEL32.DLL
 DWORD Base;                    // 1
 DWORD NumberOfFunctions;       // 650
 DWORD NumberOfNames;           // 650
 DWORD AddressOfFunctions;      // 08C138 -> 9007D, 900B3, 1E310, ETC.
 DWORD AddressOfNames;          // 08DA78 -> 90065 -> AcquireSRWLockExclusive
 DWORD AddressOfNameOrdinals;   // 08F3B8 -> 00, 01, 02, ETC.
} IMAGE_EXPORT_DIRECTORY,
*PIMAGE_EXPORT_DIRECTORY;
```


Exports: kernel32.dll

The example here is from a module that is a bit more mature, Kernel32.dll, which exports a larger number of functions than that of a traditional EXE. This library seemed like a perfect choice since it is loaded into almost every process. Kernel32.dll is also one that is relied upon very heavily by implant developers like us and by malware authors for the use of functions like ***GetProcAddress*** and ***LoadLibrary***. You might often see position independent shellcode rely on these two APIs and that is to give the malware the ability to load modules and find the addresses of other important APIs.

Let us break down this hexdump and jump straight to the *NumberOfFunctions* field and obtain that value; 0x650. Immediately following that is the *NumberOfNames*; 0x650. Next, in red, is the *AddressOfFunctions* with RVA 0x08C138. *AddressOfNames* follows with RVA 0x08DA78, and then last field is *AddressOfNameOrdinals* with RVA 0x08F3B8. Armed with the RVAs, you can then follow them to start diving into each of the arrays like the *AddressOfNames* array. Comments on the slide have already begun following the RVAs to their values. Feel free to follow along on your Windows 10 Dev VM using a tool like PE-bear or PE Explorer.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Imports: kernelbase.dll

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD  VirtualAddress;
  DWORD  Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

0:000> dx -r1 (*((combase!_IMAGE_DATA_DIRECTORY (*)[16])0x7ffb60310178))
    [0]          [Type: _IMAGE_DATA_DIRECTORY]
    [1]          [Type: _IMAGE_DATA_DIRECTORY]  // imports
    [2]          [Type: _IMAGE_DATA_DIRECTORY]
[.. SNIP ..]
    [15]         [Type: _IMAGE_DATA_DIRECTORY]

0:000> dx -r1 (*((combase!_IMAGE_DATA_DIRECTORY *)(0x7ffb60310178 + 8)))
    [+0x000] VirtualAddress   : 0x28416c
    [+0x004] Size             : 0x64
```


Imports: kernelbase.dll

The *DataDirectory* field of the _IMAGE_OPTIONAL_HEADER64 was located at offset 0x70 and since we know that it holds entries in the format of IMAGE_DATA_DIRECTORY structs, we can parse it pretty easily without using a tool, or we can just use WinDbg! The address used in the WinDbg command on the side was calculated after adding the base address to the RVA found for the *DataDirectory* field. We simply cast it to the proper type needed to see the results, but we also must do something different here because we are dealing with an array. We use the [16] in the command because we want the 16 entries to be displayed. Since we also know the size of each IMAGE_DATA_DIRECTORY entry is 8 bytes, we can manually calculate the next entry just like you see in the command on the slide. Please note, just because it says *VirtualAddress*, does not mean the actual virtual address because it is still very much an RVA that must be added to the base address of the module. Before going any further into the imports part of the PE header, let us take a look at the definition of the _IMAGE_IMPORT_DESCRIPTOR.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Imports Structure

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
 union {
  DWORD Characteristics;    // 0 here indicates end of array
  DWORD OriginalFirstThunk; // Import Lookup Table (ILT)
 } DUMMYUNIONNAME;

 DWORD TimeDateStamp;      //  0 = not bound, -1 = bound
 DWORD ForwarderChain;     // -1 = no forwarders
 DWORD Name;
 DWORD FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR,*PIMAGE_IMPORT_DESCRIPTOR;
```


Imports Structure

The imported libraries (DLLs) and their functions are stored in this array of IMAGE_IMPORT_DESCRIPTORs, which is index 1 of the *DataDirectory* array. Remember, index 0 is for the exports, which we will cover later. Highlighted on the slide are some important fields, the first one being *OriginalFirstThunk*. As the comment indicates, this is simply the RVA to get to the ILT; Import Lookup Table. Keep in mind that this is only temporary until the loader is done processing the imports and locating addresses for the imported functions. On disk, it is the RVA to the IAT; Import Address Table. Sometimes, imported functions are simply jump entries in a table that jumps program code to another DLL where the function is defined. These are called forwarded functions and there is a field called *ForwarderChain* to indicate it. If there are no functions that are forwarded, then this field will be set to -1. The next field is *Name*, which is the RVA to the name of the DLL. After that is the *FirstThunk* field, which is the RVA of the IAT. On disk, before the EXE is loaded, the *OriginalFirstThunk* and the *FirstThunk* should be pointing to the same location or have the same RVA and what they are both pointing to is another array. This array holds IMAGE_THUNK_DATA entries that represent each imported DLL. To indicate to the system loader that there are no more entries in the list, one last entry of type IMAGE_THUNK_DATA will be made but all fields will be NULL. So, if a program has 18 imported DLLs, then there will be 19 IMAGE_THUNK_DATA entries and the last one will be all NULLs to indicate the end of the array.

 © 2024 Jonathan Reiter

© SANS Institute 2024

Imports: kernelbase.dll

```
0:000> dx -r1 (*((combase!_IMAGE_DATA_DIRECTORY *)0x7ffb60310180))
  [+0x000] VirtualAddress   : 0x28416c
  [+0x004] Size             : 0x64

0:000> ? kernelbase + 0x28416c
  Evaluate expression: 140717629981036 = 00007ffb`6059416c

0:000> dt combase!IMAGE_IMPORT_DESCRIPTOR 00007ffb`6059416c
  +0x000 Characteristics  : 0x284220
  +0x000 OriginalFirstThunk : 0x284220  // import lookup table
  +0x004 TimeDateStamp    : 0         //  0 = not bound, -1 = bound
  +0x008 ForwarderChain   : 0         // -1 = no forwarders
  +0x00c Name             : 0x286a0a  // name of the imported DLL
  +0x010 FirstThunk       : 0x1be698

0:000> ? kernelbase + 0x286a0a
  Evaluate expression: 140717629991434 = 00007ffb`60596a0a
0:000> dc 00007ffb`60596a0a
  00007ffb`60596a0a  6c64746e 6c642e6c 0547006c 516c7452  ntdll.dll.G.RtlQ
```


Imports: kernelbase.dll

There is a lot happening on this slide, but do not worry, we will break it down a step at a time. The output here is from the *DataDirectory* array focusing on the imports entry. In code, it could look something like the following: *OptionalHeader.DataDirectory*[IMAGE_DIRECTORY_ENTRY_IMPORT]. Even though we went over the IMAGE_IMPORT_DESCRIPTOR on the previous slide, it is included here for easy reference while making sense of the hexdump. The size of the struct is 0x10 bytes with each field being a DWORD (4 bytes), which makes parsing it pretty easy. Remember, the first field is really a union so that is why the output in the debugger shows both at offset 0x00. The first field is the *OriginalFirstThunk*, which holds the value 0x284220. If you would follow that RVA, it would lead you to another RVA. Skipping down to the *FirstThunk* field, you can see the RVA of 0x1be698, which when followed shows you yet another RVA to find your data. The *Name* field is the name of the imported module, and it also is an RVA that can be followed. In WinDbg it is easy to follow RVAs. You can see on the slide that the *Name* RVA, when added to the base address of the module, is: ntdll.dll. The next 20 bytes would be specific for the next entry and so on and so on until you find a NULL entry. Also, you should start to see the pattern of finding your data when RVAs are given to you. Simply take the RVA value and add it to the module's base address and there you have your data. Here is a WinDbg way of getting to the next import entry.

```
0:000> dt combase!IMAGE_IMPORT_DESCRIPTOR 00007ffb`6059416c + 0x14
   +0x00c Name             : 0x286da4
0:000> ? kernelbase + 0x286da4
   Evaluate expression: 140717629992356 = 00007ffb`60596da4
0:000> dc 00007ffb`60596da4
   00007ffb`60596da4  2d697061 772d736d 652d6e69 746e6576  api-ms-win-event
   00007ffb`60596db4  2d676e69 766f7270 72656469 2d316c2d  ing-provider-l1-
   00007ffb`60596dc4  2e302d31 006c6c64 7369091e 68706c61  1-0.dll
```

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Lab 3.1: GetFunctionAddress



Parse a PE file to obtain the address of a given function

Please refer to the eWorkbook for the details of the lab.

Lab 3.1: GetFunctionAddress
Please refer to the eWorkbook for the details of the lab.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

What's the Point?

What's the point?

What's the Point?
The point of this lab was to help you become intimately familiar with the format that PE files use. You should have been able to see the important information that the loader requires to properly load a file on disk and into memory. This will not be the last time you do some PE parsing. Keep this tool handy for later.

Tamarisk OffSecExam Exam Solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Module Summary



Learned the PE structure is complicated at first

Covered that understanding the key components aids in future techniques

Discussed how messing with the PE structure can yield interesting results

Module Summary

In this module, we covered the PE header in depth and many of the important structures that make up the PE header. Armed with this knowledge, you can proceed to experiment with adding additional headers to see what the loader does when it comes across, let us say, two valid PE headers. The results can be interesting, to say the least.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Questions



What is typically the next byte that comes after the MS-DOS header?

A 0x00

B 0x90

C 0x5A

Unit Review Questions
Q: What is typically the next byte that comes after the MS-DOS header?

A: 0x00

B: 0x90

C: 0x5A

Tamarisk OffsecExam Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Answers



What is typically the next byte that comes after the MS-DOS header?

A 0x00

B 0x90

C 0x5A

Unit Review Answers
Q: What is typically the next byte that comes after the MS-DOS header?

A: 0x00

B: 0x90

C: 0x5A

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Questions



In the optional header, what magic value indicates a PE32+ binary?

A 0x20B

B 0x10B

C 0x00B

Unit Review Questions
Q: In the optional header, what magic value indicates a PE32+ binary?

A: 0x20B

B: 0x10B

C: 0x00B

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Answers



In the optional header, what magic value indicates a PE32+ binary?

A 0x20B

B 0x10B

C 0x00B

Unit Review Answers
Q: In the optional header, what magic value indicates a PE32+ binary?

A: 0x20B

B: 0x10B

C: 0x00B

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- **Operational Actions**
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 3

PE Format
- Lab 3.1: GetFunctionAddress

Threads

Injections
- Lab 3.2: ClassicDLLInjection
- Lab 3.3: APCInjection
- Lab 3.4: ThreadHijacker

Escalations
- Lab 3.5: TokenThief

Bootcamp
- Lab 3.6: So, You Think You Can Type
- Lab 3.7: UACBypass-Research
- Lab 3.8: ShadowCraft

Threads

Before we move into injections, we need to continue learning some of the internals of the system, particularly threads.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Objectives

Our objectives for this module are:

- Define a thread
- Understand various thread states
- Understand thread contexts
- Explore the structure of a thread
- Create a thread

Objectives

The objectives for this module are to define what a thread is. We will explore the various states a thread can be in and what state we need a thread to be in for a certain injection method. Each thread will have its own context that becomes relevant when a thread enters its quantum. We will also look at how a thread is structured in the system and the components of them. Lastly, we will get some hands-on practice with creating threads.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Definition



What is a thread?

According to MSDN: "A thread is an entity within a process that can be scheduled for execution."

Definition

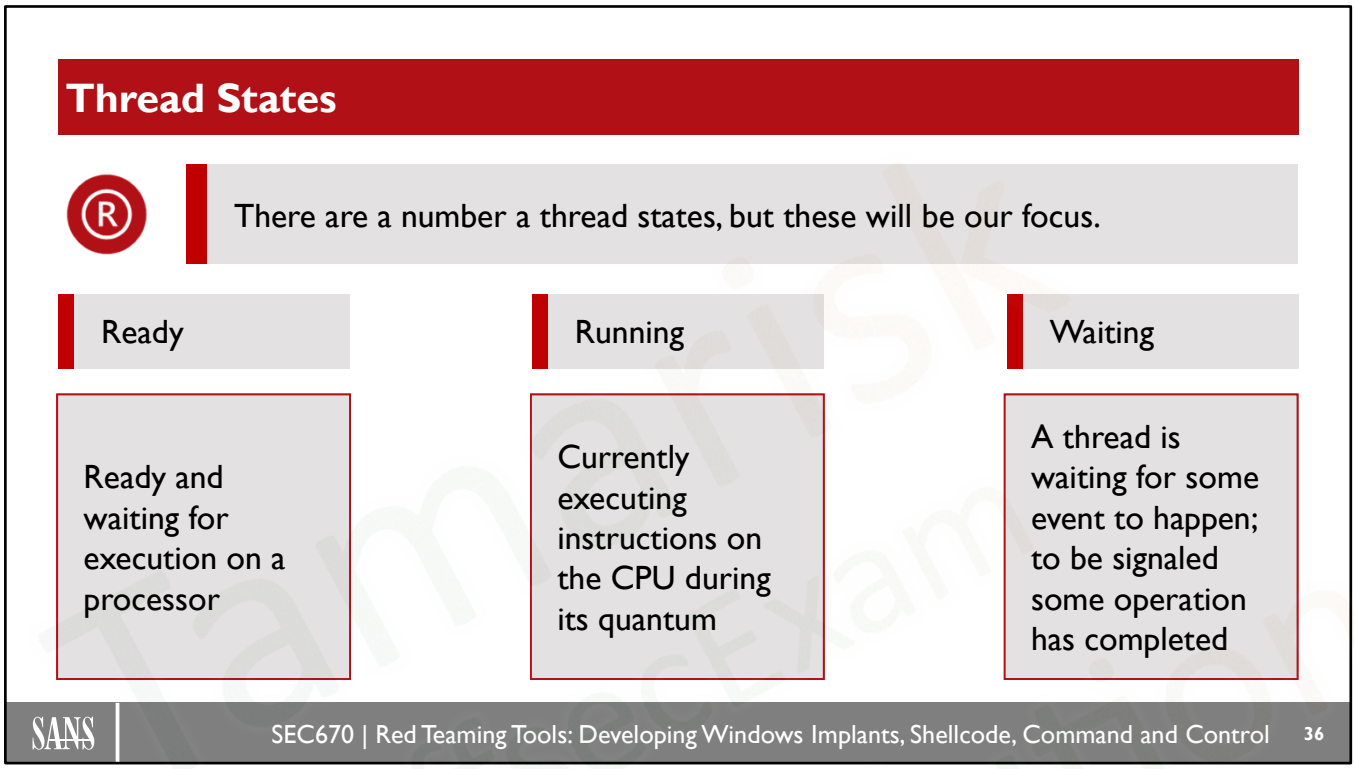
MSDN provides us with a definition of a thread: "A thread is an entity within a process that can be scheduled for execution." What does that really mean? Well, first things first, threads are what execute the instructions of a program and they are to be eventually executed by the CPU. Think of a thread as the smallest unit of execution that is tied to a process. Each process will have at least one thread that kicks off the image's code at its *AddressOfEntryPoint*. This is done after the system loader finished mapping an executable image into memory and that first is injected into the process by the kernel. This routine is how a program's main function is eventually called and inside of main's function body, more threads can be created to accomplish various tasks.

Remember, threads run and execute on a system, and processes are just mapped sections of an executable image in memory.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Thread States

There are at nine states a thread can be in at any given moment, but the focus here will only be on three states as noted on the slide: Ready, Running, and Waiting. A thread is said to be in the Ready state when it is waiting for execution and the Windows dispatcher will only schedule threads to run when they are in the Ready state. Running is the state for when a thread has officially entered its quantum on the processor and the processor has completed its switch to it. Waiting is a state where a thread is hanging around waiting for a specific event to happen. Most of the time, a thread enters the Waiting state on a voluntary basis, meaning the thread called a function like ***WaitForSingleObject***, which will make the thread become alertable. The thread then waits for that object to sync up with its execution, and a great tool to use when debugging a multithreaded program is Performance Monitor. Of course, you cannot go wrong with the best tool of all time, WinDbg Preview.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Thread Scheduling



The Windows dispatcher

Because Windows is a preemptive, and priority-based system, threads can be selected for execution but never execute because it gets preempted by a thread with a higher priority. Threads run for a certain number of clock cycles during their quantum.

Thread Scheduling

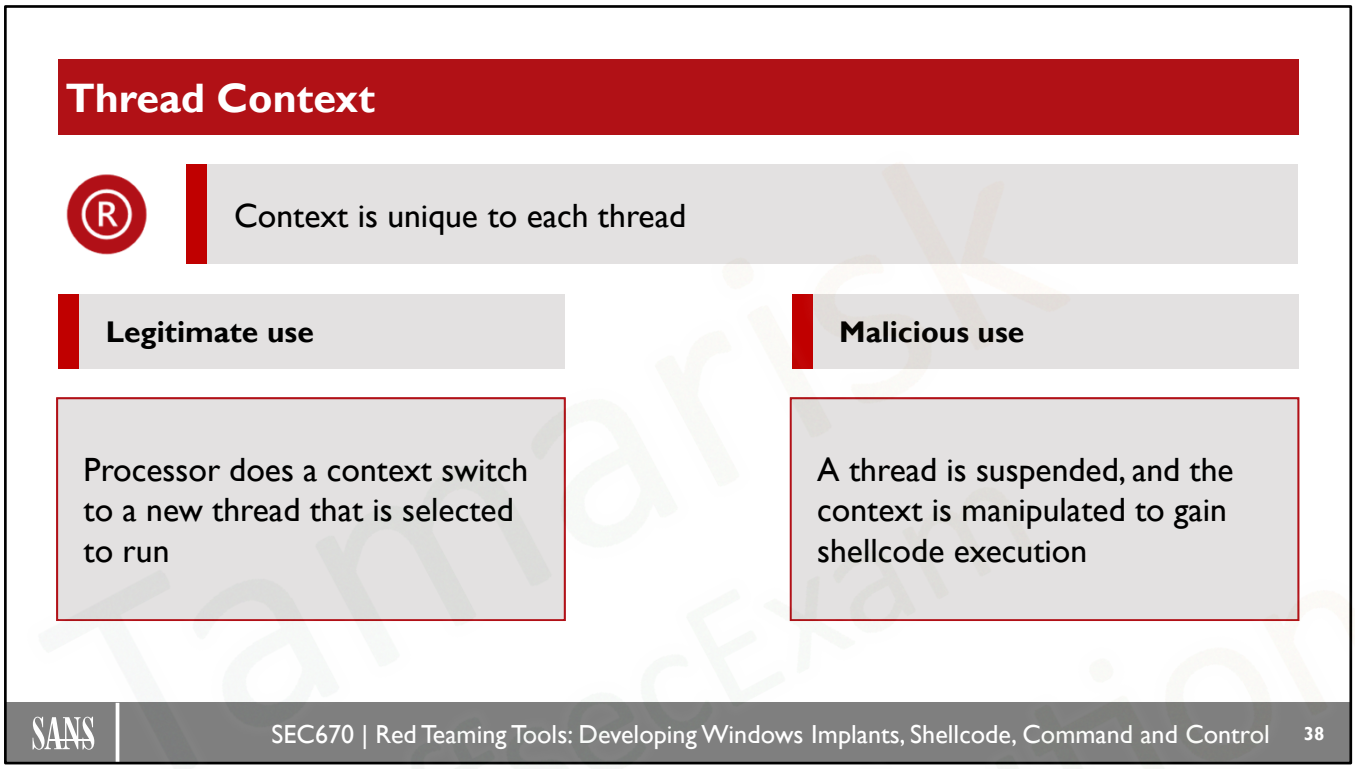
Threads can be assigned higher levels of priority that enable them to run before threads with a lower priority. When a thread with a high priority leaves its waiting state and becomes ready to run, it will preempt any other thread that is currently in its quantum if it has a lower priority. There are caveats with this, but this is the typical scheduling operation. There are a few events that might trigger the dispatcher to wake up and do its job, so let us discuss a few. One such event is the one previously mentioned: a thread leaving its waiting state. Another event is a newly created thread that is ready to run and must be scheduled. Thread priorities can be changed by system services or when it is about to hit a service call. With all the scheduling happening with threads, context switching becomes vital.

For completeness, a quantum is the time that is allotted for a thread to run on a CPU. Servers typically have a default quantum of 12 clock intervals while traditional endpoints have a quantum of 2 clock intervals. Of course, the quantum values can be changed but again, those are typically the defaults.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Thread Context

Before a different thread executes, the current state of the registers must be saved. Each thread has a context that is saved when its quantum is over or is preempted by a thread with a higher priority. These saved context states are swapped in and out each time a thread is entering its quantum. This is called context switching. The CR3 CPU control register plays a very important role when it comes to context switching because it holds the physical address of the PML4 table for x86_64 and the Page Directory Table for x86. Whatever is in the CR3 register represents the current context for the CPU. When it comes to contexts, Windows has a few APIs to deal with them: ***GetThreadContext*** and ***SetThreadContext***. We can leverage these two APIs to redirect execution to our shellcode by grabbing the context of a thread, manipulating it, then later resuming the thread. We will be doing some thread context manipulation later in this section.

For more detailed information about threads, the *Windows Internals* books are a great resources as well as the book *What Makes It Page?* for more of a memory manager perspective.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Thread Structure



ETHREAD/KTHREAD/TEB

All reside in system address space except the TEB

```
struct _ETHREAD {
 _KTHREAD       Tcb; // thread control block
 _LARGE_INTEGER CreateTime;
 PVOID          SartAddress;
 [... SNIP ...]
};

struct _TEB {
 _NT_TIB     NtTib;
 _CLIENT_ID  ClientId;
 ProcessEnvironmentBlock;    // PEB
};

struct _NT_TIB {
 ExceptionList; // EXCEPTION_REGISTRATION_RECORD
 StackBase;
 StackLimit;
};
```


Thread Structure

The kernel holds the ETHREAD and KTHREAD objects in system space, but not the TEB. The structure of a thread and its environment block are important to know, but because the ETHREAD and KTHREAD objects reside in system space, we will only be concerned with the TEB. Each thread will have its own environment block called the Thread Environment Block, or the TEB. The process and the loader must be able to read information in this structure, so that is why it resides in the process address space. You can see that the first field of the TEB is the TIB, which was originally used for Win9X applications and has remained for backwards compatibility. The TIB holds information relevant to the list of exceptions, and the size limitations of a thread's stack. For those who have taken SEC660 or FOR610, SEH should be familiar to you. For those who did not, there is a list comprised of addresses that will be called in an attempt to handle an exception. There is an entire chain of these that are linked together and the mechanism driving this is structured exception handling.

Back to the TEB, another important field is the pointer to the Process Environment Block of the parent process. The PEB, as mentioned earlier, is unique per process, and it holds valuable information we need to leverage to accomplish certain tasks, like finding loaded modules and function addresses.

In a kernel debugging session, *dt nt!_ethread* or *_kthread* will show you the structures and *dt nt!_teb* will should you the TEB struct.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

CreateThread / CreateRemoteThread



CreateThread
CreateRemoteThread

Used to create a local/remote thread

Has a HANDLE return type

```
HANDLE CreateThread(
 LPSECURITY_ATTRIBUTES lpThreadAttributes,
 SIZE_T   dwStackSize,
 LPTHREAD_START_ROUTINE lpStartAddress,
 LPVOID   lpParameter,
 DWORD    dwCreationFlags,
 LPDWORD  lpThreadId
);

HANDLE CreateRemoteThread(
 HANDLE  hProcess
 LPSECURITY_ATTRIBUTES lpThreadAttributes,
 SIZE_T  dwStackSize,
 LPTHREAD_START_ROUTINE lpStartAddress,
 LPVOID  lpParameter,
 DWORD   dwCreationFlags,
 LPDWORD lpThreadId
);
```

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 40

CreateThread / CreateRemoteThread
Arguably the easiest way to create a local thread is to call the ***CreateThread*** API. When the API successfully returns, it will provide a valid handle to the newly created thread. Let us break down the arguments to better understand what this function needs.

lpThreadAttributes, is an optional pointer to a SECURITY_ATTRIBUTES struct that is used to determine what security attributes will be applied to the returned handle. Leaving this NULL will not allow the handle to be inherited. For our purposes, we really do not care if our thread handles are inheritable, yet.

dwStackSize, lets you determine the size of the stack in bytes. The system will automatically round up your number to the nearest page since it must be on a page boundary. You can pass 0 here and the default stack size will be used, which is typically 1MB. The stack size is also stored in the executable image's header.

lpStartAddress, is a pointer to a function the new thread should start executing. This can also be shellcode, which is what we care about the most.

lpParameter, is a pointer to some variable that you want to be passed along to the newly created thread.

dwCreationFlags, is a thread can be created using a couple different flags, like CREATE_SUSPENDED. For the most part, 0 will be passed here since we would want the thread to execute the function right when it is created.

lpThreadId, is an optional pointer to some variable to store the thread's ID. Most of the time, we do not always care about the ID of our newly created thread, so we can just pass NULL here.

There is another variant of this function that will create a thread in a remote process: ***CreateRemoteThread***. Everything is the same except the first argument, which is a valid handle to the process.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Creating Threads



What happens behind the scenes?

- Parameters converted to flags; Client ID and TEB address added to an attribute list
- Determine if the thread should be created in local or remote process
- Call NtCreateThreadEx, initialize user-mode thread context, call PspCreateThread
- Thread is initially suspended and then later resumed so it can be scheduled

Creating Threads

For this scenario of thread creation, the main thread of a process has already been created and is running.

- The main thread creates a new thread using one of the Create Thread APIs.
- The function ends up calling an extended version called ***CreateRemoteThreadEx*** even if your code only calls ***CreateThread***. The extended version will take the arguments that were passed in and convert them to flags. It will also create the necessary native structures to describe the object parameters for the system to utilize.
- Before the thread can be made, the system must know if it should make the thread in the local calling process or another process. This determination can be done by checking the value of the handle.
- Once that is determined, it can call ***NtCreateThreadEx*** from Ntdll to make the jump into kernel mode with the exact same arguments.
- The user mode thread context will be made, and the executive thread object will be suspended by calling ***PspCreateThread***. After the system is done initializing everything, the thread is eventually resumed and is now ready to be scheduled for its quantum.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Module Summary



- Defined threads
- Covered various threads states
- Discussed thread contexts and structures

Module Summary

In this module, we discussed what threads are from an internal perspective, we explored how to create threads using two different APIs, we looked at contexts, quantums, etc. We now have enough knowledge to understands what happens when we inject into a thread or suspend a thread.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Questions



When a new thread is created, where does the object reside?

A User space

B System space

C Process handle table

Unit Review Questions
Q: When a new thread is created, where does the object reside?

A: User space

B: System space

C: Process handle table

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Answers



When a new thread is created, where does the object reside?

A User space

B System space

C Process handle table

Unit Review Answers
Q: When a new thread is created, where does the object reside?

A: User space

B: System space

C: Process handle table

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Questions



What state is a thread in during its quantum slice?

A Running

B Waiting

C Ready

Unit Review Questions
Q: What state is a thread in during its quantum slice?

A: Running

B: Waiting

C: Ready

Tamarisk OffsecExam Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Answers



What state is a thread in during its quantum slice?

A Running

B Waiting

C Ready

Unit Review Answers
Q: What state is a thread in during its quantum slice?

A: Running

B: Waiting

C: Ready

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Questions



What is the default quantum for servers?

A 2 clock cycles

B 8 clock cycles

C 12 clock cycles

Unit Review Questions
Q: What is the default quantum for servers?

A: 2 clock cycles

B: 8 clock cycles

C: 12 clock cycles

Tamarisk ObscExam Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Answers



What is the default quantum for servers?

A 2 clock cycles

B 8 clock cycles

C 12 clock cycles

Unit Review Answers
Q: What is the default quantum for servers?

A: 2 clock cycles

B: 8 clock cycles

C: 12 clock cycles

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- **Operational Actions**
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 3

PE Format
- Lab 3.1: GetFunctionAddress

Threads

Injections
- Lab 3.2: ClassicDLLInjection
- Lab 3.3: APCInjection
- Lab 3.4: ThreadHijacker

Escalations
- Lab 3.5: TokenThief

Bootcamp
- Lab 3.6: So, You Think You Can Type
- Lab 3.7: UACBypass-Research
- Lab 3.8: ShadowCraft

In this module, we will discuss several techniques centered around injection. There are a large number of injection methods, and as you can see, we have our hands full of exercises in this section. To kick things off, this module will start with the classic DLL injection.

Tanay OffSecExam Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Objectives

Our objectives for this module are:

- Define what injection is and reasons for it
- Explore several injection techniques and their mechanisms
- Understand more about the PE structure
- Rehash threads
- Discuss some mitigations

Objectives

The objectives for this module might seem light on the surface, but they are not. We are going to be diving deep into various methods of injection followed by a brief mention of possible mitigations.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Process Injection

What exactly is process injection and what are some reasons for injecting into a process?

Forcefully making a process execute arbitrary code

Avoid detection by having a legit process execute your shellcode

Process Injection

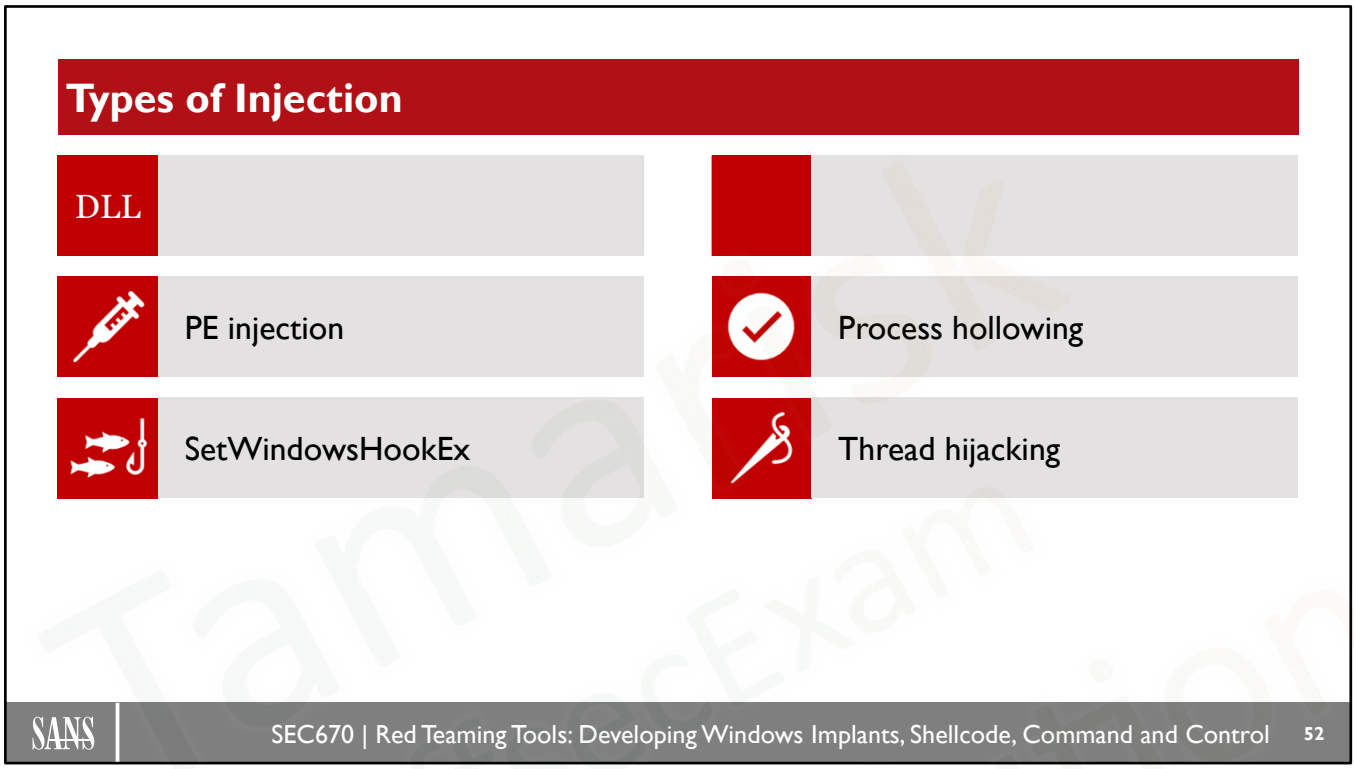
Depending on what blog post you read or what YouTube video you watch, you might get a different definition of what process injection is. For this course, we define process injection as a method of forcing code from one userland process, say malware, into another userland process to execute arbitrary code. We will discuss other techniques that are not true to that definition, like AppInit, Image File Execution Options, DLL hijacking, or other pre-execution styles. It is good to know many techniques and what some of the cons might be for a certain technique.

There could be several reasons for injecting into a process, but what we are concerned with is trying to avoid various levels of detection. You could easily develop a program that executes shellcode locally, meaning inside your own process' virtual address space. Detecting local execution could be rather trivial, but if you could have another process, say a legitimate Windows process or third party application execute your malicious deeds, why not have it do so. Depending on what technique you implement, this could help avoid detection for a longer period of time.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Types of Injection

There are several process injection methods, and this slide is not a comprehensive list of every single process injection technique that is out there, but it is enough to get us going. At this point, a few injection methods will be discussed at a high level and later on will be the deep dive into each one. First up is the famous DLL injection. There are two different versions of DLL injection: classic and reflective. In this section, we will just focus on classic and save reflective injection for later in the course. At a high level, this technique forces another process to load and execute a DLL of your choosing.

Next is APC injection. The asynchronous procedure call is a Windows mechanism that allows functions to execute, as the name implies, asynchronously in some thread. With this technique, we look for certain threads, or all of them, that we can queue an APC to and have it execute our code.

PE injection is what I call PE inception because we are hiding a PE inside of another PE, a dream within a dream. Once the attack is executed, the target process will be holding two PE images, the legitimate one and ours.

Process hollowing creates a new process, but it does so in the suspended state so no code can execute from it. Then it proceeds to hollow out the original image with a new one and once done, the main thread can be resumed. In the end, it still holds the same process name, but on the inside, it is completely different.

SetWindowsHookEx is an API that can be used to inject a DLL into a GUI process. It could also be one of many methods used to act as a key logger. This technique depends on the process having the User32 module loaded.

Thread hijacking is the manipulation of a thread's context so that it will wind up executing shellcode we give it.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Classic DLL Injection



The injector and the injectee

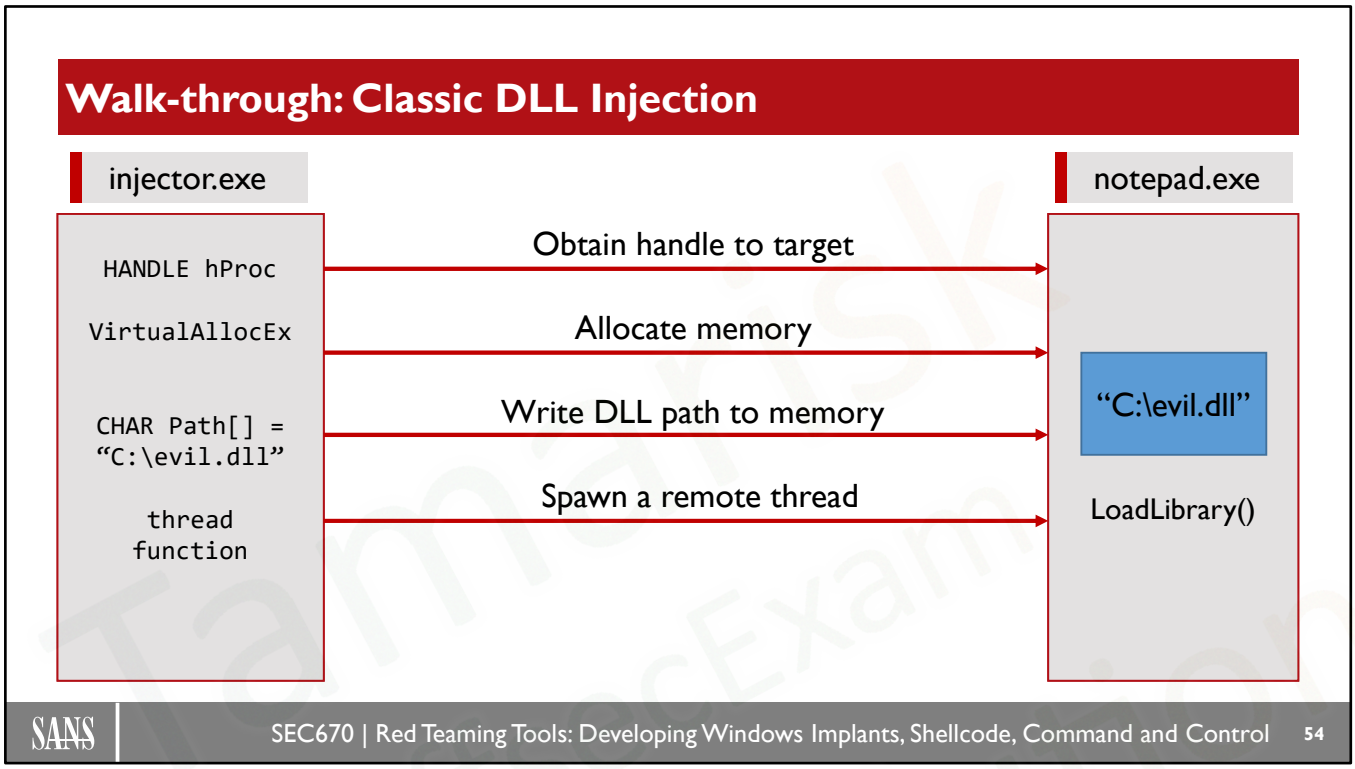
For DLL injection to work properly, you must obtain a handle to the target processes to allocate memory pages, write to them, and create a new thread. This logic will be implemented in the DLL injector. The DLL is the malicious portion of this method.

Classic DLL Injection

Classic DLL injection is fairly common as it is not too complex of a method to understand. The main functions involved with this method are ***OpenProcess***, ***VirtualAllocEx***, ***WriteProcessMemory***, and ***CreateRemoteThread***. The main purpose of this method is to forcefully load a DLL into the address space of the target process and then execute it. The items you must provide as the tool developer would be the injector portion of the tool. The DLL to be injected can be provided by the operator of the tool, or even by you. You will be creating your own at first because you need to test your tool to work out any bugs.

© 2024 Jonathan Reiter

© SANS Institute 2024



Walk-through: Classic DLL Injection

Classic DLL injection is arguably the easiest to perform. If you were to look at this from a 10,000-foot view of the overall process, this is what it would look like. There are a few things that are needed for this to be successful: the injector and the DLL. The injector has the responsibility of executing the logic for injecting the DLL into the target process. One of the first steps of the injector is to obtain a valid process handle to the target process via ***OpenProcess***. The handle must have the necessary permissions to get the job done; permissions will be discussed during the source code review. Once the handle is obtained, a chunk of memory must be allocated in the target process via ***VirtualAllocEx***. This newly allocated space is used to store the absolute path to the DLL via ***WriteProcessMemory***. The final step is to call the ***CreateRemoteThread*** API and hope everything works as desired.

With these steps in mind, let us jump over to the source code.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Lab 3.2: ClassicDLLInjection



Injecting your own DLL into a target process

Please refer to the eWorkbook for the details of the lab.

Lab 3.2: ClassicDLLInjection
Please refer to the eWorkbook for the details of the lab.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

What's the Point?

What's the point?

What's the Point?
The point of this lab was to get your feet wet with injecting a DLL into a target process. This also puts down the foundation so it can be built upon later in the course when we go over reflective DLL injection.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

APC Injection



GUI applications have threads too!

Synchronous and asynchronous execution: what is the difference and how does each come into play? Asynchronous allows an operation to execute in the background while the user application is doing something else instead of being frozen.

APC Injection

Asynchronous procedure call, or APC, is a mechanism that allows an I/O operation to be completed later. When an I/O operation is set to complete immediately, the application will wait and ends up blocking the user from being able to do anything else. This is said to be a synchronous operation. On the other hand, asynchronous operations should allow the user to continue working while the operation is carried out behind the scenes. As an example, there could be a thread running in some GUI process like Notepad and when the user attempts to load a large file, that read operation could happen asynchronously. Should it be synchronous instead, the GUI application might appear to be frozen or not responding. For asynchronous I/O, the thread that kicked off the operation will be notified, or alerted, when the operation is complete via an APC object. The APC objects are placed into a queue properly named the APC queue. Each thread has its own APC queue and when a thread gets its quantum time, a check is done to see if there are any items in the queue, and if there are, they get executed. This means that APC functions, or callbacks, will execute in the context of the thread from which they were requested, and will only be sent when that thread is in an alertable waiting state.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Queueing an APC



QueueUserApc

Return value is DWORD

Used to queue an APC to user thread

```
DWORD
QueueUserAPC(
 PAPCFUNC  pfnAPC,
 HANDLE    hThread,
 ULONG_PTR dwData
);
```


Queueing an APC

Brought to you by the processthreadsapi.h header file, the ***QueueUserApc*** API will add an APC object to the APC queue of the specified thread. Now, let us look at the function declaration. The function has a return type of DWORD, or unsigned long, and has three parameters: pfnAPC, hThread, and dwData.

pfnAPC, is of type PAPCFUNC and should be a pointer to the APC-typed function, or shellcode, that you want to be called when the thread becomes alertable or does some alertable action.

hThread, is of type HANDLE and is a valid handle to the thread of which you are attempting to queue an APC object. The handle should, at a minimum, have the THREAD_SET_CONTEXT access right.

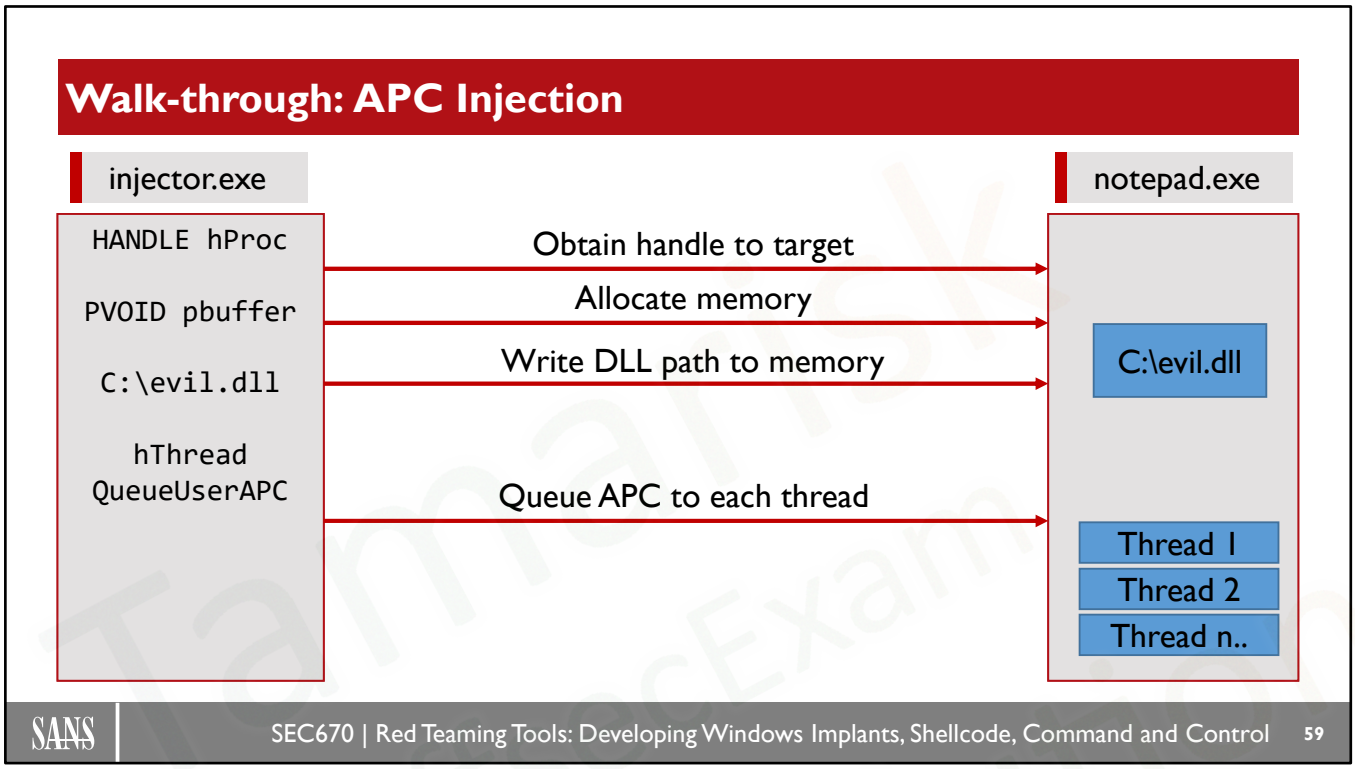
dwData, is of type ULONG_PTR and for our purposes, this can just be NULL. Other use cases, this could be one value that gets passed to the APC function being called.

If the function ever fails, be sure to call ***GetLastError()*** to find out why.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Walk-through: APC Injection

The first step for APC injection is to obtain a handle with the correct permissions to the target process. Once that is obtained, allocate a chunk of memory in the target process so the path to the DLL can be stored. Remember, this must be the full path of the DLL on disk. Before we can queue the APC to each thread, we need to figure out how many threads the process has running. This is when the snapshot comes into play. We can make a list of all threads that belong to the target process and enumerate over that list later in code. If any new threads kick off after we did our enumeration, they would not be discovered. Since the list is now populated with thread IDs, we can iterate over it and obtain a thread handle to each thread. Using that temporary thread handle, we can call the ***QueueUserApc*** function each iteration to queue an APC. What exactly are we going to queue? The APC function will be the address of ***LoadLibraryA*** and the pBuffer will be the argument passed to ***LoadLibraryA***.

With these steps in mind, let's jump over to the source code.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Lab 3.3: APCInjection



Queue an APC to a target thread.

Please refer to the eWorkbook for the details of the lab.

Lab 3.3: APCInjection
Please refer to the eWorkbook for the details of the lab.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

What's the Point?

What's the point?

What's the Point?
The point of this lab was to explore the details of the APC injection method, the APIs involved, manipulating threads, and gaining execution for your code.

Tamarisk OffSecExam Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Thread Hijacking



Taking over a process' thread

Hijacking a thread can be useful when you do not want to target every thread or create a new thread. Hijack the first thread of the target process and redirect its execution to your shellcode.

Thread Hijacking

Hijacking a thread means that we are going to take over and control what it is executing. Threads have a context that the system must keep track of and that is what we are going to modify; the context. The context of a thread cannot be modified until the thread is in a suspended state, so we will have to suspend the thread by calling ***SuspendThread***. Once suspended, we can obtain the context by calling ***GetThreadContext***. When our shellcode, or other payload, has been copied into the target process' address space, we can redirect execution to it and resume the thread. The CONTEXT structure is very complete in the sense that it will capture all registers of the CPU at the state they were in right before the context switch.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Obtaining Context



GetThreadContext

Return value is BOOL

Used to obtain a thread's context

```
BOOL
GetThreadContext(
 HANDLE    hThread,
 LPCONTEXT lpContext
);



typedef struct _CONTEXT {
 [..SNIP..]
 DWORD64 Rip;
};
```


Obtaining Context

Each thread will have a context structure that is unique to each thread. The context struct is fairly large and it must be so that the system can keep track of the context each time a different thread enters its quantum. A prime example for obtaining a thread's context is a debugger. When a thread is being debugged, its context must be made known. For our purposes, we will grab the context and modify the instruction pointer so that when we resume the thread, it will be pointing to our shellcode. Below is a larger snippet of the CONTEXT structure for a 64-bit OS.

```
typedef struct _CONTEXT {
 [...SNIP...]
 DWORD   EFlags;
 DWORD64 Dr0;
 DWORD64 Dr1;
 DWORD64 Dr2;
 DWORD64 Dr3;
 DWORD64 Dr6;
 DWORD64 Dr7;
 DWORD64 Rax;
 DWORD64 Rcx;
 DWORD64 Rdx;
 DWORD64 Rbx;
 DWORD64 Rsp;
 DWORD64 Rbp;
 DWORD64 Rsi;
 DWORD64 Rdi;
 DWORD64 R8;
 DWORD64 R9;
 DWORD64 R10;
```

© 2024 Jonathan Reiter

© SANS Institute 2024

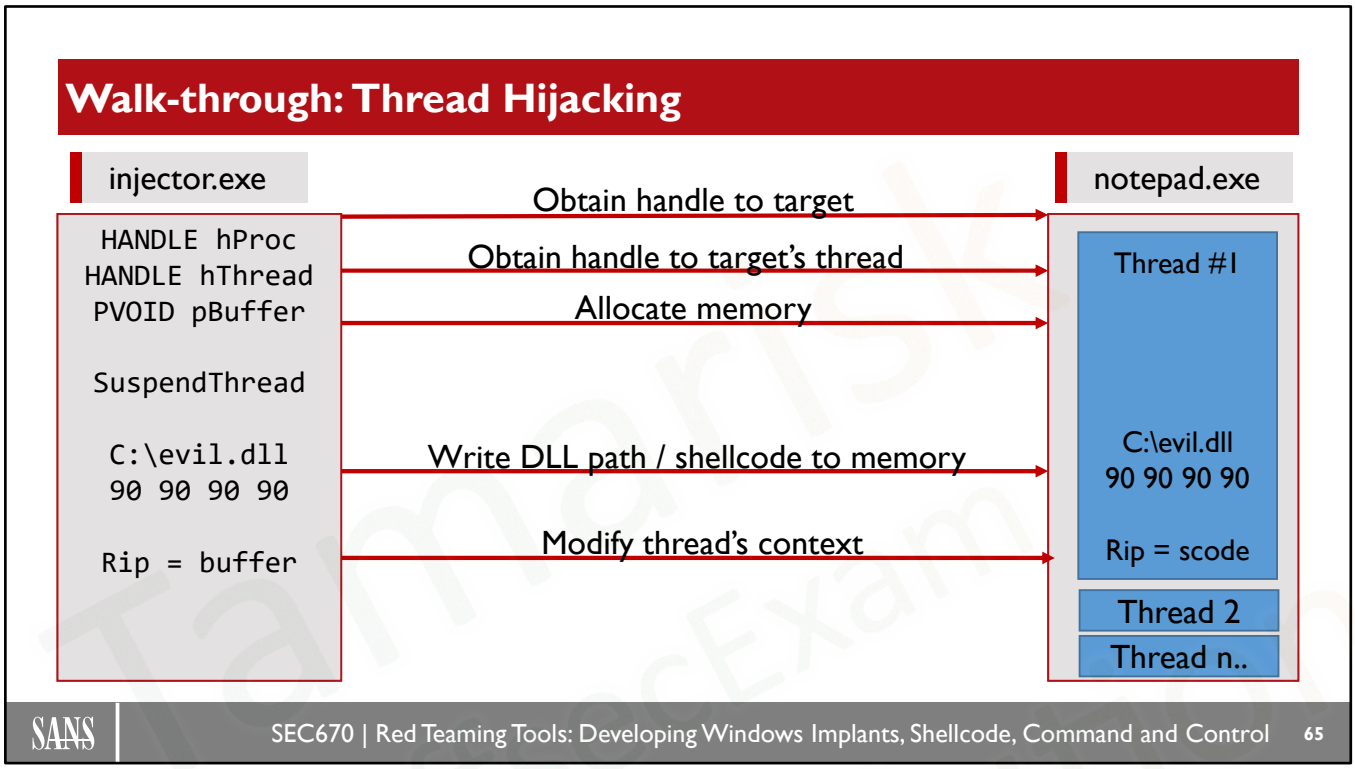

```
DWORD64 R11;
  DWORD64 R12;
  DWORD64 R13;
  DWORD64 R14;
  DWORD64 R15;
  DWORD64 Rip;
  [...SNIP...]
  DWORD64 LastExceptionToRip;
  DWORD64 LastExceptionFromRip;
} CONTEXT, *PCONTEXT;
```

Tamarisk
OffsecExam
Exam solution

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Walk-through: Thread Hijacking

As you perform the various injection methods, you might notice that some portions repeat or carry over from one method to the next. You would be right because they do have very similar portions. The slide here shows a high-level walk-through for hijacking a thread's context. There are two handles that must be obtained: the handle to the target process (hProc) and a handle to a thread (hThread). Both handles must have the minimum permissions necessary to achieve our objectives. For the process, we have virtual memory operations to carry out and we will need to write to the virtual memory. For the thread, we need to suspend it, get its context, update the context, and set its context. The only modification we need to make is changing the instruction pointer, which needs to point to the recently allocated memory (pBuffer). After everything has been completed, we can resume the thread. If we did everything correctly, the shellcode will be executed, and it will load our library.

With these steps in mind, let us jump over to the source code.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Lab 3.4: ThreadHijacker



Hijack execution of a thread

Please refer to the eWorkbook for the details of the lab.

Lab 3.4: ThreadHijacker
Please refer to the eWorkbook for the details of the lab.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

What's the Point?

What's the point?

Tamarisk OffSecExam Exam solution

What's the Point?
The point of this lab was to explore the process of hijacking a thread's context.

© SANS Institute 2024

https://linktr.ee/offsecexam

Process Hollowing



Hollow out the current process' image.

Instead of targeting threads, we can target the executable image itself by replacing it with an image of our own. The current image is removed, or hollowed out, and an image of our choosing is copied over.

Process Hollowing

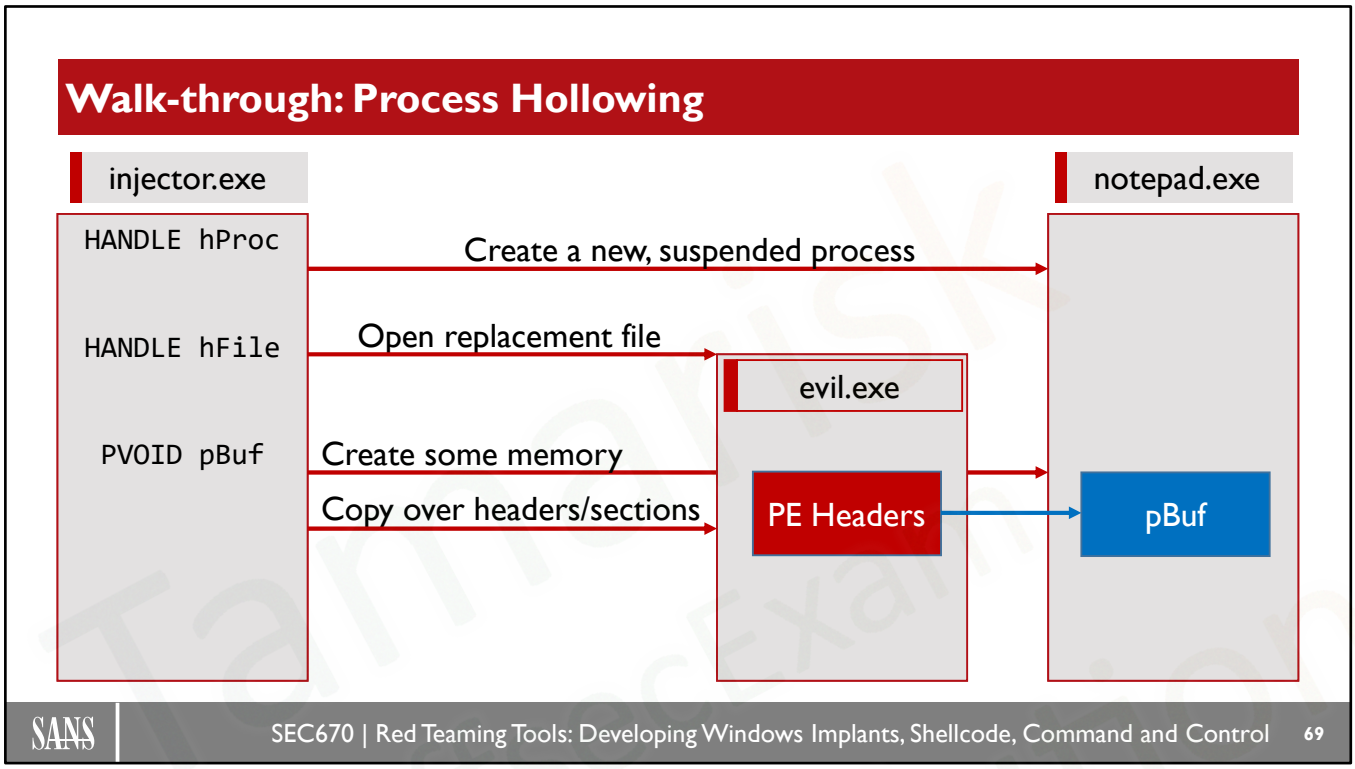
You do not always have to target a process' thread(s) to achieve your injection needs. If you recall, an EXE is mapped into memory where we then call it an image. Once the image is fully mapped into memory, certain portions of the image can be removed without causing any visible changes. Process hollowing does this removal and it removes the current process' image directly from memory. The replacement image will be written to the target process' address space, including all sections, and the main thread will be resumed after modifying its context. We have covered a few methods so far and you might already have some idea as to what APIs might be involved with this method.

Something to think about while we discuss this method: what would happen if you were to create crss.exe, rundll32.exe, or another seemingly legit process in a suspended state and hollow it out?

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Walk-through: Process Hollowing

On the slide is a very high-level overview of the steps involved for hollowing out a process. The first step is to create a new process. This new process is going to be "hollowed out" and will soon contain the PE headers and sections of a different image. The important note here when creating the process is to pass the CREATE_SUSPENDED flag so that the process does not continue its operations. Once the new process is loaded and we have our handle to it, we can allocate a buffer in that process. We do not know how large the buffer should be until we open the replacement image and determine its size. Next, we need to copy over the replacement's PE headers as well as its sections.

Now, even though we have done a lot of work so far, nothing has really changed within the suspended process. If it were to resume execution, our replacement image would not be executed. We can change that by modifying the suspended process' Process Environment Block (PEB). We give it an updated ImageBase address with that of our replacement image. Finally, after updating the thread's context, we can call ***ResumeThread*** and our replacement image should be executed.

With these steps in mind, let us jump over to the source code.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

PE Injection



A PE injected inside another PE

Instead of replacing the PE image of the target process, or hollowing it out, this method simply adds an additional PE image inside. The target process will literally be holding two PE images.

PE Injection

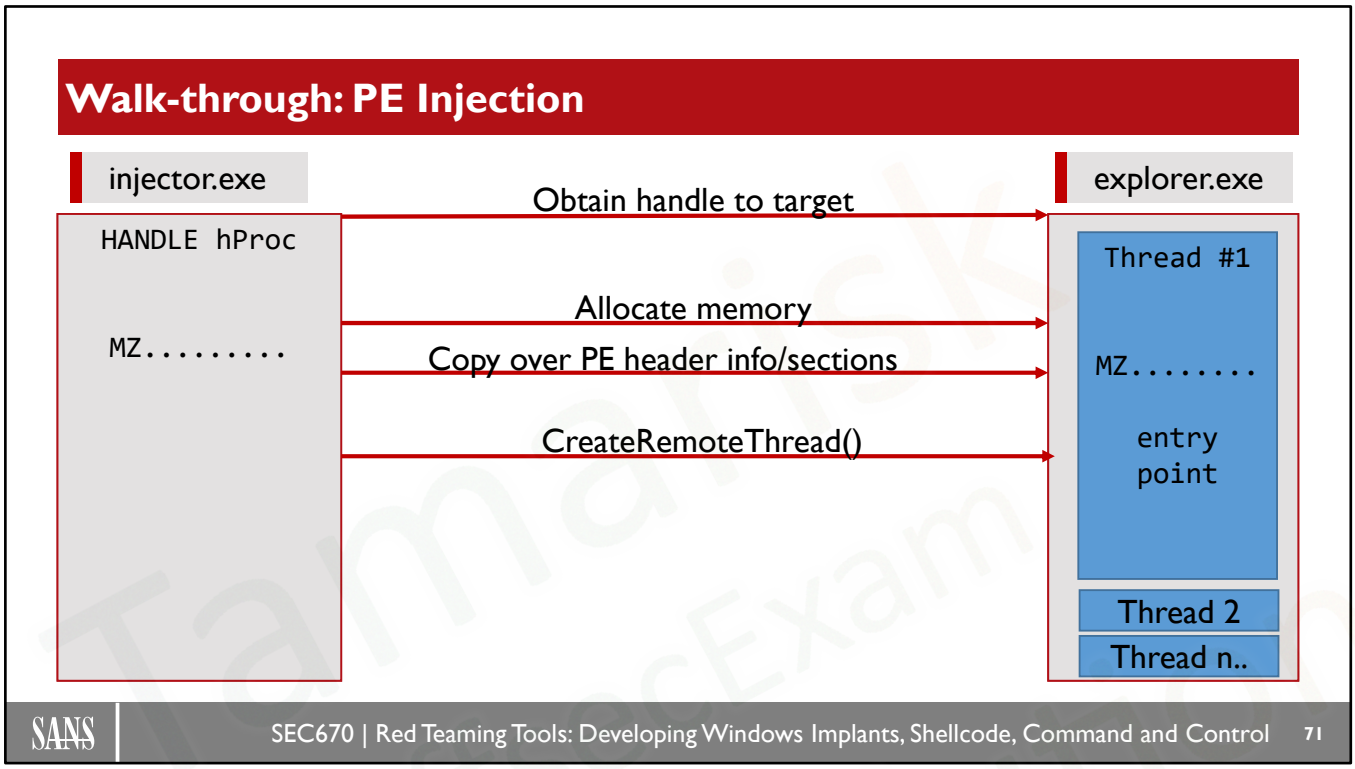
PE injection can sometimes be confused with the process hollowing method because we are injecting a PE file in another process just like with process hollowing. However, this method does not "hollow" out the image from the target process. So, instead of the "hollowing", we are adding another PE image in the target process. The main idea here is that a handle will be obtained to the target process, some virtual memory will be allocated in the target process, our PE data will be copied into the newly allocated memory space, and finally a remote thread will be kicked off to execute our program. The APIs that this method uses are not new, but here they are: ***CreateToolhelp32Snapshot*** to get the PID of the target process, unless manually entering it; ***OpenProcess*** to obtain a handle to the target process; ***GetModuleHandle*** to obtain a handle to own process; ***VirtualAlloc*** and ***VirtualAllocEx*** to handle the allocation of memory in our own process and remote process; ***WriteProcessMemory*** to copy the data over, and ***CreateRemoteThread*** to create a thread in the target process.

Some new ones that you have not seen yet are to enable the debug privileges: ***OpenProcessToken*** to obtain a token handle, ***LookupPrivilegeValue***, and ***AdjustTokenPrivileges***. There is also some PE header parsing that will take place.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Walk-through: PE Injection

We can try to visualize PE injection just like what we have done with the other injection methods thus far. Again, this method is extremely similar to process hollowing with a few differences. With this method, there is no need to have any shellcoding knowledge or experience since we can do what we need to do purely in C/C++ and the Windows APIs. Also, unlike process hollowing, we do not have to create a new process and place it in a suspended state.

- Breaking it down, we have a few of the same steps we have seen already, minus getting a handle to a thread. Assuming we have the target Process ID in hand, we must obtain a handle to the target process using an API like ***OpenProcess***.
- With the newly acquired process handle, we must then allocate a chunk of memory in the target's virtual address space. To determine how much memory to allocate, we must obtain the size of the PE image we are injecting. If you recall, this value is stored in the Size of Image member of the Optional Header structure.
- After the memory has been allocated, we can then proceed with copying over our PE image to include all sections. It would be a good idea to apply fixups to the .reloc section because we are not being loaded by the system loader who normally does that for us.
- We must find the delta of where the base of our image is—basically the address returned to us from the call to the ***VirtualAllocEx*** API—and the preferred base address.
- The image base address is under the Optional Header, so we use that to determine the delta offset. After all fixups or relocations have been applied, we are finally ready to kick it off with the execution of our injected image.
- We take care of that using the ***CreateRemoteThread*** API by passing it the address of our entry point. If everything has gone perfectly smooth, then you will have successfully executed a PE image inside the address space of another PE image.

© SANS Institute 2024

https://linktr.ee/offsecexam

SetWindowsHookEx



GUI applications can be targets as well.

GUI applications might need to hook events so they can properly respond to certain events like mouse clicks or keyboard events. This method requires the target process have the User32.dll module already loaded. This API is very robust and using it for DLL injection is barely scratching the surface for its capabilities.

SetWindowsHookEx

The ***SetWindowsHookEx*** API is a very comprehensive API that can do more than a simple DLL injection. Windows uses it primarily for hooking events like mouse clicks and keyboard events for GUI applications that have focus. There can be an entire chain of events that have been hooked and each one will be iterated over to see who needs to respond to a specific event like someone typing in a Notepad++ window. The API has both an ANSI and Wide char (Unicode) version so you would see ***SetWindowsHookExA*** or ***SetWindowsHookExW*** when looking at online documentation. The hook chain mentioned is what we will be injecting a hook into. More specifically, we will be adding a hook event into an already existing hook chain. The events are system level events, and it is up to Windows to determine what application the hook event procedure is destined. It could be meant for a specific thread or all threads running on the system, or rather the same Desktop session as us—the calling thread.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

SetWindowsHookExA



SetWindowsHookExA

Adds a hook procedure to hook chain

Has a HHOOK return type

```
HHOOK SetWindowsHookExA(
 int       idHook,
 HOOKPROC  lpfn,
 HINSTANCE hMod,
 DWORD     dwThreadId
);

// ... EXAMPLE ...

SetWindowsHookExA(WH_GETMESSAGE,
 EvilHookFunction,
 hEvilDll,
 ThreadId );
```


SetWindowsHookExA

Despite the ***SetWindowsHookEx*** API only having four parameters, they are powerful, as you will see during the lab and via your own reading/experimentation. Let us go ahead and break down these parameters.

idHook, is an *int* type and is used to indicate what hook procedure type will be utilized. The possible values start with WH_* and perhaps the WH stands for Windows Hook. Some options are CALLWNDPROC, which is used to grab messages before the OS pushes them to the appropriate window. GETMESSAGE is used to literally get messages from a message queue. KEYBOARD and MOUSE are the interesting ones because they monitor messages from the keyboard and mouse. This option would give you the chance to intercept and change those messages.

lpfn, of type HOOKPROC, is used to point to the function, or the procedure that is to be called when the message event is generated. This should be pointing to an exported function in a DLL that you make. You could have the hook procedure in the code tied to the current process, but only if the Thread ID is not 0.

hMod, of type HINSTANCE, is used to serve as the handle to the DLL that implements the hook procedure. Again, this procedure would be the one that is exported by a DLL that you make and is pointed to by the previous parameter, *lpfn*.

dwThreadId, of type DWORD, is used to associate the hook to a certain thread. We want to pass 0 here to be associated with all threads that are executing in the same desktop session as us—the calling thread. Because we are using 0 here, we must have the *lpfn* hook procedure in a DLL.

Upon successful execution, the API will return a handle to the hook (HHOOK) that was added to the system's chain of hooks.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Module Summary



Learned process injection comes in many forms; results are really the same

Discussed how some methods have similar techniques

Discussed how some methods serve as an injection method but also offer bonus features

Module Summary

In this module, we covered several methods of injection and even tossed in something extra with the **SetWindowsHookEx** API. If you are a defender taking this class, then it is great to understand how injection is implemented programmatically. It might give you some ideas as how to create detection signatures for this if your organization does not already have some implemented. For the offensive students, it is great to explore the various ways to carry out injection, and as you saw, there is no single way to do anything. Perhaps there are even more methods that are not even known by the public.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Questions



What API allows you to inject a DLL into GUI applications?

A CreateRemoteThread()

B WriteProcessMemory()

C SetWindowsHookEx()

Unit Review Questions
Q: What API allows you to inject a DLL into GUI applications?

A: CreateRemoteThread()

B: WriteProcessMemory()

C: SetWindowsHookEx()

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Answers



What API allows you to inject a DLL into GUI applications?

A CreateRemoteThread()

B WriteProcessMemory()

C SetWindowsHookEx()

Unit Review Answers
Q: What API allows you to inject a DLL into GUI applications?

A: CreateRemoteThread()

B: WriteProcessMemory()

C: SetWindowsHookEx()

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Questions



When hijacking a thread, what construct must be modified?

A Thread state

B Thread context

C Thread priority

Unit Review Questions
Q: When hijacking a thread, what construct must be modified?

A: Thread state

B: Thread context

C: Thread priority

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Answers



When hijacking a thread, what construct must be modified?

A Thread state

B Thread context

C Thread priority

Unit Review Answers
Q: When hijacking a thread, what construct must be modified?

A: Thread state

B: Thread context

C: Thread priority

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Questions



What mechanism allows threads to process routines when it enters its quantum?

A APC queue

B Contexts

C Event objects

Tamarisk OffSecExam Exam solution

Unit Review Questions
Q: What mechanism allows threads to process routines when it enters its quantum?

A: APC queue

B: Context

C: Event objects

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Answers



What mechanism allows threads to process routines when it enters its quantum?

A APC queue

B Contexts

C Event objects

Unit Review Answers
Q: What mechanism allows threads to process routines when it enters its quantum?

A: APC queue

B: Context

C: Event objects

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- **Operational Actions**
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 3

PE Format
- Lab 3.1: GetFunctionAddress

Threads

Injections
- Lab 3.2: ClassicDLLInjection
- Lab 3.3: APCInjection
- Lab 3.4: ThreadHijacker

Escalations
- Lab 3.5: TokenThief

Bootcamp
- Lab 3.6: So, You Think You Can Type
- Lab 3.7: UACBypass-Research
- Lab 3.8: ShadowCraft

In this module, we will discuss and implement various ways to escalate your local privileges.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Objectives

Our objectives for this module are:

- Discuss the reasoning for escalating privileges
- Explore several methods
- Implement a few methods in code

Objectives

The objectives for this module are to talk about the reasoning for escalating your privileges. There could be times when you do not have to operate with higher privileges than what you have already. We will also discuss and explore several methods to programmatically elevate your current privileges.

Tamarisk OffSecExam Exam Solution

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Why Escalate?

? Is there always a requirement to escalate privileges?

Knowing what you can or cannot do with your current level of privileges is important for your tool and the operator using it.

non-admin

admin

Why Escalate?

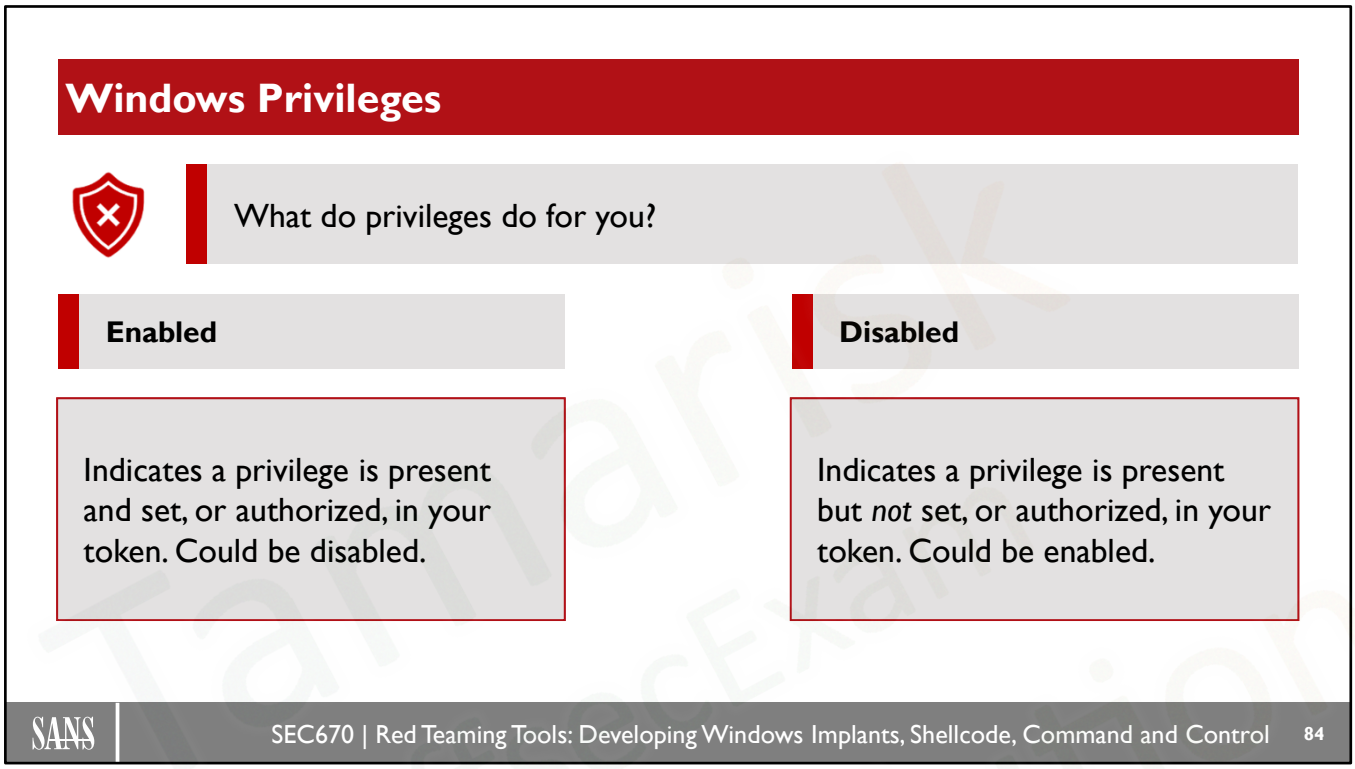
There is only so much that you can do with standard user permissions and accesses. Depending on what the end goal is or what you need to accomplish on the target, you might not even need to escalate your privileges. That might contradict from what others have said and what you might know and that is fine. All I am saying is, why waste your time coding something to be done as Admin when it all could have been done as a regular user. A lot of enumeration and survey tools can run without needing to be Admin and can still pull back enough information about your target to decide if that target is worth bringing down additional tools or capabilities.

Anyway, say you finally made the choice that you must be Admin to accomplish its needs. Now you need to find a way to escalate or gain more privileges. In my opinion, this could be something small like enabling the debug privilege. The debug privilege could come in handy when enumerating processes or opening process handles to them. Some processes will not allow you to open a handle to them because they might belong to a different user, but if you have the debug privilege (*SeDebugPrivilege*, SE_DEBUG_NAME), then you should not have much of a problem. What are privileges anyway?

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Windows Privileges

What is a privilege and what does it do for you? According to Microsoft, "A privilege is the right of an account, such as a user or group account, to perform various system-related operations on the local computer, such as shutting down the system, loading device drivers, or changing the system time." Now that we know what a privilege is, we can discuss what can be done or cannot be done. When a target is first compromised, you would typically find yourself in a standard user account. Very rarely are you exploiting a system and are greeted with domain admin right out of the gate. So, as a standard user with basic privileges, what can be done? What privileges are typically enabled for that account's token? As mentioned previously, when performing process enumeration, you will not be able to obtain a process handle to certain processes if you do not have the *SeDebugPrivilege* enabled. The process manager is the gatekeeper for that action and will happily reject your attempt if the access checks do not pan out. Typically, when Windows creates your token for your user account, you cannot change the privileges that are present, in other words, the privileges that are displayed after issuing the *whoami /priv* command. However, if you make the jump into the kernel, you can do whatever you want and give yourself new privileges. What about stealing privileges in the form of tokens? Let us get ready to dive into that.

Reference:
https://docs.microsoft.com/en-us/windows/win32/secauthz/privileges

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Securable Objects



Objects that have a corresponding security descriptor

Most objects are created at the request of the user: *CreateProcess*, *CreateThread*, *CreateFile*, etc. The functions typically accept a pointer to a SECURITY_ATTRIBUTES structure. There are several object types that can be secured.

files | **processes** | **reg keys** | **threads**

Securable Objects

Remember the section where we were talking about the ***Create**** family of Win32 APIs? It might seem like a long time ago, so here is a refresher. The Create* API family is a way for a user-mode application to ask the kernel to create on object. The kernel does so, after checking a few items, and gives the application a handle to the newly created object. The handle is safe and sound, stored away in the process' handle table. The *SECURITY_ATTRIBUTES* parameter for some of the Create* functions, like ***CreateProcess*** or ***CreateFile***, allow objects to be secured, or hardened, depending on what is passed in as an argument. If you pass in NULL for the SECURITY_ATTRIBUTES parameter, you are saying to the system that the default security descriptor is good enough, and typically the default is good enough. So, when someone attempts to call ***OpenProcess*** on that process, the process manager is going to check to see what the security descriptor is for that process. If everything checks out, then your handle will be returned. If not, you will be greeted with the infamous 0xC0000005 (*ERROR_ACCESS_VIOLATION*) error.

There are several types of objects that you can specify security descriptors for to control access to them. The types noted on the slide are not an all-inclusive list, but rather some of the more common objects you might create and/or come across.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Can I Have a Token?



Access tokens are given after successful authentication.

Each process that you create after logging in will have a primary token, which is tied to you, the user. The security descriptor of the token can and will be checked when attempting to access a securable object.

- `SID/logon SID`
- `Privileges`
- `Default DACL`
- `Primary or impersonation`

Can I Have a Token?

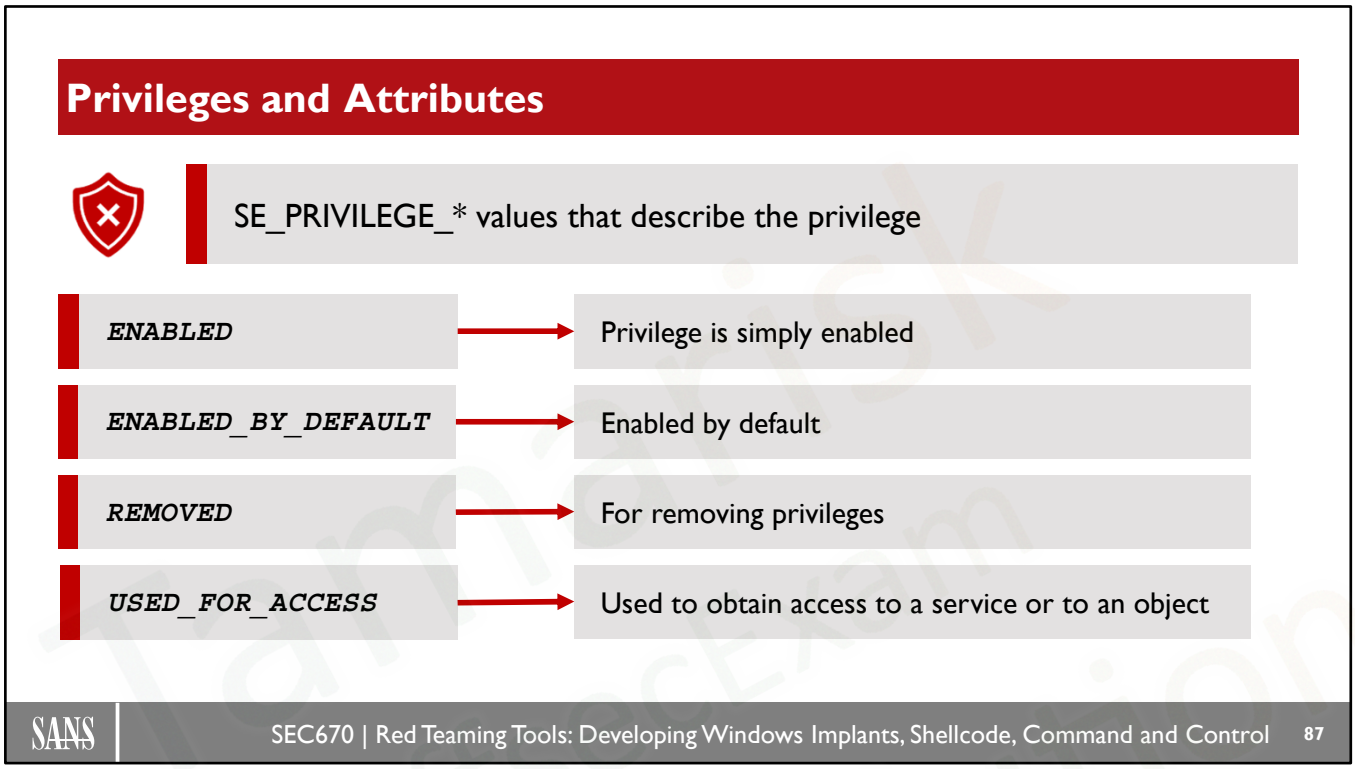
Right when you log on to a system, your user account will be given a token, which holds information like what operations you are or are not allowed to carry out. This is called your access token as it describes your level of access. When it comes to processes that you create, whether being done programmatically or by double-clicking an executable, it should have a primary access token tied to it, which is then tied back to you. There is also an impersonation token, which is created when a process, like a server process, impersonates a client thread to interact with the object using the client's permissions. Impersonation is a way to protect access to sensitive objects since services typically run at a higher integrity level than that of a client process connecting to it.

What is in a token and what is inside a token? There are several items annotated on the slide that can be inside of a token, but of course there are more items that a token holds. For starters, we have the security identifiers (SIDs), which represent the user or group, but also in the token is where the access token originated, and the SID for the primary group. Our favorite header file, winnt.h, defines many *TOKEN_** structures, like *TOKEN_USER*, *TOKEN_ORIGIN*, *TOKEN_PRIVILEGES*, *TOKEN_GROUPS*, and many more. The *TOKEN_PRIVILEGES* structure is an interesting one because it contains an array of privileges for a specific access token. The array entries are of type *LUID_AND_ATTRIBUTES* structures, which, by no surprise, holds the *LUID* and attributes of a privilege. As the name eludes, privileges do have attributes.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Privileges and Attributes

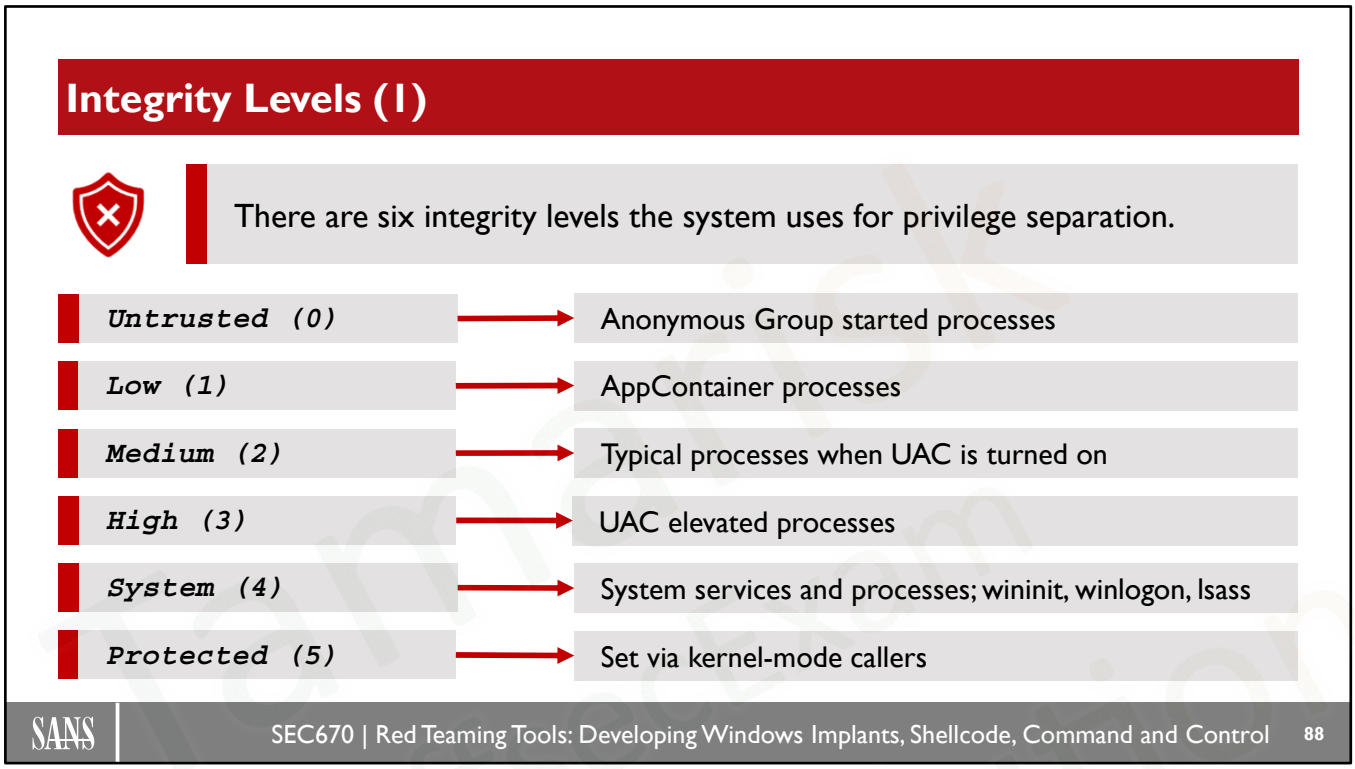
Privileges are what determine if a user, or process, is allowed to carry out an operation within the system. Privileges are not necessarily tied directly to an object, but rather are tied to what can be done. Debuggers like WinDbg Preview that attach to processes must have the *SeDebugPrivilege* privilege, and programs that wish to load a kernel mode driver must have the *SeLoadDriverPrivilege* privilege. All privileges have attributes, and they can be any one of the attributes listed on the slide, as well as a combination of attributes. As mentioned previously, the attributes for a privilege are stored in the *TOKEN_PRIVILEGES* structure; more specifically, the privileges member of the structure, which is a *LUID_AND_ATTRIBUTES* struct. Here is how that struct is defined.

```
typedef struct _LUID_AND_ATTRIBUTES {
  LUID Luid;
  DWORD Attributes;
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

The Luid is a locally unique identifier that is used for identifying privileges unique to the current boot and system. Furthermore, they are 64-bit values, which aids in their uniqueness for that current boot. The Luid has attributes that are comprised of bit flags; 32 of them. The *LUID* structure has two members: *DWORD LowPart* and *LONG HighPart*. The members hold the unsigned low portion of the ID and the signed high portion of the ID, respectively.

© SANS Institute 2024

https://linktr.ee/offsecexam



Integrity Levels (1)

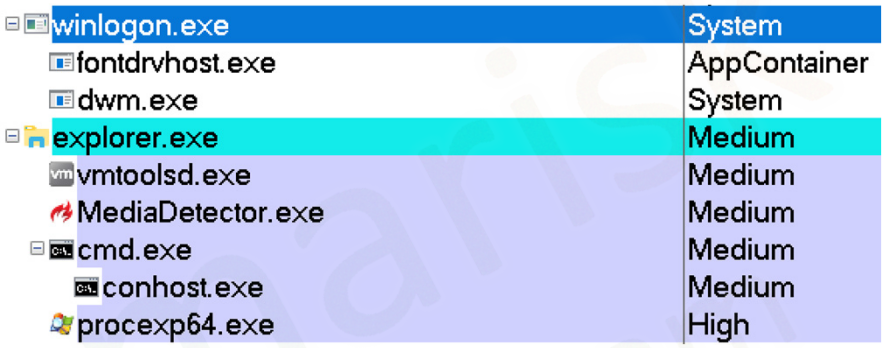
Tokens can have an integrity level (IL) tied to them, which can be queried using the ***GetTokenInformation*** function. There are six levels implemented by the OS and they aid in the separation of privileges, with the lowest level being 0, the Untrusted level. It has the SID S-1-16-0x0 tied to it and this level blocks almost all write access. Processes that are executed from the Anonymous Group typically fall into this level. Levels 1-4 are the levels you mostly read about online, even with MSDN online documentation. The Low-IL is what the AppContainer processes will use when they start. These are the applications built for UWP, or the Universal Windows Platform. Basically, applications that can run universally, like on Xbox and your PC are part of the Universal Windows Platform. You could think of AppContainer as its own IL, but it really boils down to Low-IL. Most objects cannot be written to at Low-IL, like Registry Keys. Now, when you have UAC enabled, the default should be set to Medium-IL, which gives you a little bit more freedom. As a matter of fact, most processes will be launched at this level. At the High-IL is where a process with Administrative rights will show the prompt for the UAC consent box. This could be thought of as an elevated process. The highest integrity level is System-IL where the system services like lsass operate. Many antivirus products have components that run at this IL as well. Level 5, the Protected level, is not enabled by default and is rarely used. The Protected level can only be set from callers in kernel-mode.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Integrity Levels (2)



Integrity Levels (2)

The screenshot on the slide is from Process Explorer, which, as you can see, is running with High-IL. The reason you see this is because I right-clicked on the executable image icon, selected “Run as Administrator” and gave consent to the UAC prompt. You can see other processes and what their corresponding integrity levels are, like explorer.exe and winlogon.exe.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

whoami /priv: non-admin (1)

```
Privilege Name                Description                          State
============================= ==================================== ========
SeShutdownPrivilege           Shut down the system                 Disabled
SeChangeNotifyPrivilege       Bypass traverse checking             Enabled
SeUndockPrivilege             Remove computer from docking station Disabled
SeIncreaseWorkingSetPrivilege Increase a process working set       Disabled
SeTimeZonePrivilege           Change the time zone                 Disabled
```


whoami /priv: non-admin (1)

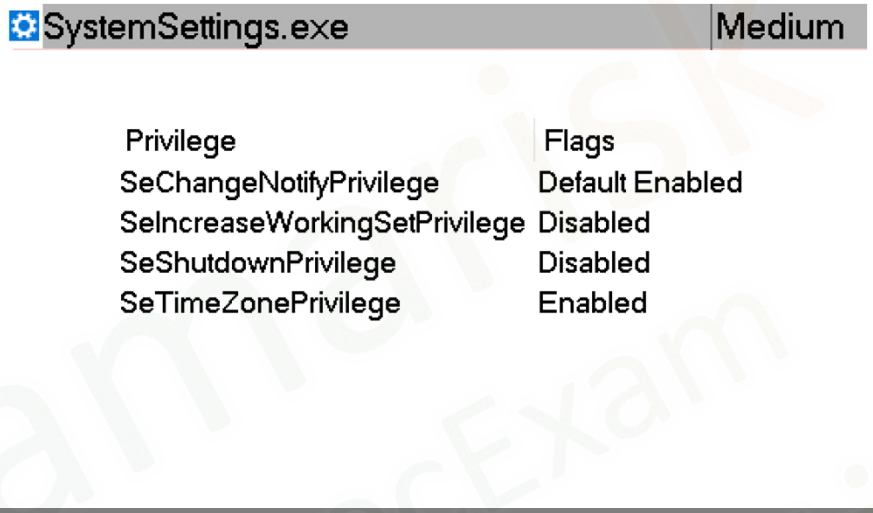
As mentioned previously, privileges are tied to your primary token. The privileges shown on the slide are from the standard user account on the Windows Dev VM. After issuing the *whoami /priv* command, any privileges that are marked as Enabled will be listed in the command's output. The only privilege that is enabled at the moment is *SeChangeNotify*, which, as the name suggests, allows you to traverse different directories to get to files or subdirectories that you can access. The privilege does not give you the access to list contents of every directory you navigate. This privilege should be present and enabled by default for every account. The remainder of the privileges are disabled, but that does not mean they cannot be enabled when you need to perform some operation that the privilege allows. Say, for example, you needed to shut down the system and you already have the privilege to do that action, so it would become enabled. Let us look at another example for changing the system time or the time zone.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

whoami /priv: non-admin (2)



whoami /priv: non-admin (2)

While going through something as simple as changing the time zone, the *SeTimeZonePrivilege* has changed from Disabled to Enabled, as shown on the slide. This was done without any prompts showing up before, during, or after the modification. Another item to note is that the SystemSettings.exe is running with a Medium-IL. There was no need for it to prompt the user for elevation to change the time zone.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

whoami /priv: High Integrity

```
Privilege Name                            Description                                                          State
========================================= ==================================================================== ========
SeIncreaseQuotaPrivilege                  Adjust memory quotas for a process                                   Disabled
SeSecurityPrivilege                       Manage auditing and security log                                     Disabled
SeTakeOwnershipPrivilege                  Take ownership of files or other objects                             Disabled
SeLoadDriverPrivilege                     Load and unload device drivers                                       Disabled
SeSystemProfilePrivilege                  Profile system performance                                           Disabled
SeSystemtimePrivilege                     Change the system time                                               Disabled
SeProfileSingleProcessPrivilege           Profile single process                                               Disabled
SeIncreaseBasePriorityPrivilege           Increase scheduling priority                                         Disabled
SeCreatePagefilePrivilege                 Create a pagefile                                                    Disabled
SeBackupPrivilege                         Back up files and directories                                        Disabled
SeRestorePrivilege                        Restore files and directories                                        Disabled
SeShutdownPrivilege                       Shut down the system                                                 Disabled
SeDebugPrivilege                          Debug programs                                                       Disabled
SeSystemEnvironmentPrivilege              Modify firmware environment values                                   Disabled
SeChangeNotifyPrivilege                   Bypass traverse checking                                             Enabled
SeRemoteShutdownPrivilege                 Force shutdown from a remote system                                  Disabled
SeUndockPrivilege                         Remove computer from docking station                                 Disabled
SeManageVolumePrivilege                   Perform volume maintenance tasks                                     Disabled
SeImpersonatePrivilege                    Impersonate a client after authentication                            Enabled
SeCreateGlobalPrivilege                   Create global objects                                                Enabled
SeIncreaseWorkingSetPrivilege             Increase a process working set                                       Disabled
SeTimeZonePrivilege                       Change the time zone                                                 Disabled
SeCreateSymbolicLinkPrivilege             Create symbolic links                                                Disabled
SeDelegateSessionUserImpersonatePrivilege Obtain an impersonation token for another user in the same session Disabled
```

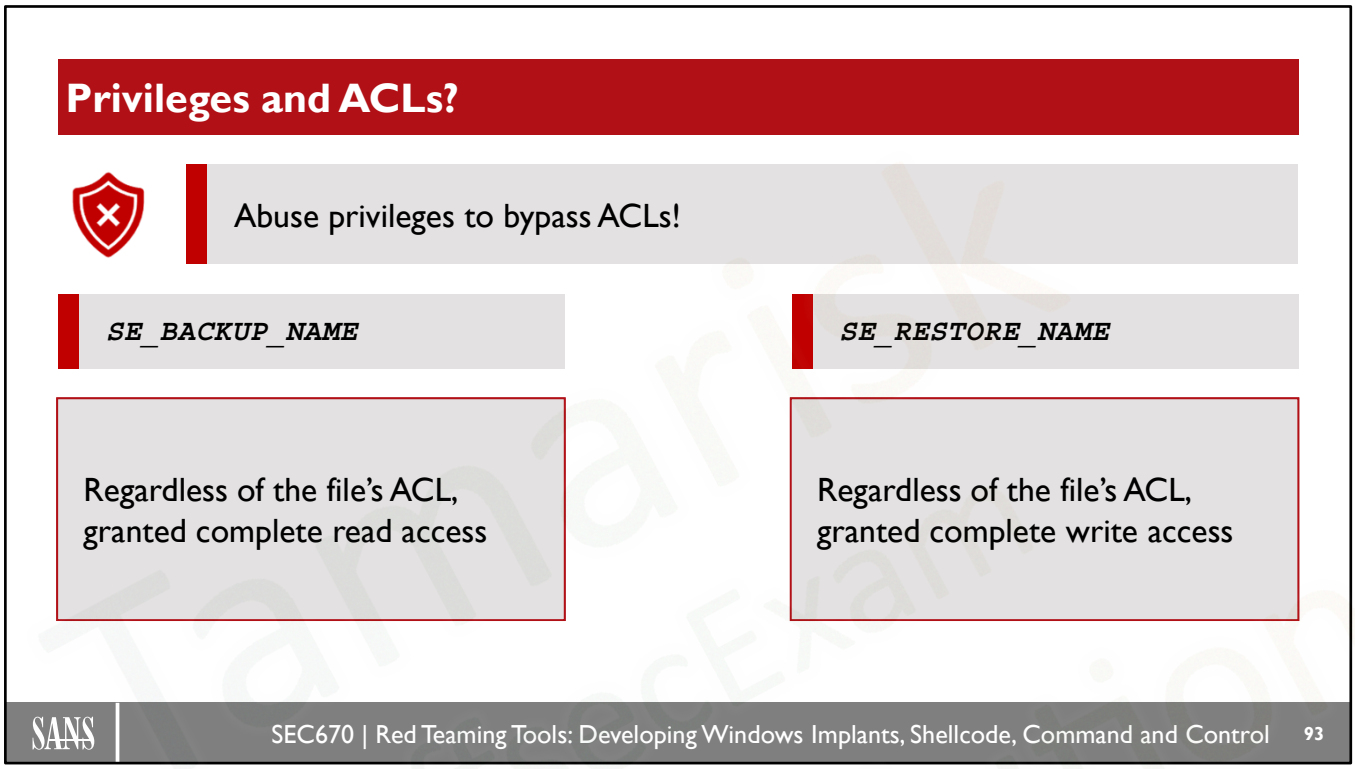

whoami /priv: High Integrity

Let us look at a process with the High-IL. As can be seen on the slide, a High-IL process, command prompt in this instance, has a large increase in the number of privileges that are present. As we have seen already, even though most of the privileges are disabled, they can be enabled on the fly on an as needed basis.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Privileges and ACLs?

There is an interesting tidbit when it comes to privileges. Most privileges allow you to perform some operation, but still only after the system does a privilege check. Well, there are two privileges that bypass that check: *SeBackupPrivilege* and *SeRestorePrivilege*. MSDN describes these two privileges as ones that are used to back up files and directories, and restore files and directories, respectfully. Also, they clearly indicate that with just the *SeBackupPrivilege*, you will be given all the read access you could ever want to the file system without a single check being done. With this privilege, you will have several access rights, including FILE_GENERIC_READ. You might be thinking that while it is nice to read any file, what about being able to write? Well, the *SeRestorePrivilege* has you covered. The restore privilege allows system-wide access to write to any file you would like. Again, it does not matter what the ACL says because this trumps it. As with the backup privilege, this one comes with several access rights, including FILE_GENERIC_WRITE.

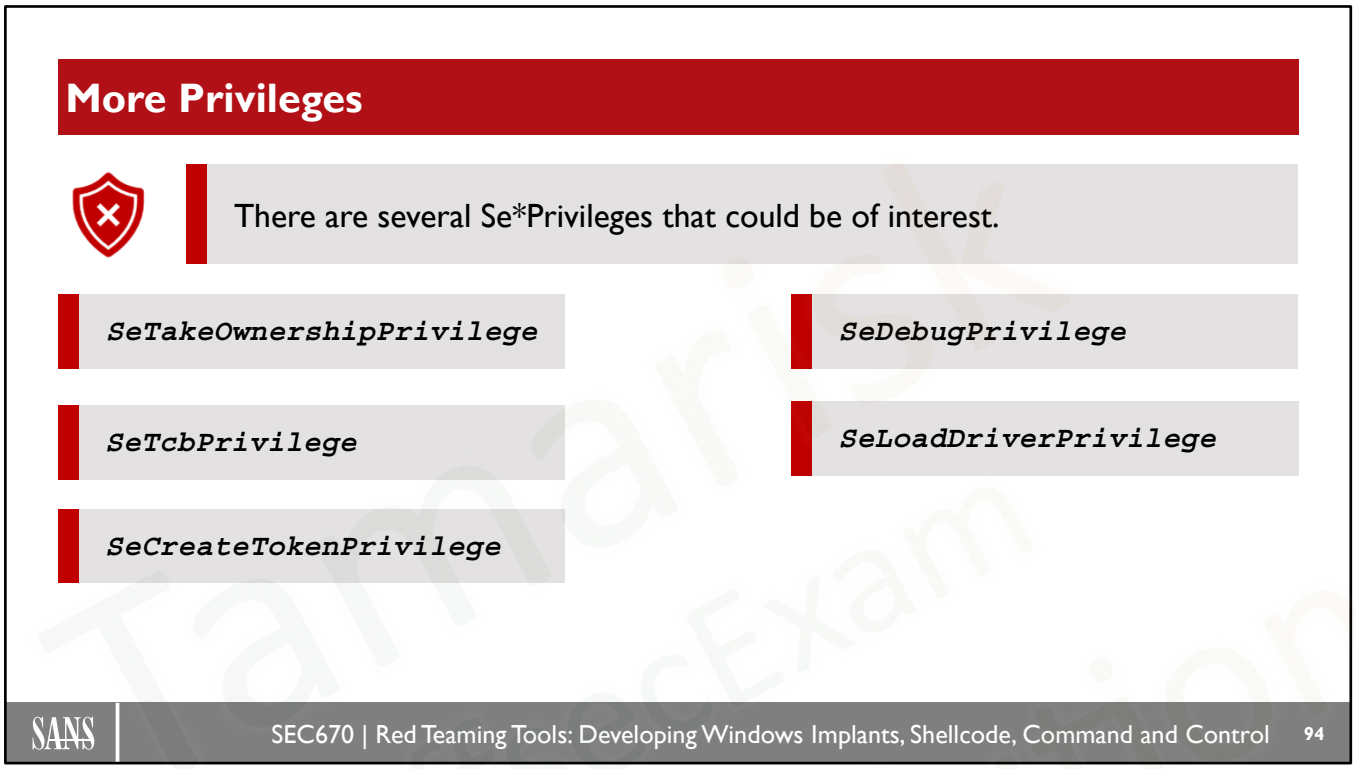
If you would like to explore more privilege constants, you can find them in the winnt.h header file where they are defined.

Reference:
https://docs.microsoft.com/en-us/windows/win32/secauthz/privilege-constants

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



More Privileges

We talked about several privileges up to this point and what they can enable you do, but there are more that would be of interest. The first thing you might be wondering is why none of the privileges noted on the slide are even present for most standard user accounts, and that observation is accurate. The privileges will not be present until you have been able to escalate your privileges. The privileges on the slide can be abused to help you escalate from Admin to SYSTEM.

The first privilege on the slide is *SeTakeOwnershipPrivilege*. This is a powerful privilege because it allows you to take ownership of any securable object you desire. The privilege even allows you to take ownership of securable objects tied to protective processes!

SeTcbPrivilege is described as the Trusted Computing Base (TCB). With this privilege, you become a trusted part of the computer just like how some of the Windows subsystems are.

The *SeCreateTokenPrivilege* allows you to do just what it sounds like it would do—create tokens. You could abuse this to create tokens for arbitrary users on the local system.

We talked about *SeDebugPrivilege* previously, but one item to note is that even with this privilege enabled, you cannot mess around with protected processes.

SeLoadDriverPrivilege does just what it sounds like it would do—load drivers. In the kernel, pretty much everything is trusted, so bringing your own vulnerable driver to the game would greatly boost your efforts.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Privileges: Programmatically



Privileges can be enabled/disabled programmatically.

When you have a set of privileges that are present, but listed as disabled, you can programmatically adjust those privileges to be enabled. The opposite is also true, but why limit yourself?

- `LookupPrivilegeValue`
- `OpenProcessToken`
- `AdjustTokenPrivileges`

Privileges: Programmatically

With the help of the Win32 API, we can create programs that can enable or even disable privileges that are present in our access token. Present means they are listed in the output of the *whoami /priv* command. If you were able to find a UAC bypass to gain an elevated CMD prompt, you would then have many more privileges that would be present, but just not enabled, yet. There are three main APIs involved with this process: ***LookupPrivilegeValue***, ***OpenProcesToken***, and ***AdjustTokenPrivileges***. One common privilege that can be enabled is the *SeDebugPrivilege*. Of course, there are many others that you can select to use, but we will use this one for an example. Let us take a look at the APIs involved and what arguments they will need.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

LookupPrivilegeValue



LookupPrivilegeValue

Gets the current LUID

Has a Boolean return type

```
BOOL LookupPrivilegeValueA(
 _In_opt_ LPCSTR lpSystemName,
 _In_     LPCSTR lpName,
 _Out_    PLUID  lpLuid
);

// EXAMPLE

if ( !LookupPrivilegeValue(...) )
{
  // code here
}
```


LookupPrivilegeValue
Whenever you need to retrieve the locally unique identifier for a privilege constant or privilege name like *SeDebugPrivilege*, this is the function to use. It has a BOOL return type, so it is simple to error check. Simply wrap the function call inside the condition of an if statement and add the code you want to execute inside the body like shown on the slide. Let us take a look at the function's parameters.

lpSystemName, indicates what system to retrieve the Luid for, which indicates that the function should use the local system instead of a remote system. This parameter can optionally be NULL.

lpName, is a pointer to the privilege name as it is defined in the Winnt.h header file. So, you could give it a constant like SE_DEBUG_PRIVILEGE or *SeDebugPrivilege*.

lpLuid, will be a pointer to a variable you use to store what the function finds.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

OpenProcessToken

OpenProcessToken

Obtains a handle to a process' access token

Has a Boolean return type

```
BOOL OpenProcessToken(
 _In_  HANDLE  ProcessHandle,
 _In_  DWORD   DesiredAccess,
 _Out_ PHANDLE TokenHandle
);

// EXAMPLE

if ( !OpenProcessToken(...) )
{
  return FALSE;
}
```


OpenProcessToken

You cannot change any privileges in a token without having a handle to it. The ***OpenProcessToken*** gets you that token handle, when successful, of course. As with the ***LookupPrivilegeValue*** function, it has the same BOOL return type. One example for how to call this function has been noted on the slide. The function arguments have been omitted for space constraints. Speaking of arguments, let us take a look at the parameters.

ProcessHandle, is the handle to a process for which you want a token handle. Be sure that the PROCESS_QUERY_INFORMATION permission is present.

DesiredAccess, is the access mask that you will pass to dictate, or rather ask, what type of access you wish to have for the access token. This is what gets compared with the DACL, the discretionary access control list of the token. This will ultimately be used to approve or deny the requested access type.

TokenHandle, is the pointer to some variable that will hold the handle to this token.

The *TokenHandle* can then be used for the call to ***AdjustTokenPrivileges***.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

AdjustTokenPrivileges



AdjustTokenPrivileges

Enables or disables privileges

Has a Boolean return type

```
BOOL AdjustTokenPrivileges(
 _In_      HANDLE  TokenHandle
 _In_      BOOL    DisableAllPrivileges
 _In_opt_  PTOKEN_PRIVILEGES NewState
 _In_      DWORD   BufferLength
 _Out_opt_ PTOKEN_PRIVILEGES PreviousState
 _Out_opt_ PDWORD  ReturnLength
);

// EXAMPLE

if ( !AdjustTokenPrivileges(...) )
{
  return FALSE;
}
```


AdjustTokenPrivileges
This is the function you can use to enable or disable privileges and is really the last and final step you would have to take if you wanted to enable or even disable a token's privileges. Compared to the previous two functions, ***LookupPrivilegeValue*** and ***OpenProcessToken***, there is a lot more going on with this function. It has a BOOL return type, so we can continue to plug this into an if condition just like the previous two functions. Depending on what you would like to do, it will require a different setup before calling the function. We only care about enabling privileges, so that is the setup of interest.

Let us take a closer look at the function's parameters and types.

TokenHandle, HANDLE, is used for the handle to the access token you obtained by calling ***OpenProcessToken***.

DisableAllPrivileges, BOOL, if set, will disable *all* privileges for the access token. We do not want that, so FALSE it is.

NewState, PTOKEN_PRIVILEGES, is a pointer to a filled-out structure that indicates the Luid and its attributes for the privilege being enabled.

BufferLength, DWORD, should be the size of the TOKEN_PRIVILEGES struct that you made somewhere before this call; sizeof(TOKEN_PRIVILEGES); .

PreviousState, PTOKEN_PRIVILEGES, is a parameter we do not really care about because we are making a new state; (PTOKEN_PRIVILEGES)NULL; .

ReturnLength, PDWORD, is also a parameter we do not care about; (PDWORD)NULL; .

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Stealing Tokens

Source code review!

Stealing Tokens
Time to jump into the source code and explain it before you implement it on your own.

Tamarisk OffSecExam Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Lab 3.5: TokenThief



Nothing more fun than being a token thief

Please refer to the eWorkbook for the details of the lab.

Lab 3.5: TokenThief
Please refer to the eWorkbook for the details of the lab.

Tamarisk OffsecExam Exam solution

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

What's the Point?

What's the point?

What's the Point?
The point of this lab was to explore the steps and APIs involved with stealing a token for escalating privileges.

Tamarisk OffSecExam Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Put Our Service to the Test!



Services and what they can do for your escalation needs

Services are a special kind of process that interact with the SCM. Services do not need a user to login to start as they can be started at boot and run without any user logged on to the system.

- `schedule`
- `EventLog`
- `gupdate`
- `iphlpsvc`
- `BITS`

Put Our Service to the Test!
There are several services that execute behind the scenes, even when there is no user logged on to the system. Services can be configured to start with the system at boot, shortly after boot, when a user logs on, manual start, etc. Regardless of the start trigger, all services must answer to the Service Control Manager (SCM). The installed drivers and services are in a database that is maintained by the SCM. The SCM database can be found under the *HKLM\SYSTEM\CurrentControlSet\Services* registry key with each subkey being the name of the service. Because the SCM acts as a Remote Procedure Call (RPC) server, services can be configured and managed remotely. Depending on what applications your system has installed, you could see some of the services listed on the slide: schedule for the Task Scheduler, EventLog for the Windows Event Log, gupdate for Google Chrome's update service, iphlpsvc for the IP Helper net service, and BITS for the Background Intelligent Transfer Service.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Services: Attributes



Each service will come with some attributes.

Start Type	Service Type	Error Level
Indicates if the service is auto-start, on-demand, or disabled	Indicates if the service runs in its own process or share one with other services	Indicates the severity level if the service or driver fails to start, and what action to take

Services: Attributes

There are certain pieces of information that are good to know about services that I just call service attributes. The service's type is used to indicate to the system, and SCM, if the service is going to execute in its own process, or if it will share a process with some other services. The service's start type indicates how the service, or driver, will begin its execution. Auto-start means the service, or driver, will kick off automatically when the system boots. On-demand means that a user must manually start it; however, if an auto-start service depends on some on-demand service, then the SCM will happily start the on-demand service. Disabled means the service, or driver, cannot start at all. The error control level is used to indicate how bad things will be if a service, or driver, fails to start when it should. Perhaps the system will not boot properly if Driver A does not start before Driver B starts. With this, a predefined action can be specified, which can simply be to attempt to start the service again. Another attribute is the absolute or fully qualified path to the service/driver. Services should have the EXE extension and drivers should have the SYS extension.

The SCM does a few housekeeping steps when it starts a service:

1. Search the SCM database for the account information.
2. Make sure the service account is logged on. If not, log on.
3. Load the user profile.
4. Like process hollowing, the service is started, but in a suspended state.
5. Give the process its logon token and resume execution.

The service should now be ready to go on the system.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Services: Handles



Interacting with the SCM requires handles

There are several handles to objects that are required to be obtained when you want to interact with, modify, delete, or create a new service.

`SCManager` `Service` `Database lock`

Services: Handles

The SCM database can be queried, modified, etc. once you obtain a handle to it. What you really have is a handle to the SCManager object and within that container object are the service objects. To request such a handle, you must call the ***OpenSCManager*** API. With the returned handle, you can then use that to enumerate services, delete services, install services, open services, or lock the services database lock. For those coming from red team or penetration testing team background, enumerating services might be something you do when looking for some low hanging fruit for privilege escalation. Searching for services with unquoted paths could be one such method that boosts you up to SYSTEM as long as the service is running as SYSTEM, which some of them do. Maybe you ran a command like the one below before to find those unquoted service paths. While it is nice to be able to run a Windows command to do this, doing this programmatically would be better.

wmic service get name,pathname,displayname,startmode | findstr /i auto | findstr /i /v "C:\Windows\\" | findstr /i /v ""”

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Services: Enumeration



You cannot find what you do not look for.

Enumerating services is just another part of on target recon with hopes of finding something to exploit. Most tools that deal with Windows services enumerate and query them to show the operator some potential LPE vectors.

EnumServicesStatus

QueryServiceStatus

Services: Enumeration

LPE via services has been a great success for red teamers. Regardless of the tool you may have used in the past, it probably did some form of enumeration of services. To do this, you could manually parse the subkeys from the services registry key and then manually query each one, but a great way is to use the service-specific Win32 APIs that are available, like ***EnumServicesStatus*** or ***EnumServicesStatusEx***. One of the items to look for after enumerating the services would be an unquoted path.

As a refresher, services with an unquoted service path means you are looking for spaces to be present in the absolute path along with missing quotation marks around that path, so something like this: *C:\Users\student\SEC 670\Labs\Day 3\Unquoted Service Paths*. That example path has spaces in it and there are no quotations marks around it. When the system goes to find the EXE, it will stop at the first space that it comes across and use that path to start its search. Sticking with that example path it would become *C:\Users\student\SEC.exe*. Most likely, that EXE would not exist on the system, so it would have to keep searching: *C:\Users\student\SEC 670\Labs\Day.exe.* The system would keep searching until the EXE is found. The idea is to give it something to find early in the search process to force your EXE to run.

You could also take advantage of services with weak permissions. As a reminder, regular users should not be able to modify the configuration information of services. Local administrators should be the only ones with those permissions enabled. Say for an example there was a service with a weak configuration or incorrect permissions—the service might be pointing to some directory where you have write access. The write access could let you drop a DLL to that location and gain execution there. One real world example was the CVE-2019-1322 notice with the UsoSvc service. The service's binary path could be modified to point to your malicious executable, like a msfvenom payload or a custom one you developed.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

EnumServicesStatusEx



EnumServicesStatus

Enumerates services in SCM database

Has a Boolean return type

```
BOOL EnumServicesStatusExA(
 _In_  SC_HANDLE      hSCManager,
 _In_  SC_ENUM_TYPE   InfoLevel,
 _In_  DWORD          dwServiceType,
 _In_  DWORD          dwServiceState,
 _Out_opt_ LPBYTE     lpServices,
 _In_  DWORD          cbBufSize,
 _Out_ LPDWORD        pcbBytesNeeded,
 _Out_ LPDWORD        ServicesReturned,
 _Inout_opt_ LPDWORD  ResumeHandle,
 _In_opt_    LPCSTR   pszGroupName
);
```


EnumServicesStatusEx

Before you can query the status of a service, you must first find a service to query, which can be done using the ***EnumServicesStatusEx*** API. All the services in the SCM database will be enumerated with this API. Let us take a look at the parameters for it.

hSCManager is the SC_HANDLE returned from the ***OpenService*** or ***CreateService*** APIs. The handle should at least have the SC_MANAGER_ENUMERATE_SERVICE access mask.

InfoLevel has only one option, SC_ENUM_PROCESS_INFO.

dwServiceType is a filter for a certain service type. Typically, you would pass in SERVICE_WIN32 here to indicate that you want all service types.

dwServiceState gives you the option to enable finer filtering by not only specifying the service type but also the current state of the service. Typically, you would want to pass in SERVICE_STATE_ALL.

lpServices is the buffer that will wind up holding this information.

cbBufSize is the size of the buffer in bytes. Most times you might not know how many bytes to pass here, but we can find that out. If you really wanted to save another function call, just use 256KB as the size since the buffer cannot be larger than that.

pcbBytesNeeded will let you know if the buffer you passed in is too small to hold everything. If so, the API will fail. Then you can call the API again with the adjust value.

lpServicesReturned is a pointer to a variable that will be used to store how many services were returned.

lpResumeHandle is optional, but it could be used if there is a need to make multiple calls since not all of the information might be given after the first call.

© 2024 Jonathan Reiter

© SANS Institute 2024
https://linktr.ee/offsecexam

pszGroupName could be used to filter the services according to their group name. NULL here means ignore any group a service is a part of and enumerate all groups.

Tamarisk
OffsecExam
Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

QueryServiceStatusEx



QueryServiceStatus

Obtains the status of a service

Has a Boolean return type

```
BOOL QueryServiceStatusEx(
 _In_   SC_HANDLE      hService,
 _In_   SC_STATUS_TYPE InfoLevel,
 _Out_opt_ LPBYTE      lpBuffer,
 _In_   DWORD          cbBufSize,
 _Out_  LPDWORD        pcbBytesNeeded
);
```


QueryServiceStatusEx

After you have obtained a list of the services installed on your target, you would most likely want to query them for more detailed information. To do that, you would call the ***QueryServiceStatusEx*** API. Let us take a look at the five parameters.

hService is the SC_HANDLE from ***OpenService*** or ***CreateService*** APIs. Regardless of the chosen API, the handle must have SERVICE_QUERY_STATUS at a minimum.

InfoLevel indicates the service attributes that you would like to know. The buffer you pass in for *lpBuffer* will be filled out with this information. The only value supported here is SC_STATUS_PROCESS_INFO. Maybe someday in the future this will change.

lpBuffer is a pointer to a buffer that will store the status information of the service.

cbBufSize is the size of the buffer that you are pointing to via *lpBuffer*. This is in bytes too, by the way.

pcbBytesNeeded is a pointer to a variable that will store the number of bytes needed to hold the status information of the service. This is useful when you do not know how many bytes you need to allocate for your *lpBuffer*.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

QueryServiceConfig



QueryServiceConfig

Obtains configuration of a service

Has a Boolean return type

```
BOOL QueryServiceConfigA(
 _In_   SC_HANDLE hService,
 _Out_opt_ LPQUERY_SERVICE_CONFIGA pSvcCfg,
 _In_   DWORD     cbBufSize,
 _Out_  LPDWORD   pcbBytesNeeded
);
```


QueryServiceConfig

The ***QueryServiceConfig*** API will obtain the configuration parameters for whatever service you give it. There is also another version of this API named ***QueryServiceConfig2*** that will obtain the configuration parameters that are optional. This API only has four parameters, so let us dive into them.

hService is the SC_HANDLE from ***OpenService*** or ***CreateService*** APIs. Regardless of the chosen API, the handle must have SERVICE_QUERY_STATUS at a minimum.

lpServiceConfig, *pSvcCfg* on the slide for brevity, is a pointer to the buffer that will be used to store the configuration information about the service. As can be seen by the type, this will be in the format of the QUERY_SERVICE_CONFIG structure, which is made up of members like *dwServiceType*, *dwStartType*, *dwErrorControl*, *lpBinaryPathName*, etc. The binary path name could be of interest if the path has spaces and is missing quotes around it.

cbBufSize, just like for the other APIs mentioned thus far, is the size of the buffer that is pointed to by the *lpServiceConfig* parameter, in bytes, of course. The buffer need not be larger than 8KB as this is the max size.

pcbBytesNeeded: If the API would ever fail with ERROR_INSUFFICIENT_BUFFER, this pointer to a variable will end up holding the number of bytes needed so that the buffer pointed to by the *lpServiceConfig* parameter could be sized accordingly for a secondary call.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

ChangeServiceConfig



ChangeServiceConfig

Modifies a service's configuration

Has a Boolean return type

```
BOOL ChangeServiceConfigA(
 _In_     SC_HANDLE hService,
 _In_     DWORD     dwServiceType,
 _In_     DWORD     dwStartType,
 _In_     DWORD     dwErrorControl,
 _In_opt_  LPCSTR   lpBinaryPathName,
 _In_opt_  LPCSTR   lpLoadOrderGroup,
 _Out_opt_ LPDWORD  lpdwTagId,
 _In_opt_  LPCSTR   lpDependencies,
 _In_opt_  LPCSTR   lpServiceStartName,
 _In_opt_  LPCSTR   lpPassword,
 _In_opt_  LPCSTR   lpDisplayName
);
```


ChangeServiceConfig

The ***ChangeServiceConfig*** API is very similar to the ***CreateService*** API that will be introduced later. The purpose of this API is to make a change to the configuration of an already existing service. This function can be used to change almost any configuration except the service's name. After you have enumerated all the services and decided what service you would like to modify, this is the function you use to get it done. Perhaps one example could be that you found a service that points to a writable location, and you change the binary path for it; a simple way to gain execution. With this API you can keep everything the same and only change the *lpBinaryPathName*. The parameters are, for the most part, the same as the ***CreateService*** API. The major differences here are that if you are not going to make a change to one of the parameters, you either pass in SERVICE_NO_CHANGE or NULL. The SERVICE_NO_CHANGE can only be used for the first three parameters and then NULL must be used for the remainder.

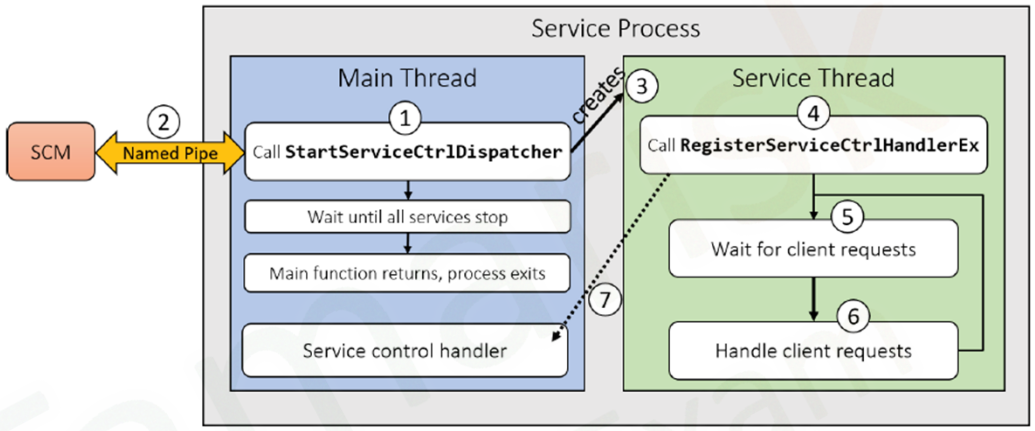
There is also the ***ChangeServiceConfig2*** API, which allows you to change a service's trigger. Triggers are an interesting concept because they can be used to start or stop your service. An example trigger could be when an IP address is first pulled to a NIC for availability or when the NIC loses its availability. Another trigger for a service could be when the computer joins an active directory domain. Your malicious service could also have a trigger, which is something that is up to you since you are the developer. You just have to do what makes sense and what would meet the requirements for your end customer—say, a red team operator.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Services: Creation (1)



Services: Creation (1)
For us to programmatically create and eventually install a service, there are a few requirements your code must meet.

At a high level, it must have a service entry point, a service ***ServiceMain*** function, and a service control handler function. Our service will not be sharing its address space with other services, so it will be of type SERVICE_WIN32_OWN_PROCESS. The ***ServiceMain*** function is the main function you might already know. If not, the ***ServiceMain*** function has several important items to carry out when it is called by the service control dispatcher. One item is that it must call ***StartServiceControlDispatcher*** so that we can be connected to the SCM via a named pipe. The next item we must take care of is call the ***RegisterServiceCtrlHandler*** function so that it can register a Handler function that will be called when control requests come to the service. Some of the control requests that could come into the service are start, stop, pause, etc. Finally, we must call the ***SetServiceStatus*** function to report to the SCM that we are running, SERVICE_RUNNING. The Handler function is required because it is the function that is called by the control dispatcher when the service receives a control request. The function given is executed in the context of the control dispatcher.

Reference:
Pavel Yosifovich. *Windows 10 System Programming, Part 2*. Pavel Yosifovich, 2022.

© SANS Institute 2024

https://linktr.ee/offsecexam

Services: Creation (2)



Small code snippet for a service's main function

```
VOID WINAPI EvilMain(...);

INT main() {
 CHAR ServiceName[] = "notEvil";
 SERVICE_TABLE_ENTRY table[] = {
  { ServiceName, EvilMain },
  { NULL, NULL }                          // the end of the array
 };
 if (!StartServiceCtrDispatcher(table)) { return 1;}
}
```


Services: Creation (2)

As mentioned previously, the ***main*** function has the responsibility of calling the ***StartServiceCtrlDispatcher*** function. If other services are going to share this process' address space, the table is passed as an argument to the function, so SCM knows about them. The table array holds SERVICE_TABLE_ENTRY structs that have two members: the service name (LPSTR) and a pointer to the service's main function (LPSERVICE_MAIN_FUNCTION). The *lpServiceName* struct member is the name of the service that you would see listed in Task Manager or in the registry. The *lpServiceProc* is the pointer to the ***ServiceMain*** function that has a very specific signature, or prototype.

VOID LpServiceNameFunction(
DWORD dwNumServicesArgs, // the number of arguments that are in the lpServiceArgVectors array
LPSTR lpServiceArgVectors // null-terminated strings that are passed into the service via the **StartService** function call when the service was started.*
);

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Services: Creation (3)



Code snippet for ServiceMain()

```
SERVICE_STATUS g_ServiceStatus;
SERVICE_STATUS_HANDLE g_ServiceStatusHandle;
HANDLE g_ServiceStopEventHandle = NULL;

EvilMain(...){
 g_ServiceStatusHandle = RegisterServiceCtrlHandlerEx(...);
 g_ServiceStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
 g_ServiceStatus.dwServiceSpecificExitCode = 0;
 g_ServiceStatus.dwWaitHint = 500;
}
```


Services: Creation (3)
The ***ServiceMain*** function, ***EvilMain*** in our case, is where most of the service initialization takes place. There are some global variables near the top of the slide that are filled out and set like the g_ServiceStatusHandle and the g_ServiceStatus. Below is the function prototype for the ***RegisterServiceCtrlHandlerExA*** function.

SERVICE_STATUS_HANDLE
WINAPI
RegisterServiceCtrlHandlerExA(
  LPCSTR lpServiceName, // the name of the service that is executed by the calling thread and the name passed to ***CreateService***
  LPHANDLER_FUNCTION_EX lpHandlerProc, // the handler function to be registered with the SCM. Below is the signature for the handler.
  LPVOID lpContext // simply a pointer to some user-defined data that could be used to identify the service if it is running in a shared process with other services
);

This is the signature for a handler function:

DWORD
WINAPI
HandlerEx(
  DWORD dwControl,
  DWORD dwEventType,
  LPVOID lpEventData,
  LPVOID lpContext
);

© SANS Institute 2024

https://linktr.ee/offsecexam

If you recall, the service type of SERVICE_WIN32_OWN_PROCESS indicates that we do not intend to be part of a shared process. This is particularly important because if your malicious service is going to be shared in a process with other services, what happens when another service crashes? Your service will crash right along with it, thus it is best to be in your own process. As a fail-safe, you could configure your service failure action to simply restart the service and be just fine.

Tamarisk
OffsecExam
Exam solution

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Services: Installation



sc.exe is a common tool for service installation.

It has been common for users to use the service-related PowerShell cmdlets or the sc.exe built-in utility for installing and querying services. Regardless of the tool being used, the underlying API for installing a service is the *CreateService()* API.

Services: Installation

A very common tool for installing services is the sc.exe (service create) built-in utility. The name passed to this should also be the same name passed in your code's call to ***RegsiterServiceCtrlHandlerEx***. For example, if your service is going to be named *notevil*, then at the command line you would use *sc screate notevil binPath= "c:\path\to\your\notevil.exe"*. Later, after successful installation, you can query your service using *sc query notevil*. The sc.exe utility, along with PowerShell cmdlets, uses Windows APIs under the hood to perform their operations. The installing a service capability of those tools utilize APIs like the ***CreateService*** API, which has whopping 13 arguments! It is a powerful function and a complicated one behind the scenes. Thankfully, there are no structures that must be set up and passed into it. We will go into more details about that function next. In the meantime, it is important to know the difference between the code for your actual service, and the service control program, or the service installer program. Typically, your program will not install itself as a service, although it can. It would be best for your implant to install a separate binary as a service; one that could be used for persistence, even. Metasploit's Meterpreter's persistent service is a good example and is freely available for you to see on GitHub.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

CreateService



CreateService

Creates service object

Has a SC_HANDLE return type

```
SC_HANDLE CreateServiceA(
  _In_      SC_HANDLE hSCManager,
  _In_      LPCSTR    lpServiceName,
  _In_opt_  LPCSTR    lpDisplayName,
  _In_      DWORD     dwDesiredAccess,
  _In_      DWORD     dwServiceType,
  _In_      DWORD     dwStartType,
  _In_      DWORD     dwErrorControl,
  _In_opt_  LPCSTR    lpBinaryPathName,
  _In_opt_  LPCSTR    lpLoadOrderGroup,
  _Out_opt_ LPDWORD   lpdwTagId,
  _In_opt_  LPCSTR    lpDependencies,
  _In_opt_  LPCSTR    lpServiceStartName,
  _In_opt_  LPCSTR    lpPassword
);
```


CreateService

MSDN describes the ***CreateService*** API as one that will create a service object when called. It will also add the service to the SCM database, which is most likely the default database for the local system. Let us tackle some of these parameters and describe what they do.

First up is *hSCManager* of type SC_HANDLE, which indicates that it needs a handle—one you would have obtained from calling the ***OpenSCManager*** API. The ***OpenSCManager*** API accepts three arguments, the name of the machine to connect to, the name of the SCM database to connect, and finally, the desired access to the database.

lpServiceName, is the unique name you would see in the registry. Here, you are limited to 256 characters.

lpDisplayName is what is known as the friendly name and is what you would normally see in the Services tab in the Task Manager. This string is also limited to 256 characters.

dwDesiredAccess is the access the returned service handle will require. So, to start a service by calling the ***StartService*** API, you would need to specify the SERVICE_START mask.

dwServiceType is the type of service that will be created. For our purposes this will be WIN32_OWN_PROCESS (0x10).

dwStartType indicates when the service should be started. Services can have AUTO_START (2), DEMAND_START (3), or DISABLED (4).

dwErrorControl dictates what is to be done when service initialization fails. It can be IGNORE (0), NORMAL (1), SEVERE (2), or CRITICAL (3).

© 2024 Jonathan Reiter

© SANS Institute 2024

lpBinaryPathName must be the full path where the executable is located. Command-line arguments can also be passed in here after the executable's name.

lpLoadOrderGroup is optional, so NULL is just fine here.

lpdwTagId is only for kernel drivers and as this is not a kernel class, we do not need to worry about this one.

lpDependencies is an optional list of strings naming other services that this service depends on for successful initialization.

lpServiceStartName is the account that this service should execute under.

lpPassword would be for the password to the given user account.

Tamarisk OffsecExam Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Pipes!



One of many method of interprocess communications (IPC)

Application data sharing and communications, together, are known as IPC. Pipes are one such mechanism of IPC. They allow for two-way communications either with parent-child processes, or processes with no relationship.

Anonymous pipes

Named pipes

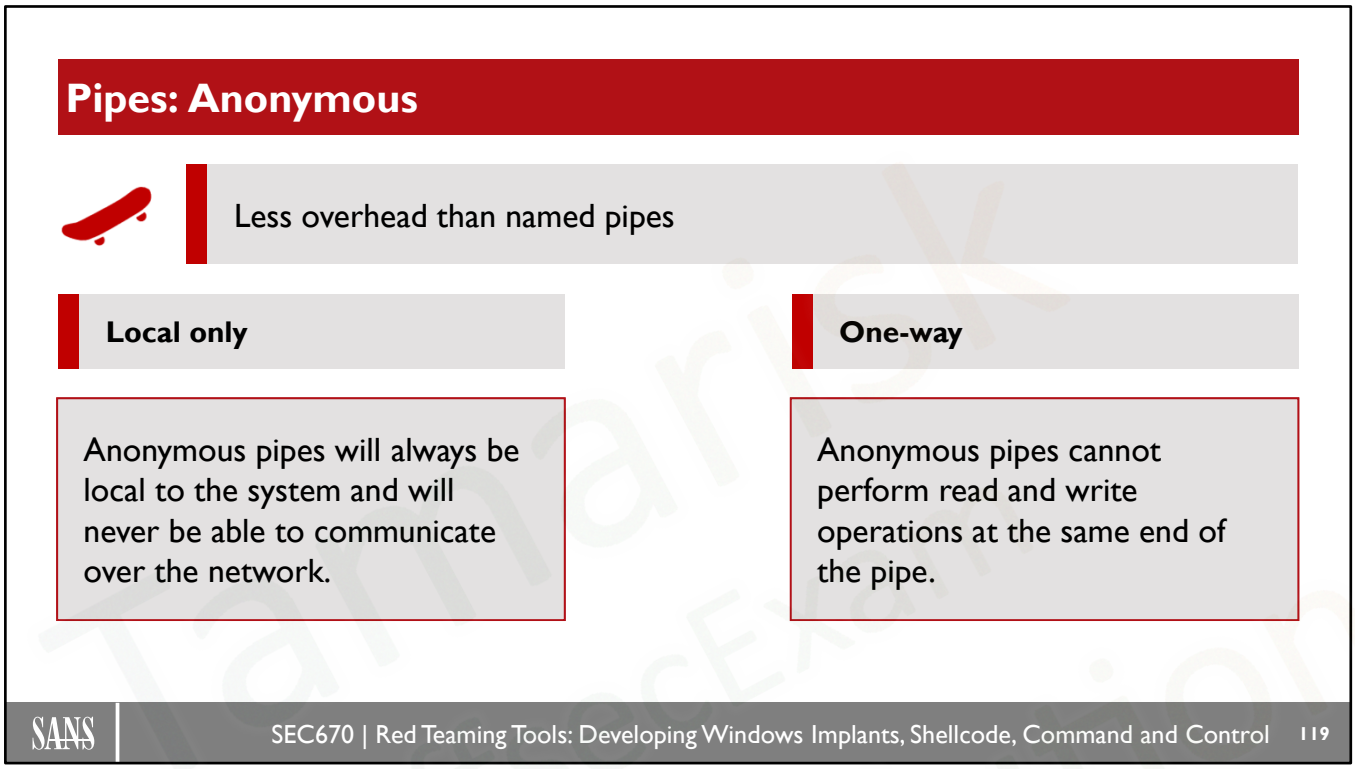
Pipes!

There are times when processes must communicate with each other or share some data. Windows provides a mechanism that allows the communication and sharing to take place. Such a mechanism is called interprocess communications, or IPC for short. Even if you have not heard of IPC before now, you have definitely used it at least once if you have ever cut, copy, or pasted anything. The clipboard could arguably be one of the most commonly used IPC mechanisms. Some of the other mechanisms are COM, the Component Object Model, which is the backbone to how Excel data can be embedded in Word documents. Another mechanism is file mapping, which has many uses! Windows Sockets are yet another example of IPC. The IPC mechanism we are focusing on at the moment is pipes. Pipes can allow communication to flow in both directions called duplex operations. Think of a conduit—a pipe that an electrician might use to run wires through. Windows pipes are very similar: they have two ends, and data can flow through them. The end points of the pipes are where the reading and writing of data takes place. A duplex pipe allows the read and write operations to take place at the same end of the pipe. A one-way pipe is where a write operation takes place at one end and the read operation takes place at the other end. Pipes come in two flavors: anonymous and named.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Pipes: Anonymous

Anonymous pipes are pipes without names, which have certain restrictions that named pipes do not have. For instance, they cannot be used to communicate over the network, and they cannot be used to communicate with processes that are not related to them (parent/child). Anonymous pipes come to play when you need a quick IPC mechanism between your process and a child process. As mentioned previously, your child process could take commands and also send the output of commands back to the parent process. This flow can happen over anonymous pipes with the redirection of standard in and standard out. When a process creates a pipe, that process is the pipe server. Pipe clients would then connect to it and reading and writing to the pipes can take place.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Creating Anonymous Pipes



`CreatePipe`

Used to create anonymous pipe

Has a Boolean return type

```
BOOL
CreatePipe(
 PHANDLE hReadPipe,
 PHANDLE hWritePipe,
 LPSECURITY_ATTRIBUTES lpSecAttr
 DWORD nSize
);
```

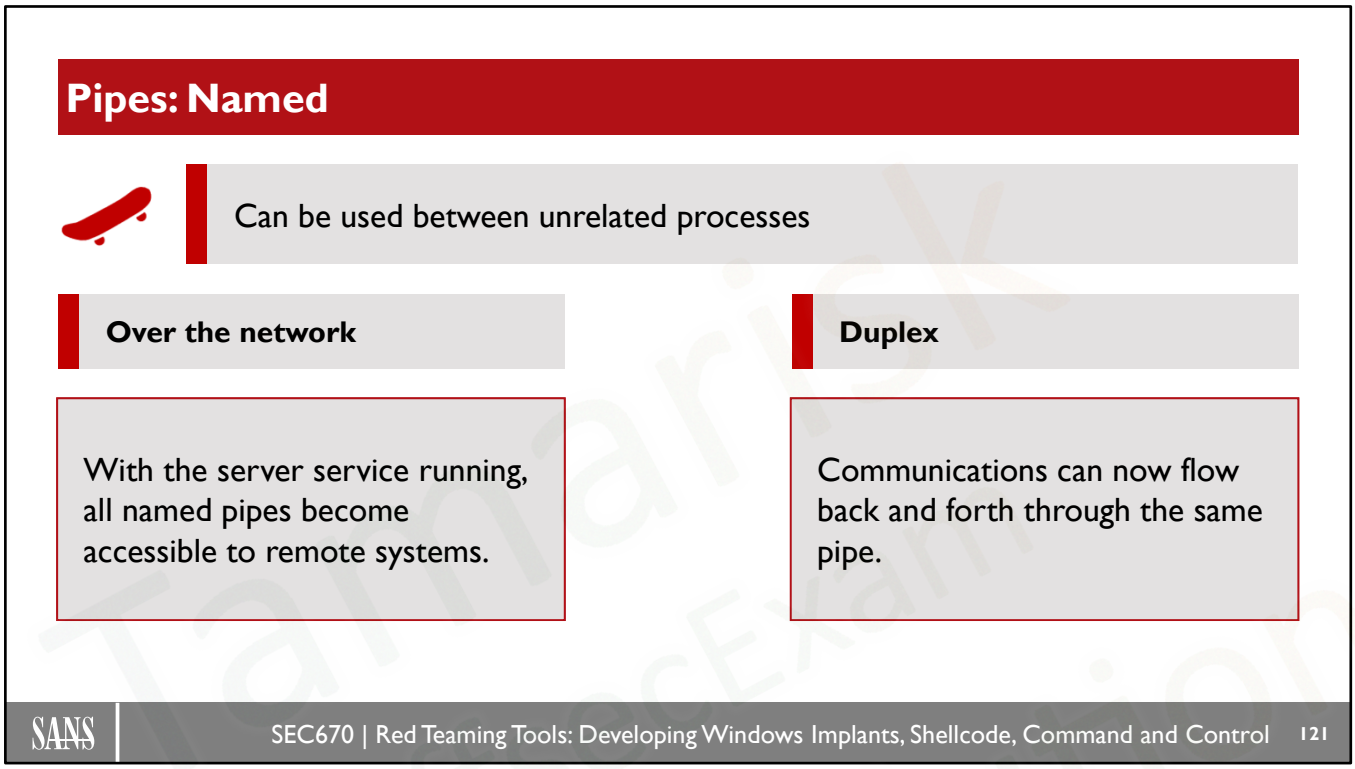

Creating Anonymous Pipes

Anonymous pipes are created using the ***CreatePipe*** function. The function will return handles to the read end of the pipe as well as the write end of the pipe. The handle for the read end of the pipe will naturally only have read access. The same thing goes for the write end of the pipe—it will only have write access. The process calling this function is the pipe server and clients will connect to it. To do so, they will need to be allowed to access the proper handle, which is easily done using handle inheritance. The child process, or the client, can either be sent the handle for the read or write ends of the pipe. If the pipe server sends the read handle, the client would have to use the ***ReadFile*** function. The ***ReadFile*** function will return as soon as data has been written to the pipe using the ***WriteFile*** function. Inheriting handles is just one method a process can be given pipe handles. The other method is using a function named ***DuplicateHandle***, which basically makes a copy of a handle that could be shared using shared memory.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam



Pipes: Named

Named pipes are pipes that have a name associated with them, and there are several advantages that named pipes have over anonymous pipes. Named pipes can be one directional like anonymous pipes, but they can also be duplex pipes. Duplex pipes are great for pipe servers that will communicate with one or many pipe clients. This also means that any process could become the pipe server and the pipe client. Depending on the security attributes you choose for your pipe, any process could be allowed to interact with your named pipe. This is so much more convenient than having to worry about sharing handles to a pipe you created. Perhaps one of the bigger advantages is that named pipes can be used to communicate with remote processes. Should you have the server service running, then your named pipes will be remotely accessible.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Creating Named Pipes



`CreateNamedPipe`

Used to create named pipes

Has a HANDLE return type

```
HANDLE
CreateNamedPipe(
 LPCSTR lpName,
 DWORD  dwOpenMode,
 DWORD  dwPipeMode,
 DWORD  nMaxInstances,
 DWORD  nOutBufferSize,
 DWORD  nInBufferSize,
 DWORD  nDefaultTimeOut,
 LPSECURITY_ATTRIBUTES lpSecAttr
);
```


Creating Named Pipes

Named pipes are created using the ***CreateNamedPipe*** function. Instead of returning handles to the ends of pipes, it will create an instance of the pipe. The process that calls ***CreateNamedPipe*** is the pipe server and can use the function to create the first instance of the pipe or, with an existing pipe name, it can create another instance. The pipe server can later call the ***ConnectNamedPipe*** function to wait for a pipe client to connect to the pipe instance. The client will use the ***CreateFile*** function or the ***CallNamedPipe*** function to connect. Clients can use the following format when connecting to a named pipe: *ComputerName*\pipe*PipeName*. The *ComputerName* is going to be the name of the computer you wish to connect. Of course, this can also be the local computer. The *PipeName* is the unique name for the pipe and the name can contain almost any character except a backslash.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Creating Pipes

Source code review!

Creating Pipes
Time to jump into the source code and explain it before you implement it on your own.

Tamarisk OffSecExam Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

UAC Me, Now You Don't



UAC is not a security boundary.

Processes that could lead to a system compromise typically will run with a Low-IL. Browsers often do this in case you browse to a malicious site.

Processes that typically run with High-IL are ones that have system-wide configurations or operations like Lsass.exe and Winlogon.exe.

UAC Me, Now You Don't

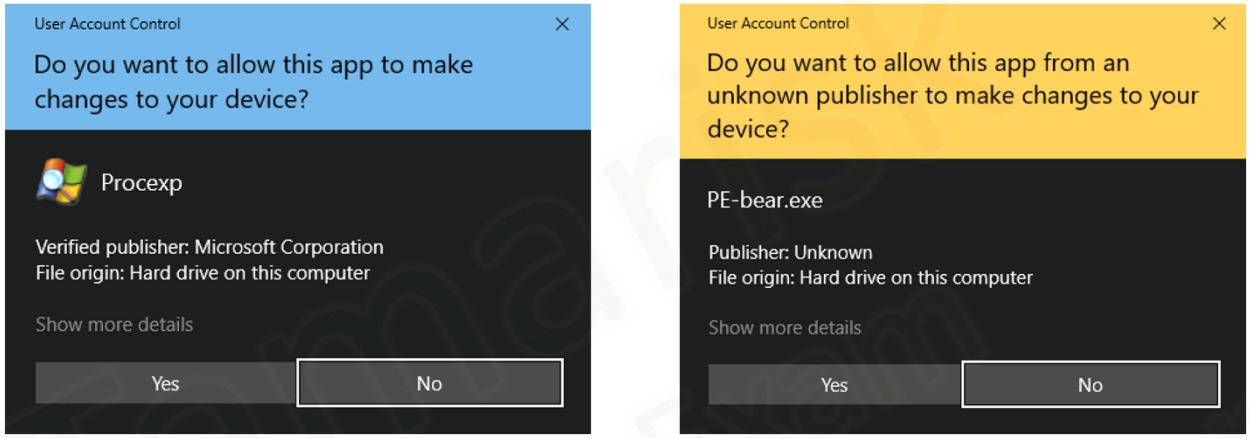
Many people interpret UAC as a security boundary, thinking that UAC is a mechanism that is protecting them. UAC, to me, is really just an annoying method of protecting you from yourself but definitely not a true security boundary. There are a couple of positive items about UAC, so it is not all that bad. UAC gives you the convenience of running an application as Admin without having to switch user accounts. Another convenience is that you no longer have to issue the RunAs command if you want to execute a command as a different user. Perhaps one final positive note about UAC is that it can possibly mitigate how effective malware is on a system. There are a number of UAC bypass projects out there and it is possible to find the resources to bypass UAC, as we will see later.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

UAC: Elevation Prompts



UAC: Elevation Prompts

It was mentioned before that when you are performing operations as a standard user, you typically run with Medium-IL. When you need to elevate a process because you want it to have more privileges, then you most likely will see a UAC prompt or consent pop-up. The example on this slide is from when I right-clicked on the Process Explorer icon and selected ”Run as Administrator.” The UAC prompt with the blue title bar is an indication that the application is trusted and signed by Microsoft. The yellow UAC title bar prompt indicates that a process’ publisher could not be verified or was not signed. The user should proceed with caution when a yellow UAC prompt presents itself. Since this was PE-bear.exe, I trust the application to do no harm to the system. One other title bar color you might see is red. A red UAC elevation prompt indicates that the application you are trying to run has a publisher that has been blocked. A block in Group Policy could also trigger a red UAC prompt. The user should be able to click through with confidence knowing that nothing malicious is about to happen to their system.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

UAC: Fusion



Applications and their manifests

Many applications have a manifest file tied to it that is used to describe the application itself. This XML file contains detailed information about the application's security context. There are several elements in the manifest.

- `supportedOS`
- `autoElevate`
- `heapType`

UAC: Fusion

When a process is being created, there are several checks that the ***CreateProcess*** API performs. One of them is calling into the system's Fusion database where information from an application's manifest file is stored. .NET applications make heavy use of manifests, but traditional Win32 applications can also use them. The manifest file is simply an XML file that can be embedded in the EXE as a resource. Including manifests as a resource is the Microsoft recommended method. The manifests have many case-sensitive elements in them and there are a few that can directly apply to us when it comes to elevating privileges or simply gathering information.

When developing applications, there is a choice to make as to what OS your program will support. The *supportedOS* element has an Id attribute that can indicate what OS is supported, and each OS version has a GUID tied to it. The *heapType* element is used to override whatever the default heap implementation is. For example, specifying *SegmentHeap* for the program will make it use the segmented heap, which tends to lower the amount of memory your application uses. The *autoElevate* is the most interesting to us as it indicates that the program can automatically elevate its privileges without prompting the user for consent via the UAC consent prompt. When the *autoElevate* element is set to TRUE, it means it is enabled and there will be no elevation prompt. Finding applications with *autoElevate* set to TRUE could give you a means to bypass UAC and escalate your privileges.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

UACMe Project



GitHub repo hosts many UAC bypass methods

Inside the repo are a few files that handle parsing the manifest files, and fusion.c is one such file. The main idea is to find files that have embedded manifests to parse and checking to see what the autoElevate element value is. There are roughly three key functions.

`FusionScanDirectory` `FusionScanFiles` `FusionCheckFile`

UACMe Project

One part of the process of finding UAC bypasses is to search an application's embedded manifest for the *autoElevate* field. The applications that have this value set to TRUE would be of interest because they could be part of a step in elevating privileges. Instead of recreating the wheel and making your own parser, UACMe by hfiref0x has done this already. The project is hosted on GitHub and the entire project demonstrates several bypass methods. Of particular interest, though, are the code portions that deal with parsing the embedded manifests. If you look under Source\Yuubari\fusion.c, you will find the logic for that parsing. The header file fusion.h holds the definitions for several custom structures, as well as function signatures. In the fusion.c source file, it appears there are three functions that directly relate to the parsing: *FusionScanDirectory*, *FusionScanFiles*, and *FusionCheckFile*. To better understand what each function is doing, we should take a deeper look at the code being used.

The UACME project can be found here: https://github.com/hfiref0x/UACME.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

UACMe Project: FusionScanDirectory



Responsible for scanning current directory

This function takes in a directory path and kicks off the file scanning process. Once that is done, it does a directory walk looking for any other items in that folder, like subfolders and application files. There are three functions that help get this done.

- `RtlSecureZeroMemory`
- `FindFirstFile`
- `FindNextFile`

UACMe Project: FusionScanDirectory

This function pretty much does what it sounds like—scans a directory. It will scan a directory for files, like applications, as well as any subfolders. With any subfolders that are found, it will recursively call itself to process any files in the subfolder. This logic is implemented by using functions like ***FindFirstFile*** and ***FindNextFile***. Both functions, when used in combination, enable an application to programmatically enumerate a directory. The ***RtlSecureZeroMemory*** function is used to securely zero out items like arrays and structures. The system guarantees that the block of memory will be zeroed out. For any files that are found, the function ***FusionScanFiles*** is called.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

UACMe Project: FusionScanFiles



Responsible for scanning EXE files

This function takes in a directory path and will look for EXE files in the directory. Any EXE that is found, the function will call FusionCheckFile. The file's information is stored in the process heap.

`HeapAlloc/HeapFree` `FindFirstFile` `FindNextFile`

UACMe Project: FusionScanFiles

The ***FusionScanFiles*** function will scan for any EXE files in the directory path that was passed in as a function argument. Since ***FindFirstFile*** and ***FindNextFile*** return information about a file in the form of the WIN32_FIND_DATA structure, the function allocates some space in the process heap to hold that information. When using the ***HeapAlloc*** function to allocate your block of memory, you should also use the ***HeapFree*** function to indicate the block is no longer being used by your program. The stored WIN32_FIND_DATA structure is passed as an argument to ***FusionCheckFile***.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

UACMe Project: FusionCheckFile



Responsible for parsing embedded manifests

The function takes in the path to the EXE, opens a file handle to it, maps it into memory, and searches the image to see if there is an embedded manifest.

`LdrResSearchResource` `NtCreateFile` `NtCreateSection`

UACMe Project: FusionCheckFile

The ***FusionCheckFile*** function accepts three arguments, but the ones of focus are the path of the EXE and the WIN32_FIND_DATA structure that describes the file. The function uses this information to open the file with read access (***NtCreateFile***) so that it can then map the entire image into memory (***NtCreateSection*** and ***NtMapViewOfSection***). Once mapped into memory, the function can then proceed to find the embedded manifest, if there is one. Applications can have embedded resources like the application manifest, and each one has a type associated with it. The resource type for our manifest is RT_MANIFEST. The ***LdrResSearchResource*** function will look over the mapped image and indicate if the desired resource type was located. If one is found, a flag will be set in the *IsFusion* member of their custom structure, *FusionCommonData*.

After locating the resource, an activation context is created. The context allows certain elements to be queried, like the *autoElevate* element. The *autoElevate* element will have a value of TRUE or FALSE with TRUE indicating that indeed, the application can elevate automatically without an elevate prompt. This information is queried with the assistance of the **RtlQueryActivationContextApplicationSettings** function. Now, with the foundation established, we can look at some source code.

If you're interested in other resource types, check out this URL: https://docs.microsoft.com/en-us/windows/win32/menurc/resource-types.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Module Summary



- Covered many ways to escalate your privileges
- Discussed pipes, services, tokens
- Discussed finding bypasses for UAC

Module Summary

In this module, we covered several methods for how you could programmatically escalate your privileges. Learning how Meterpreter's *getsytem* command works on the back end can lend a hand when creating your own version of it or making a modification to it so that it would work in your target environment. The more you understand basic services, privileges, tokens, etc., the better equipped you will be for creating your own novel local privilege escalation (LPE) technique. Perhaps you might even discover and create a remote method as well. In the end, persistence pays off.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Questions



What type of pipe can operate over a network?

A Half pipe

B Named pipe

C Anonymous pipe

Unit Review Questions
Q: What type of pipe can operate over a network?

A: Half pipe

B: Named pipe

C: Anonymous pipe

Tamarisk OffsecExam Exam solution

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Answers



What type of pipe can operate over a network?

A Half pipe

B Named pipe

C Anonymous pipe

Unit Review Answers
Q: What type of pipe can operate over a network?

A: Half pipe

B: Named pipe

C: Anonymous pipe

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Questions



What API gives you a handle to a process' token?

A OpenProcessToken()

B OpenToken()

C OpenProcess()

Unit Review Questions
Q: What API gives you a handle to a process' token?

A: OpenProcessToken()

B: OpenToken()

C: OpenProcess()

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Answers



What API gives you a handle to a process' token?

A OpenProcessToken()

B OpenToken()

C OpenProcess()

Unit Review Answers
Q: What API gives you a handle to a process' token?

A: OpenProcessToken()

B: OpenToken()

C: OpenProcess()

Tamarisk OffSecExam Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Questions



What privilege gives complete write access regardless of the ACL?

A SE_BACKUP_NAME

B SE_RESTORE_NAME

C SE_WRITE_NAME

Unit Review Questions
Q: What privilege gives complete write access regardless of the ACL?

A: SE_BACKUP_NAME

B: SE_RESTORE_NAME

C: SE_WRITE_NAME

Tamarisk OffSecExam Exam solution

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Unit Review Answers



What privilege gives complete write access regardless of the ACL?

A SE_BACKUP_NAME

B SE_RESTORE_NAME

C SE_WRITE_NAME

Unit Review Answers
Q: What privilege gives complete write access regardless of the ACL?

A: SE_BACKUP_NAME

B: SE_RESTORE_NAME

C: SE_WRITE_NAME

Tamarisk OffsecExam Exam solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- **Operational Actions**
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 3

PE Format
Lab 3.1: PEParser
Threads
Injections
Lab 3.2: ClassicDLLInjection
Lab 3.3: APCInjection
Lab 3.4: ThreadHijacker
Escalations
Lab 3.5: TokenThief
Bootcamp
Lab 3.6: So, You Think You Can Type
Lab 3.7: UACBypass-Research
Lab 3.8: ShadowCraft

In this module, we will discuss several techniques centered around injection. There are a large number of injection methods and as you can see, we have our hands full of exercises today. Let's kick this off with the classic DLL Injection.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Bootcamp

- So, You Think You Can Type
- UAC Bypass-Research
- CustomShell

Bootcamp
For this bootcamp, this is the time to put the concepts learned from this section into practice. There are several challenges and all of them are optional. The challenges can also be done in any order since they do not depend on each other. The first challenge is to develop a program that implements similar functionality as ***GetProcAddress***.

Another challenge is to figure out how to intercept keystrokes that a user types into the notepad process.

The UAC Bypass challenge gives you the skillset to find a novel UAC bypass. The lab will only take you so far and then you must continue on your own. There is an excellent blog that heavily inspired this lab found here: https://github.com/Yet-Zio/WusaBypassUAC.

The final challenge is to continue creating your first baby implant. It should combine all features that have been covered so far.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Lab 3.6: So, You Think You Can Type

- Must handle Windows messages
- Must intercept user key presses
- Must create a DLL to inject into target

Lab 3.6: So, You Think You Can Type
This challenge offers a great opportunity to explore building a keylogger. Since most keystrokes are commonly meant for GUI applications like Notepad, we will make that our focus. The idea for this challenge is to create a DLL that will install a Windows hook that will then be used to monitor keystrokes. Once you have the main logic completed, you can spice it up a bit by rejecting certain keystrokes. As an example, you can prevent the user from sending any vowels to the Notepad process. More robust keyloggers will just log everything and write it all out to a log file that will eventually be sent out to the C2 back end for analysis.

Please refer to the eWorkbook for the details of this bootcamp challenge.

© 2024 Jonathan Reiter

© SANS Institute 2024

https://linktr.ee/offsecexam

Lab 3.7: UACBypass-Research

- Find system binaries that have autoElevate set to true.
- Explore the process behavior using Process Monitor.
- Find a vulnerability and weaponize it to bypass UAC.

Lab 3.7: UACBypass-Research
This challenge is not for the faint of heart. There are two paths you could take with this one: you can follow along with the lab guide that only takes you so far, or you can dive off and attempt to find a brand-new UAC bypass! If you have time, do both. If you find a new method, please share it with the class and consider contributing it to the UACme project on GitHub.

Please refer to the eWorkbook for the details of this bootcamp challenge.

Tamarisk OffsecExam Exam Solution

© SANS Institute 2024

https://linktr.ee/offsecexam

Lab 3.8: ShadowCraft

- Create a basic shell.
- Implement features covered in this section.
- Implement thorough error checking.

Lab 3.8: ShadowCraft
Please refer to the eWorkbook for the details of this bootcamp challenge.

Tamarisk OffSecExam Exam solution

© 2024 Jonathan Reiter