FORENSICS 610
REVERSE-ENGINEERING
MALWARE: MALWARE
ANALYSIS TOOLS AND
TECHNIQUES

Malicious Code Analysis

The right security training for your staff, at the right time, in the right location.
FOR610.3 focuses on examining malicious executables at the assembly level. You will discover approaches for studying inner-workings of a specimen by looking at it through a disassembler and, at times, with the help of a debugger. The day begins with an overview of key code reversing concepts and presents a primer on essential x86 assembly concepts, such as instructions, function calls, variables, and jumps. You will also learn how to examine common assembly constructs, such as functions, loops, and conditional statements. The second half of the day discusses how malware implements common characteristics, such as keylogging, packet spoofing, and DLL injection, at the assembly level. You will learn how to recognize such characteristics in malware samples.

The materials in this section were created by Michael Murr, and incorporate feedback and recommendations from FOR610 course participants. Michael maintains a blog at http://www.forensicblog.org and can be found on Twitter at http://twitter.com/mikemurr.
FOR610.3 Goals

- Comfortable reading code
  - Don't need to be a coder
- General understanding of language and environment
  - x86 assembly on Win32 platform
  - Similar concepts for other languages/platforms
- Understand techniques used by malware

Our goal for the FOR610.3 course module is to get you comfortable reading code and understanding what you're looking at. While you don't need to be a software developer, it definitely doesn't hurt.

The focus of the malicious code in this course will be Intel x86 assembly for the Win32 platform. This is one of the most popular platforms in businesses today, and there is a wealth of malware that exists. While we do focus on a specific platform, many of the concepts and techniques exist for other languages and platforms. In fact, there is some very "popular" malware that has been written in Visual Basic and Delphi.

This course isn't a "course about assembly", instead it's about understanding how malicious code works from a low level. It's quite often that malware analysts find themselves sitting in a disassembler or debugger looking at assembly, examining the various system calls, etc. To help facilitate understanding, we will cover some of the techniques used by rootkits and other types of malicious code.
We’ll begin by reviewing the core reversing concepts, setting the stage for the rest of this section.
Core Reversing Concepts

We will now go over the core concepts central to reverse engineering code.
This slide shows the overall picture of malicious code from initial inception to execution.

First, the malicious code author has to come up with the idea for what the malicious code will do. This could include activities such as hiding files, stealing sensitive information, logging keystrokes and passwords, etc.

Then the malicious code author writes the malicious code in a human readable source code. Examples of programming languages used to create malicious code include C, C++, Delphi, Visual Basic, Java, Python, etc.

After the source code has been written, it is then translated into executable code by compiling and linking the executable modules together.

Finally, the executable code is run on one or more machines. It is common for malicious code to communicate with other pieces of malicious code across a network.

The analyst is required to reverse engineer what the malicious code does, given only the executable code, and network traffic. In addition, the analyst may also be tasked to figure out what the original intentions of the malicious code author were.
Let's see how software goes from the development phase all the way to how it runs in memory. This will give you an understanding of some of what happens "behind the scenes." Sometimes it's useful to "think like a compiler" in order to understand what is happening.

A compiler is a tool that translates from one language to another. When malicious code is developed, it is often written in a high level language such as Microsoft Visual Basic, or Borland Delphi. A compiler takes the high level language and translates it into a low level language such as machine code. The resulting machine code is called an object file.

Linking is the next step in the process. A linker takes a series of object files (usually the output from a compiler) and combines the object files together, with their dependencies (external libraries) into a final executable file. Generally speaking, an executable can be statically or dynamically linked. A statically linked executable contains the dependencies, while a dynamically linked executable does not. One analysis technique is to examine the external dependencies of dynamically linked files. Some packers attempt to thwart this analysis technique by (temporarily) obfuscating the dependency tables.

Loaders are used to load an executable file into memory. One of the common tasks a loader performs is to locate and resolve any dependencies an executable has. The dependency resolution process can happen when an executable file is first being loaded into memory, or while it is running. Once a loader has loaded an executable file into memory, it transfers control to the first instruction the program by performing a jump. The first instruction executed is called the "entry point". Malicious code such as viruses and packers will often modify the entry point, so execution starts with the malicious code. In this instance, the program's original entry point is often referred to as the "OEP".
Here is a high level overview of the various steps that code goes through from source to final destination.

The source code is translated into object code by a compiler. The object code is then combined with libraries, and an executable file is created.

To run the file, the operating system reads in various information from the executable file, allocates memory and loads required libraries into memory.

Finally, control is transferred to the code to execute. Libraries may be loaded during the program's execution, after loading. It is at this final stage that we typically examine the code with a debugger.

Realize that at each step some information is lost and some new information is gained. There are also several places where authors of malicious code can take steps to hinder analysis. For instance, there was a "Tiny PE" challenge that created a 97 byte executable! It would run, but it violated many of the specifications for a Portable Executable (PE) file.
Flow of Execution

- Order that instructions are executed
- Sequential, until a branch instruction
  - E.g. function call, conditional jump, etc.
- Code block: group of consecutive instructions, logically grouped together
  - Perform computations
    - E.g. Loops, crypto, etc.
  - Interact with environment (OS)
    - E.g. Read/write files and network, DLL injection, etc.

Once the program has started executing, the CPU executes instructions sequentially, one right after the other. However, there are also instructions that tell the CPU to jump to another location in memory (sometimes only if specific conditions are met) and continue executing code at the new location. This is called branching.

A series of sequential instructions is typically referred to as a "block" of code (or "code block").

When a program runs, it typically performs one of two primary activities. It can perform various types of computations, which include mathematics, Boolean operations, loops, jumps, cryptography, floating point instructions, etc. The program can also interact with its environment, such as reading and writing files, network traffic, or even performing activities such as DLL injection.
Here is a graphical example of program flow. This screen shot was taken from the Agobot.ct bot.

In this screen shot there are 4 blocks of code that are visible. The upper most block has a conditional branch to two separate blocks of code. These two blocks of code then lead to execution of the fourth block of code.

Regardless of which block of code execution is currently taking place in, the instructions are executed sequentially until a branching instruction occurs.
Functions

- Block of code that performs a specific task
  - read/write files, computation, etc.
- Three basic components
  - Input (values passed in)
  - Body (code to perform the task)
  - Return (values passed back)
- Calling functions typically requires jumps to another location in memory
  - After function is done, execution continues at instruction after original function call

Blocks of code that perform specific (and often times reusable) tasks are typically called functions. There are functions to read and write files, perform various computations (e.g. a custom crypto routine), etc.

Similar to mathematical functions, there are three basic components to a code function:
- Input that is passed into the function (from the caller)
- The body of the function, which is the code to perform the task
- The value(s) returned from the function

In order to call a function, a branch (jump to another location) is typically required. After the function is finished executing, the CPU continues processing instructions after the location where the function was originally called. In order to do this, when a function is called, the current location is typically saved in a special area of memory (the stack).
Example (Agobot.ct)

```
push esi
call edi ; GetProcAddress
push offset aCreateToolhelp
push esi
mov dword_436E14, eax
call edi ; GetProcAddress
push offset aProcess32First
push esi
mov dword_436E18, eax
call edi ; GetProcAddress
push offset aProcess32Next
push esi
mov dword_436E2C, eax
```

Here is a screen shot of various functions from the Agobot.ct bot. The GetProcAddress function is being called multiple times, each time with a different value.

In the first call to GetProcAddress, the address of the string "CreateToolhelp32Snapbot" is passed in. The GetProcAddress function locates the address of a function (in memory) given an ASCII string of the function name. The return value is in the EAX register, which is saved at address 0x436E10.

This process continues calling GetProcAddress for a number of different functions.
Inlined Functions

- Functions are typically in one location
  - Smaller code size, increased reusability, etc.
  - Entrance and exit (add'l overhead)
- Inlined functions have body in-line with "calling" code
  - No extra overhead for entrance and exit
- Common with string related functions

Code for a specific function is typically in one location in memory. This has the benefit of creating a smaller overall code base, increasing modularity, and a few other software engineering goals. The drawback, however, is that there is additional overhead when entering and leaving the function.

Some functions, especially smaller ones, can be "inlined." This means the body of the function is placed in the block of code that calls the function. The advantage is that there is no extra overhead for entering or exiting the function. You often see this with some of the string-related functions such as strlen and strcmp.

Of course this adds to the complexity of the original code block. It is difficult (if not impossible) to tell if a function has been inlined, or if the code block originally contained the instructions (not in the form of a function).

In addition, if the name of the function being called is in a library (e.g., strlen) then it is often easier to figure out what the function does (as it is more likely to be documented).
Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>push eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call strlen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add esp, 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Non-inline vs Inlined

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea edx, [eax+1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov cl, [eax]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add eax, 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test cl, cl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jnz short loc_401043</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub eax, edx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here is an example of both inlined, and non-inline function calls. These are identical pieces of code, the first compiled with optimizations turned off, and the second with "normal" optimizations.

The top code block shows a call to strlen, which computes the length of a NULL terminated string.

The bottom code block shows the inlined version of strlen. As you can see (by the arrow on the left hand side) there is a loop, which runs until a byte 0x0 (NULL) is found. Each time the loop iterates, the value of the EAX register is increased by one.
System Calls

- Purpose of OS is to manage hardware
  - Provides managed access for user space code
- Typically wrapped in libraries
- Can provide useful hints
  - Tend to be fairly well documented

The purpose of any operating system is to manage hardware. With a few exceptions, everything else the operating system does is to provide managed access for user space code. For instance, allocating memory requires management of the system RAM.

Typically, calls to the operating system are wrapped in libraries. This can greatly assist a malware analyst, since these libraries tend to be fairly well documented. Sometimes determining what a library function does is as simple as searching Google for the name of the function.

This type of analysis (analyzing system calls) has been used for many years. The tool strace (or Strace-for-NT) monitors all of the system calls made by a program. You can learn quite a bit of information by monitoring the system calls a specimen makes. The Sysinternals tool System Monitor monitors system calls for registry and file related activity. (System Monitor is a tool that combines prior stand-alone tools Regmon and Filemon.)
Process Address Space, Segments

- Each process has 4GB of space
  - Even if less physical memory
  - Accomplished by processor and OS cooperation
  - Protected mode (a.k.a. flat memory model)
- Any address is 32 bits
  - For 32 bit architecture
- Simpler for developer than other models
- Divided into contiguous chunks called segments
  - Use to hold code and data
  - Permissions describe what segments can do
    + Read/Write/Execute

When a program runs, it has memory allocated to it for use. This memory is called the process address space. Many of today’s programs run in "protected mode" (sometimes called the flat memory model).

In protected mode, each process has an address space that is 4 gigabytes large, and is separate from another process' address space. This is accomplished (even if there is less than 4 gigabytes of RAM) by cooperation between the CPU and the operating system.

What protected mode means is that any address, whether it be for an instruction, a function, a pointer, data, etc., is 32 bits for a 32 bit architecture. This is a simpler model for developers than some of the other models available.

To help separate code and data, from various libraries as well as the program itself, the process address space is often divided into contiguous blocks of memory called segments.

Segments can hold either code or data, and have access permissions. The permissions typically are read, write, and execute.

For example, many stack-based buffer overflow attacks take advantage of the fact that the stack on some systems is in a memory segment marked as executable. One protection mechanism for stack-based buffer overflow attacks is to put the stack in a memory segment that is not marked as executable.
Variables

- Used to hold information (code or data)
- Types
  - Local: only for current function, not saved
  - Global: for all functions
  - Static: only for current function, saved
  - Access restrictions implemented and enforced by compiler
- Local variables are typically on stack
- Global and static typically have own segment of memory

In the world of code, the term variable represents some piece of information (such as code or data) without necessarily needing to know the actual contents of the variable.

There are generally three types (or classes) of variables that are in common use. They are described by their "scope" (where they are valid and can be used). The first type is local, which means that this variable is only accessible by the function that allocates it. For this reason, local variables are allocated on the program stack. When a function exits, the code must "free" (allow for reuse) the local variables it allocated.

The second type of variable is called a global variable. This variable can be used from anywhere within the program.

The third type of variable is called static. It can only be used from within the function that allocates it, but unlike local variables, it does not get marked for re-use when the function exits. Global and static variables typically have their own segment of memory.

Any scope related "access restrictions" are enforced by the compiler, not the CPU.
Data Structures

- Layout of how we represent information in memory
  - Also how to access and manipulate
- Common types
  - Strings, linked lists, sockets, file handle, etc.
- When reversing, determined by usage
  - If a system call, we can get a good idea

The term data structure refers to the layout and representation of information, and how we access and manipulate that representation.

Data structures pervade computing. Common examples of data structures include strings (both ASCII and Unicode), linked lists, network sockets, file handles, arrays, etc.

When reversing code, the way to figure out what data structure you're looking at (and the components) is to look at how it is used. If the structure is passed as a parameter to a system or library call, it's often possible to get a good idea what the structure is, as the interfaces to system and library calls tend to be fairly well documented.
Pointers

- Variable that holds a memory address
  - Variable "points" to a memory location
- Access memory address is called dereferencing
- Provides a level of indirection
  - Can also be more efficient

One very important concept in reversing (and software development in general) is the concept of a pointer. A pointer is simply a variable that contains the address of some location in memory. In essence, a pointer "points" to some address in memory. The value of a pointer is an address in memory. This is a recurring theme, and when analyzing code (malicious or otherwise) you will more often than not work with many pointers.

When the address that the pointer points to is accessed, it is called "dereferencing", since the pointer references (points to) another location in memory.

Pointers are a fundamental building block of computing, and allow for a level of indirection. There are also efficiency reasons for using pointers. For instance, rather than copying an entire data structure around in memory, it is far more efficient to copy the value of a pointer (which is typically 4 bytes on a 32-bit system.)
In this example, there are two variables, variable A and variable B. The value in variable A is 255. The address of variable A is location 100.

Contrast this with variable B. The value of variable B is 100 (the address of variable A), while the address of variable B is 157.

In this instance, we say that variable B "points to" variable A. This means that the value in variable B is the address of variable A.
Self Referencing/Modifying Code

- Which is code and which is data?
  - `push 0×6F6C6C65`
  - "hello" (without the quotes)
  - 448378203247
- Depends on how code is used (context)
  - Can be either or both
- Means code can reference and modify itself
  - Treat itself as if it were data
- Worms and viruses leave copies of themselves
- Packers compress and obfuscate themselves

One property of most computers is that the only way to distinguish the difference between code and data is by how the information is used. For example, the following are all represented by the same binary sequence:

- The computer instruction “push 0x6F6C6C65”
- The ascii string “hello”
- The number 448378203247

The determination of whether information is code or data depends on how the information is used at any given point in time. This means information that is treated as code at one point during program execution can be treated as data at another point during execution.

This “code-data duality” allows programs to reference themselves (self-referencing code) and even modify themselves (self-modifying code) during run time. Malicious code such as worms and viruses leave behind copies of themselves. Packers will often modify themselves by compressing or obfuscating their own code.
Understanding Meaning

- Reverse engineering is more than just understanding flow of execution
- Understand meaning
  - From understanding how code executes
- Malicious code specimen
  - Loop that sends multiple spoofed packets
  - Function that launches the DOS attack
- Not something that can be automated
  - Only humans can apply meaning

It is important to keep in mind some of the goals during a reverse engineering session. Remember that reverse engineering is more than just understanding the flow of execution.

One of the common goals of reverse engineering is to understand the meaning of the code you are examining. We do this by understanding how the code executes.

For example, a malicious code specimen that has a loop to send out multiple spoofed packets may be part of a function that launches a denial of service attack. Alternatively, the loop may be used to send obfuscated communication between an attacker and a compromised system.

The ability to assign meaning to code is something that cannot be automated, since only humans can apply meaning.
Program Comprehension

- There is no "standard method" for meaning
  - Experience and knowledge of malicious code helps
- Identify functions
- Identify system calls
- Identify variables
  - By usage in sys calls and arithmetic
- Examine flow control (esp. loops)
  - Pay attention to what conditions cause the flow control to change
- Rename variables / functions as you go
- Write pseudocode

When attempting to comprehend the purpose of a program, there is no "standard method". Understanding the flow of execution, combined with experience and knowledge of how malicious code works, facilitates the comprehension process. This is why we teach both assembly constructs and malicious code tactics in this course.

There are several activities you can perform to help comprehend the meaning and purpose of a program. First, you can identify the functions, both internal and external. You can also identify the purpose of variables by their usage on the stack, in system calls, and how they are manipulated arithmetically.

Examining the flow of execution is a very useful activity. When doing so, pay close attention to loops, especially the conditions which cause the flow of control to change.

It is common for reverse engineers to rename variables as they proceed through a reverse engineering session. One obstacle to this activity is that compiler optimizations may use the same variable for different purposes. In this case, I like to either find a common name for the variable, based on usage, or give the variable a generic name (e.g. "local_var1") and then create comments at various points in the disassembly to remind me of the purpose of the variable.

Another technique I like to use during a reverse engineering session is to write pseudocode. Pseudocode is a simple, very high level description of what a program is doing. There is no specific format or language for pseudocode. Instead, the format used for pseudocode will vary from analyst to analyst. An example of pseudocode can be found at:
http://en.wikipedia.org/wiki/Pseudocode#Examples
Program Slicing

- Find all code that references a resource
  - E.g. variable, function, etc.
- Examine usage
  - Help with flow analysis
- Easy trick is to set a break point

Program slicing is a technique that refers to understanding a piece of a program by examining all of the places that the code, variable, function, etc. is referenced. This is a very popular technique.

The idea is that once all (or many) of the places a specific part of a program is referenced, meaning can often be inferred by examining the usage of the part of the program that is of interest. For instance, finding a function that is called from the main loop of a program, where the output of the function is compared to a number of commands is quite possibly part of a control channel.

One very easy trick to find where and how a program is referenced is to simply set a break point on the start of the code, or an on-access memory breakpoint if examining a variable. One drawback to this is that some pieces of code, or areas of memory, are accessed many times, slowing down a reverse engineering session.
FOR610.3 Roadmap

- Core Reversing Concepts
- Assembly Primer
- Anti-Disassembling
- Exception Handling
- Hands-On Exercises (Throughout)

1st half of FOR610.3

... then 2nd half of FOR610.3

Next on our course roadmap is the discussion of key assembly concepts.
In this module we will go over some of the concepts from Intel x86 assembly that you will likely encounter on a regular basis when analyzing malicious code.
Register Based Machines

- CPU has series of registers
  - On chip memory locations
- Instructions act on registers and memory locations
- Some registers are general purpose
  - Some have particular use
  - Some are both

The Intel x86 architecture is based on the concept of a register machine. A register-based CPU is essentially a CPU with on-chip memory locations called registers. There are other models of computing, although the register based machine is by far the most wide spread.

When a CPU executes an instruction, it must fetch the instruction from memory, decode the instruction, and then perform actions based on the instruction. The actions a CPU takes can manipulate information in registers or in memory.

As far as the registers are concerned, some are general purpose, meaning they can be used for a variety of purposes. Other registers have a particular use. Some of the general purpose registers (e.g., ECX) also have special purposes, depending on the instruction that is being executed.
General Purpose Registers

- General purpose: EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI
  - EAX: used by addition and multiplication
  - ECX: used as a counter
  - EBP: used to reference arguments and local variables
  - ESP: points to last item on stack
  - ESI/EDI: used by memory transfer instructions

There are 8 general purpose registers that you will encounter on a regular basis. They are EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. Each register also has a specific purpose, depending on the type of instruction currently being executed.

The EAX register is commonly used as a default for addition and multiplication instructions. The ECX register is commonly used as a counter for looping. The EBP register is often used to reference arguments passed into a function, as well as the local variables within a function. The ESP register is used to point to the last item on the stack, and is affected by stack related functions. The ESI and EDI registers are used with memory transfer instructions.
**Special Use Registers**

- **EIP**: points to instruction to execute next
- **EFLAGS**: Bits represent outcome of computations, control CPU operation
- **Segment Registers**: CS, DS, ES, FS, GS, SS
  - CS: Code segment
  - DS: Data segment
  - SS: Stack segment

There are also special use registers, which have a particular purpose. The EIP register contains the address of the next instruction to execute (it "points" to the next instruction).

The bits in the EFLAGS register are used for two purposes: to represent the outcome of computations, and to control the operation of the CPU.

The segment registers (CS, DS, ES, FS, GS, and SS) are used to describe different segments of memory. The CS register is the default segment register when fetching instructions. The DS register is the default segment register for accessing data with the ESI and EDI registers. The SS register is the default segment register for accessing data with the ESP register.

There are other registers (e.g. cr0), which have various purposes. The registers listed on the slide above, are what you're likely to encounter on a daily basis. For more information about the various other registers, not described here, see section 3.4 of volume 1 of the "Intel Architecture Software Developers Manual."
Here is a graphical example of how the EIP register works. In this example, the first value of the EIP register is 0x40174. At this address is the instruction "sub eax, edx". This instruction is two bytes long, hence the next value of EIP is 0x40176 (two bytes more than the previous instruction).

The next three instructions are one byte in length, and the values for EIP are 0x40176, 0x40177, and 0x40178 respectively. The jump instruction at address 0x40178 tells the CPU to start executing code at memory address 0x40288. This moves the value 0x40288 into the EIP register.

Once the value 0x40288 is in the EIP register, the CPU will start executing instructions at this address. In this case the instruction is "test eax, eax". This instruction is also two bytes long, so the next value of EIP is 0x4028A.
Instructions

- Two components: operation and arguments
  - Arguments typically called operands
- Can have 0, 1, or 2 operands
- Operands can be:
  - A register
  - A memory location
  - An immediate value (i.e. a number)
- Destination first
  - E.g. MOV EAX, 0x6453
  - Move into the EAX register the value 0x6453

When working with instructions, there are two components. The operation and the arguments. The arguments are also called operands.

An instruction (depending on the operation) can take 0, 1, or 2 operands. Where the operands can be a register, a location in memory, or an immediate value (such as the number 0x6453).

In the syntax used to represent instructions in this course, the destination operand is listed first. In the instruction "mov eax, 0x6453" (which moves the value 0x6453 into the EAX register) the EAX register is the destination.
Operand Addressing

- Allow us to specify where to read and write data
- Four ways
  - Immediate (e.g. 0x6453)
  - Register (e.g. EAX)
  - Memory
    - Specify (or calculate) address
  - Implied (e.g. string and stack)

Computer instructions need to be told where to read data from, and write data to. The way this is communicated is by specifying different types of operands. The different ways of specifying where to read and write data are collectively called operand addressing modes.

The first addressing mode is immediate, which corresponds to immediate arguments. In this case the value is included as an operand to the instruction. The second addressing mode is register, and corresponds to using a register argument. The third addressing mode is when a memory address is used as an operand. In this case the memory address can be specified (as an immediate value) or it can be calculated.

There is another addressing mode, where the operands are implied by the instruction. For instance, the SCASB instruction scans memory pointed at by the EDI register for a value that is equal to the value in the AL register. Another example of an implicit operand are the stack operations, which move data to and from the ESP (extended stack pointer) register.
Directly Addressing Memory

- Dealing with address of data (not the data itself)
  - Pointers in C/C++
- Brackets mean fetch data at specified address (dereference)
  - E.g. mov eax, [0x80481234]
  - This is direct addressing (dereference immediate value)
  - Some tools omit brackets for direct addresses (e.g. IDA Pro shows dword_80481234)

We're now going to talk about scenarios when a memory address is used as an operand. Realize that for the next few slides we're going to be dealing with the address of the data, not the actual data itself. If you're familiar with C or C++, we'll be working with pointers.

Many tools will put brackets around an operand if the instruction is fetching data at the address the operand points to. In the example on the slide above, the mov instruction is accessing data at address 0x80481234. When the address is specified as an operand, it is called direct addressing.

Different tools represent direct referencing differently. By default IDA Pro doesn't use brackets, instead it uses the notation <size> <address>. For instance, it might use dword_80481234 instead of [0x80481234]. In contrast, OllyDbg uses the square bracket notation. This is a superficial difference, both representations mean the same thing.
Example

- Instr w/immediate Operand
  - mov eax, \textbf{0x6453}
- Instr w/register operand
  - imul eax
- Instr w/direct mem operand
  - mov ebx, \textbf{dword\_403028}
  - mov ebx, \[\textbf{DS:403028}\]

Here are three examples of different operand addressing modes. The bold operand corresponds to the type of addressing mode described.

In the first example, the operand \textbf{0x6453} is an immediate operand. In the second example the operand \textbf{eax} is a register operand. In the third example the values \textbf{dword\_403028} (for IDA Pro) and \[\textbf{DS:403028}\] are direct memory operands.
Indirectly Addressing Memory

- Address is calculated (or in register)
  - Called "Effective Address"
- Efficiently work with data structures
- Format: Base + (Index * Scale) + Disp

<table>
<thead>
<tr>
<th>Base</th>
<th>Index</th>
<th>Scale</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX EBX</td>
<td>EAX EBX</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>ECX EDX</td>
<td>ECX EDX</td>
<td>2</td>
<td>8-bit value</td>
</tr>
<tr>
<td>ESP EBP</td>
<td>EBP ESI</td>
<td>4</td>
<td>16-bit value</td>
</tr>
<tr>
<td>ESI EDI</td>
<td>EDI</td>
<td>8</td>
<td>32-bit value</td>
</tr>
</tbody>
</table>

To complement direct memory addressing, there is indirect memory addressing. In this case, the address of the destination is calculated, or resides in a register. The calculated address is called the effective address (EA).

If the address sits in a register, this is still different than direct memory addressing, where the register is the destination. In indirect memory addressing the register holds the *address* of the destination.

One very large advantage of indirect memory addressing is the ability to efficiently work with data structures. Incrementing the value of a single register can be used to step through fields of a data structure, or the same field of an array of data structures.

The general format for indirect memory addresses is:

\[ \text{base} + (\text{index} \times \text{scale}) + \text{displacement} \]

Where the values can come from the columns displayed on the slide. Any value can be null (not used) with the exception that if a scale is used, and index register must also be used.

You will encounter this addressing mode fairly often when reversing.
Example

- **[EAX]**
  - Access dynamically allocated memory (base)
- **[EBP + 0x33A2]**
  - Access data on the stack (base + displacement)
- **[EAX + EBX * 4]**
  - Array with 4-byte structures
    (base + index * scale)
- **[EAX + EBX + 0x33A2]**
  - Access fields of a two dimensional array of
    structures (base + index + displacement)

Here are some examples. The first utilizes only the base register. One common scenario for this type of memory access is the register will hold the address of dynamically allocated memory.

The second example uses both a base register and a numeric displacement. This is commonly used with accessing local variables and parameters on the stack.

The third example uses base and index registers with a numeric scale. This can be used when accessing an array with 1 byte structures (such as an array of pointers).

The final example uses base and index registers with a numeric displacement. This can be done when accessing the fields of a two dimensional array of data structures.
Two Dimensional Array

- Two dimensional array of IRC servers Undernet, EFnet, DALnet (3 servers per row, 1 row per network)

\[
\begin{align*}
\text{Displacement (start of array)} & \quad \text{EBX} + (\text{ECX} \times 4) + \text{Displacement} \\
\text{(e.g. 0x404000)} & \quad 12 + (2 \times 4) + 0x404000 \\
& \quad \{0x40128056, 0x42BA3B32, 0x0D0531482\} \\
\text{Base (row, EBX)} & \quad \{0x0D8F63971, 0x42C65043, 0x42C3FC05\} \\
& \quad \{0xC2442D32, 0xD063C182, 0xCA32209\} \\
\text{Index} \times \text{Scale (column, ECX} \times 4)) & \quad \text{Index} \times \text{Scale (column, ECX} \times 4))
\end{align*}
\]

Here is a graphical example of how all of the addressing components can be used together in an efficient manner.

Pretend that an IRC bot is programmed to try to connect to three different IRC servers, on three different networks (Undernet, EFnet, and DALnet.) One way of storing this in the code is as a two dimensional array. On the slide above, you can see a table of IP addresses of IRC servers. There are three rows (one for each IRC network, Undernet, EFnet, and DALnet) and three columns (three different IRC servers from each network) for a total of nine entries. Each IP address is four bytes. The entries shown in the table above are in network byte order, and can be directly converted to dotted quad notation. E.g. 0x42C3FC05 = 0x42 0xC3 0xFC 0x05 = 66.195.252.5 = irc.blessed.net.

Let's pretend we wanted to access the third item on the second row (the entry that has been bolded and underlined.) Using all of the addressing components, the value for displacement would be the start of the array in memory. In our example, we chose the array to start at memory address 0x404000. The displacement is the starting address for efficiency purposes. If it wasn't the starting address, iterating over each element would require a new constant for each element. Not a very efficient use of code.

To describe the row, we use the Base. In this example, we chose the EBX register to be the base register. Since each row has three columns, and each entry is four bytes in size, each row is twelve (3 \times 4) bytes in length. Therefore, we would increment EBX by multiples of twelve. In our example, we want to access the second row, so the value of EBX is twelve.

To describe the column, we use the scale. For this example we chose to use the ECX register. The value multiplied with the scale in this case is four. This is because each entry is four bytes. Since we want the third entry, the value of the ECX register is two.

The total calculation is EBX + (ECX \times 4) + Displacement. Substituting in our values of 12, 2, and 0x404000 respectively we get: 12 + (2 \times 4) + 0x404000 = 12 + 8 + 0x404000 = 0x404014. This means that the third entry on the second row starts at memory address 0x404014.
Exercise 1:
Identify Addressing Modes

What type of addressing mode is used, and what is left in the EAX register?

1. mov eax, 0xEBFE

The value at address 0x1234 is: 42

2. mov eax, dword_0x1234

3. mov eax, [0x1234]

EBP = 100, ECX = 255, value at address 614: 6453

4. mov eax, [ebp+ecx*2+4]

For this exercise, you need to identify the type of addressing mode used, and the final value that gets moved into the EAX register.

1. mov eax, 0xEBFE
   Addressing mode: Immediate
   Value in EAX: 0xEBFE

2. Assume the value at memory address 0x1234 is 42.
   mov eax, dword_1234
   Addressing mode: indirect memory
   Value in EAX: 42

3. Assume the value at memory address 0x1234 is 42.
   mov eax, [0x1234]
   Addressing mode: indirect memory
   Value in EAX: 42

4. Assume the value at memory address 614 is 6453, EBP = 100, ECX = 255
   mov eax, [ebp+ecx*2+4]
   Addressing mode: indirect memory
   Value in EAX:
## Answers

### Identify Addressing Modes

1. `mov eax, 0xEBFE`
   - Immediate addressing
   - Value in EAX = 0xFBFE

2. `mov eax, dword_0x1234` (IDA Pro)

3. `mov eax, [0x1234]` (OllyDbg)
   - In both cases direct memory operand
   - Value in EAX = 42

4. `mov eax, [ebp+ecx*2+4]`
   - Indirect w/base, scaled index, and displacement
   - Value in EAX = 6453

---

Here are the answers to the exercise:

1. `mov eax, 0xEBFE`
   Addressing mode: direct, immediate operand
   Value in EAX: 0xEBFE

2. Assume the value at memory address 0x1234 is 42.
   `mov eax, dword_1234`
   Addressing mode: direct, memory operand
   Value in EAX: 42

3. Assume the value at memory address 0x1234 is 42.
   `mov eax, [0x1234]`
   Addressing mode: direct, memory operand
   Value in EAX: 42

4. Assume the value at memory address 614 is 6453, EBP = 100, ECX = 255
   `mov eax, [ebp+ecx*2+4]`
   Addressing mode: indirect, with base, scaled index, and displacement
   Value in EAX: 6453
In the world of assembly, there are two commonly used notations for describing instructions and their operands. The format that this course uses is called Intel syntax.

The other popular notation is called AT&T syntax. There are some subtle but important differences. The source register comes first, as opposed to the destination register with Intel syntax. Also, AT&T syntax precedes registers with a percent sign (e.g. %EBX). Immediate values start with a dollar sign (e.g. $0x6453).

Finally, the format for memory addressing is:
offset(base,index, scale)

So [ECX-EBX*4+8] becomes 8(%ecx,%ebx,4) (Note that in this case the 8 and 4 are not preceded by a dollar sign).
Types of Instructions

- Data Manipulation
  - Arithmetic, Boolean, other
  - ADD, SUB, SHR, AND, OR, etc.
- Data transfer
  - MOV, XCHG, etc.
- Branching and conditionals
  - JMP, CALL, CMP, etc.

When working with assembly code, there are a number of different instructions. Instructions typically fall into one of three categories.

The first category is data manipulation. Instructions in this category would include arithmetic (e.g. add, sub, fmul, etc.) Boolean (e.g. and, or) bit manipulation (e.g. shr, shl, etc.) and other miscellaneous commands.

The second category is data transfer. Instructions in this category would include mov, xchg, etc.

The third category consists of branching and conditional instructions. Examples of this include the jmp, call, cmp and test instructions.
Branching

- Change execution to new location in memory
  - Update the value in EIP
- Common uses
  - Jumps
    - Conditional (e.g. jne)
    - Unconditional (e.g. jmp)
  - Call a function (CALL), return (RET)
  - Looping

Branching is transferring control of execution to another location (address) in memory. Essentially the value in EIP is updated, although depending on how the branch is initiated (call/ret vs. jmp), other actions may be taken as well.

There are several instances where branching is used. First are unconditional and conditional jumps. Branching also occurs when a function is called (call) or returns (ret). Finally, looping is also an example of branching.
Jumps

- Jump to a new location in memory to execute code
  - Transfer of control
  - Think of stack based buffer overflows
- Unconditional: always jump
  - JMP, CALL, RET instructions
- Conditional: only jump if condition is met
  - Jcc, Loop instructions

A jump instruction tells the CPU to start executing code from a new location in memory. It is a transfer of control.

There are two types of jumps, unconditional and conditional jumps. Unconditional jumps always branch to a new location in memory. Examples would be the jmp, call, and ret instructions.

Conditional jumps only branch to a new location in memory if a certain condition is met. Examples of conditional jumps are the Jcc and loop instructions.
The idea behind conditional jumps is that a specific condition must first be evaluated, and the jump is followed only if the condition was met.

Examples of conditions could be Boolean instructions such as and, or, not, etc. In addition, arithmetic instructions can be considered conditional. In either case, the side effects of the operation are stored in the flags register. For example if the instruction "add eax, ecx" results in zero in the EAX register, then the zero flag (ZF) bit is set in the flags register.

There is one large disadvantage of using arithmetic or Boolean instructions to evaluate a condition. They modify the value in the destination operand. To evaluate a condition without modifying the value in an operand, the cmp and test instructions exist. The cmp instruction performs an implied subtraction (sub). Implied means while the value of the destination is not changed, the bits in the flags register are set as if a subtraction instruction had actually been executed. It is common to see a cmp instruction followed by one of the conditional jump instructions.

Similarly, the test instruction performs an implied Boolean and (and).
Jcc Instructions

- Jump if condition is met
- Form: Jcc
  - A = above, B = below, E = equal, N = not equal, G = greater than, L = less than, Z = zero
  - Above / Below for unsigned comparison
  - Greater than / Less than for signed comparison
- Examples:
  - JA = jump if above
  - JNGE = jump if not greater than or equal
  - JNZ = jump if not zero

A conditional jump takes place if a specific condition is met. The form of a conditional jump instruction is typically in the form of Jcc, where the c's describe the conditions. The following values are used for conditions:

A: jump if above
B: jump if below
E: jump if equal
G: jump if greater than
L: jump if less than
Z: jump if zero
N: jump if not condition (e.g. JNZ is jump if not zero)

Jump instructions with above/below condition codes evaluate the arguments as if they were unsigned. Jump instructions with greater/less than condition codes evaluate the arguments as if they were signed. The developer, not the processor, determines if data is signed or unsigned, by selecting the appropriate jump instructions (e.g. above for signed data, greater than for unsigned data.)

The condition associated with a jump is evaluated by examining bits in the flags register. Since the processor does not know if data is signed or unsigned when it performs computations, it sets bits for both signed and unsigned operations.

The processor examines the following bits to evaluate the corresponding condition:

(Undefined) JB: True if Carry flag - 1
(Signed) JL: True if Sign flag != Overflow flag
JE/JZ: True if Zero flag = 1
JA: True if both Carry and Zero flag - 0
JG: True if Zero flag = 0 and Sign flag = Overflow flag

Some examples of conditional jumps include: ja (jump if above), jnge (jump if not greater than or equal, same as less than), and jnz (jump if not zero).
Example (DoS.Win32.Synflood.e)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>804018C5</td>
<td>jmp short loc_4018D9</td>
<td>Unconditional jump</td>
</tr>
<tr>
<td>80403880</td>
<td>mov edx, [eax]</td>
<td></td>
</tr>
<tr>
<td>80403882</td>
<td>test edx, edx</td>
<td>Affects flags register</td>
</tr>
<tr>
<td>80403884</td>
<td>jz short loc_4038BE</td>
<td></td>
</tr>
<tr>
<td>80403886</td>
<td>mov ecx, [edx-8]</td>
<td></td>
</tr>
<tr>
<td>80403888</td>
<td>dec ecx</td>
<td></td>
</tr>
<tr>
<td>8040388A</td>
<td>jz short loc_4038BE</td>
<td></td>
</tr>
</tbody>
</table>

This screen shot shows three jumps from the synflooding tool ath0 (DoS.Win32.Synflood.c).

The first jump is an unconditional jump to address 0x4018D9. The second and third jumps are conditional jumps. The first instruction to affect the flags register is the test instruction (which performs an implied boolean "and"). Recall that a boolean "and" is true if and only if both values are 1. If either value is 0, then a boolean "and" is false. The jz instruction will then jump if the result of the test instruction was zero, by examining the status of the ZF bit in the flags register.

The instruction that affects the flags register for the third jump is the "dec ecx", which decrements the value in the ECX register by one. The jz instruction that follows jumps if ECX was decremented to zero.
Exercise 2: Examine Conditional Jumps

- Examine several conditional jump statements
  - Goal: Identify condition and when branch will be followed
- Examine irc handler of Sdbot.c
  - sdbot.exe

For this exercise you will examine several conditional jumps. The goal of this exercise is for you to become comfortable identifying conditions and when branches will be followed. This is a key component of analyzing code.

The specimen you will be analyzing is a bot called Sdbot, specifically Sdbot.c. The function you will be examining handles IRC communication.

The specimen is called sdbot.exe and is located on the DVD you received for this course.
Questions
Examine Conditional Jumps

• 4 conditional jumps between addresses 0x403365 and 0x4033E1 in sdbot.exe.
• For each conditional jump:
  1. Identify address of the conditional jump instruction
  2. Write out the acronym
     • E.g. jnie is jump if not less than or equal to
  3. Identify the address of the instruction that denotes the condition (changes the flags register)
  4. Describe the conditions for the jump to occur

Between the addresses 0x403365 and 0x4033E1 there are four conditional jumps in sdbot.exe. For each jump, answer the following four questions:

1. Identify the address of the conditional jump instruction
2. Write out the acronym (e.g. jule is jump if not less than or equal to)
3. Identify the address of the instruction that denotes the condition (changes the flags register)
4. Describe in your own words, the conditions for the jump to be followed.
## Hints

### Examine Conditional Jumps

- Conditional jumps are in the form `Jcc`
  - `cc` is the condition
  - Condition codes 5 slides ago
  - `jmp` is not a conditional jump
- CMP and TEST instructions
  - Perform implied `SUB` and `AND`
  - Modify flags w/o modifying other registers
  - Mathematics (e.g. `inc`, `dec`) also modify flags
- To describe the condition
  - Look at argument(s) to the instruction of the condition
  - Look at condition code for jump
  - Combine the two

Here are some hints to assist you in case you become stuck, or are having a difficult time.

Remember that conditional jumps are in the form `Jcc`, where the "cc" part is a condition code (what must be true for the jump to be taken.) The condition codes are listed five slides back. Don't forget that the `jmp` instruction is not a conditional jump (and therefore doesn't count towards the four conditional jumps).

When looking for the instructions that evaluate a condition, remember that you are looking for mathematical operations (arithmetic and boolean). These instructions modify the flags register, as well as the destination register. The `cmp` and `test` instructions perform an implied `sub` and `boolean` and. This means the instructions modify the flags registers, and do not modify the destination registers.

When describing the when the jump is taken, examine the condition, the arguments to the condition, as well as the condition code in the conditional jump. Then try to logically combine the two.
Answers (1)
Examine Conditional Jumps

- First conditional jump
  1. jnz at 0x403368
  2. Jump if not zero
  3. cmp eax, 0 at 0x403365
  4. Jump if eax is not zero (eax \not= 0)

- Second conditional jump
  1. jnb at 0x40339F
  2. Jump if not below
  3. cmp eax, 3 at 0x40339C
  4. Jump if eax is not below (above or equal) to 3
     (eax >= 3)

Here are the answers to the questions.

The first conditional jump is at 0x403368, a jnz instruction. The jnz stands for "jump if not zero." The condition instruction is at 0x403365, cmp eax, 0. Combining these two, this can be described as: The jump is followed if the value in the eax register is not zero.

The second conditional jump is at 0x40339F, a jnb instruction. The jnb stands for "jump if not below" (which is the same as above or equal.) The condition is at 0x40339C, cmp eax, 3. Combining these two, this conditional jump can be described as: The jump is followed if the value in the eax register is not below 3.
Answers (2)

Examine Conditional Jumps

- Third conditional jump
  1. jge at 0x4033C0
  2. Jump if greater than or equal to
  3. cmp eax, 0x40 at 0x4033BD
  4. Jump if eax is greater than or equal to 0x40 (eax >= 0x40)

- Fourth conditional jump
  1. jle at 0x4033E1
  2. Jump if less than or equal to
  3. cmp eax, 0 at 0x4033DE
  4. Jump if eax is less than or equal to 0 (eax <= 0)

The third conditional jump is at 0x4033C0, a jge instruction. The jge stands for “jump if greater than or equal to.” The condition is at 0x4033BD, cmp eax, 0x40. Combining these two, the conditional jump can be described as: The jump is taken if the value in the eax register is greater than or equal to 0x40.

The fourth conditional jump is at 0x4033E1, a jle instruction. The jle stands for “jump is less than or equal to.” The condition is at 0x4033DE, cmp eax, 0. Combining these two, the conditional jump can be described as: The jump is taken if the value in the eax register is less than or equal to 0.
High Level Logic

- Let's put together what we know
  - Start translating back to source language
  - Easier to understand
- Branching
  - If-else, Switch
- Looping
  - For, While
- Compound expressions
- Not always an exact 1-for-1
  - Goal is to understand program flow, not recover code
  - TIMTOWTTI

We're now going to start putting together what we know about conditionals and branching to build an understanding of higher-level logic. We're going to start translating (or at least understanding) the flow of execution from the perspective of the source language. This tends to be easier to understand than assembly code.

We'll cover branching-related logic constructs, such as if-else. We'll also cover looping-related logic constructs, such as for and while loops. Finally, we'll examine the logic of compound expressions.

Remember that we will not create the original source code. Our goal is to understand the flow of the program. We will provide sample translations of assembly and C code. However, they are only samples, and don't account for optimizations that a compiler may make. Remember, there is more than one way to translate it (TIMTOWTTI). In the field, you will likely encounter many of the same logic constructs, but implemented in a variety of different ways.
If-Else Statements

- Execute first code block IF condition is true
  - Else execute second code block
- General form (in C/C++):
  ```c
  if ( condition ) {
    first code block
  } else {
    second code block
  }
  continue here after if...
  ```

The first type of high level logic construct that we'll examine is the "if-else" statements.

If-else statements have a condition and two code blocks. The first code block is executed if the condition is true. Otherwise the second code block is executed.

Here is the general form of an if-else statement in C/C++

```c
if ( condition ) {
    first code block
} else {
    second code block
}
```

Keep in mind that only one of the two code blocks is executed, not both.
Here is an example of how an if-else statement from C/C++ could translate to assembly.

In the C/C++ code, the variable var1 is checked to see if it equals the value 0xBAADF00D. This value is used by the Microsoft memory allocation routines to indicate uninitialized memory. This comparison can be seen in the first three lines of assembly code, where the cmp instruction is comparing the value of the EAX register with the immediate value 0xBAADF00D.

If the two values are not equal (if var1 does not equal 0xBAADF00D) then the conditional jump is followed to the label for the second code block.

If the value of var1 does equal 0xBAADF00D, the conditional jump is not followed and the first code block is executed. Notice that at the end of the first code block there is an unconditional jump to the end of the if-else statement. This is necessary so that the second code block is not executed after the first code block is finished.
If-ElseIf-Else Statements

- Nest a series of if-else statements
  - Else contains another if-else
  - If all conditions are equality, a switch statement
- General form (C/C++):

```c
if ( condition-1 ) {
    code block #1
} else if ( condition-2 ) {
    code block #2
} else if ( condition-Y ) {
    code block Y
} else {
    code block Z
}
```

It is possible to nest multiple if-else statements together, creating if-elseif-else statements. The idea is that multiple conditions can be evaluated, and a single code block executed, depending on the first condition that is met. If none of the conditions are met, the code block in the else clause (which is optional) is executed. Essentially each else clause contains another "if" statement.

If all of the conditions are for equality (checking to see if one variable equals a specific value) then it may be called a "switch" statement. While a switch statement has a different syntax, it is functionally equivalent to a series of nested if-else statements.

The general form for if-elseif-else statements is:

```c
if( condition-1 ) {
    code block #1
} else if( condition-2 ) {
    code block #2
} else if( condition-Y ) {
    code block Y
}
```
In this example, we will show one translation of an if-else-if-else to assembly code.

In the C/C++ code on the slide above, the variable "a" is checked to see if it equals the value 1. If it does, the first code block is executed. Otherwise the variable "a" is then checked to see if it equals the value 2. If it does, the second code block is executed. Finally if the variable "a" does not equal 1 or 2, the third code block is executed.

The assembly translation has a few more components to it. The first comparison ("cmp eax, 1") checks to see if the value in the EAX register is 1. If it is not, it jumps to the Secondif label. If the value is 1, the first code block is executed and then a jump to the end of the if-else-if-else clause (so the other code blocks are not executed.)

The second if checks to see if EAX is equal to 2. If it is the second code block is executed, with a jmp command to the end of the if-else-if-else clause. If the value of EAX was not 2, control is transferred to the label Else, which executes the third code block.
Switch Statements

- Execute one code block (of many) based on integer value of a variable
  - Similar to nested if/else statements
- General form (C/C++)

```c
switch( variable ) {
    case 0:
    /* Code executed when variable == 0 */
    break;
    ...
    case N:
    /* Code executed when variable == N */
    break;
}
```

The switch statement allows one (of many) code blocks to be executed, based on the integer value of a variable. A switch statement is functionally equivalent to a series of nested if/else if statements.

When working with switch statements, each code block is started with the line “case XXX:”, where XXX is a constant value that the variable must match for the code block to be executed. The “case XXX” line is often called a “label”.

Code blocks will often be terminated by a “break” statement. The break statement tells the compiler that the code inside the switch statement has finished, and to continue executing code after the switch statement.

If the break statement is not present, then execution will flow from one case (code block) into the next. While this is usually not what the programmer wants (i.e. for execution from one code block to flow into the next), there are some instances (e.g. a Duff’s device) where this trait can be used for efficiency purposes.
Jump Tables

- Switch statements are equivalent to nested if/else statements
- When all case labels are sequential compilers often use jump tables
- Address for each code block is stored in an array (table)
- Jump to the address: array[variable*4]
  - Variable is used as an index into the array
  - 4 bytes for a 32-bit instruction

Switch statements are functionally equivalent to a series of nested if/else statements. This means that switch statements have the same assembly language representation as nested if/else statements.

When all of the case labels are sequential, there is an optimization technique called "jump tables" that compilers often use.

A jump table is a list of addresses of each code block (an array). Control is transferred to the desired code block by using the variable to look up the address of the code block in the jump table.

It is common to see the variable used to index the jump table multiplied by 4, since addresses on 32-bit systems are 4 bytes (32-bits).
Example Translation: Switch

switch(var1) {
  case 0:
    first code block
    break;
  case 1:
    second code block
    break;
}

Here is an example of how a switch statement could translate to assembly, using jump tables.

In the C/C++ code, the value of the variable "var1" determines which code block is executed. If "var1" is 0 then the first code block is executed. If "var1" is 1, then the second code block is executed. For any other values, neither of the code blocks are executed. Execution continues after the end of the switch statement.

In the assembly, the value of "var1" is moved into the EAX register. The EAX register is then compared to the value 1. If the value of EAX is above 1, control transfers to the "end" label. This check prevents any values outside the valid range (in this example 0–1) from executing either of the code blocks.

Before we continue, it is important to notice that the first code block is located at address 0x1000, and the second code block is located at address 0x2000.

The "jmp code_block_address_table[eax*4]" instruction performs several actions. First the value of EAX is multiplied by 4, and then used to access entries in the array "code_block_address_table". If the value of EAX is 0, the result of the multiplication is 0, and the jump is made to address 0x1000 (start of the first code block). If the value of EAX is 1, the result of the multiplication is 4, and the jump is made to address 0x2000 (start of the second code block).

It is also important to notice the "jmp end" instruction at the end of the first code block. When the code for the first code block has finished, the "jmp" will skip over (and not execute) the code in the second code block.
Looping

- Repeat code block until some condition is met
  - Each repetition is called an iteration
- Methods
  - Use a conditional jump
  - LOOPcc instructions
    - Examines ECX register
    - Z = zero, E = equal, N = not
    - loopnz: loop if ECX is not 0
    - loop: loop if ECX is not 0 (used for short jumps)
    - Automatically decrements ECX

The concept of looping is repeatedly executing a block of code until some condition is met. Each time the block of code is executed, it is called an "iteration."

There are two ways to create a loop, either using one of the conditional jumps, or by using one of the LOOPcc instructions.

Similar to Jcc, the c's in LOOPcc represent the condition code that must be met in order for the loop instruction to branch to the address specified. The conditions are:

- Z: loop if zero
- E: loop if equal
- N: inverts the logic of the looping condition

The loop instruction (without a condition code) loops if ECX is not zero, and is used for branches that are less than 128 bytes away.

All of the loop commands automatically decrement the value of the ECX register. The loopnz command will branch (loop) if, after the decrement, the value of ECX is not zero (if the ZF bit in the flags register is not set).
Loop Components

- All loops have 5 components
  - Control variable
    - The variable(s) that are used to determine if a loop exits
  - Loop initialization
    - The starting values for loop control variables
  - Loop body
    - The code block that gets executed
  - Loop update
    - How the control variables are modified during each loop iteration
  - Stopping conditions
    - The conditions to determine if a loop exits

From a high level perspective, looping constructs have five components:

1. Control variable: The variable(s) that are examined in the condition, and determine if the loop repeats or exits.
2. Loop initialization: The initial values for the control variable(s).
3. Loop body: The block of code that is executed for each iteration of the loop.
4. Loop update: The instructions that update the control variables.
5. Stopping conditions: The conditional instructions (typically Jcc or LOOPcc) that determine if the loop repeats or exits.
This is an example of the loop instruction from the Bagle.ac worm.

Address 0x40342C through 0x403439 comprise the loop. You can see the value of 9 is pushed and then popped into the ECX register. This means the body of the loop will be executed 9 times.

Essentially there are some data transformations applied to the EDX register (a right shift and an exclusive-or).

When the loop instruction is encountered, it decrements the value of the ECX register by one, and loops if ECX is not decremented to zero.
Here is an example that we saw earlier of the inlined string function (strlen).

In this case, a conditional jump is used at address 0x40104A. The previous instruction (test cl, cl) checks to see if the value in the CL register (low byte of the ECX register) is zero. If it is not zero, the loop is restarted.
Common Types of Loops

- For loop
  ```
  for(initialization; stopping condition; update) {
    code block
  }
  ```

- While loop
  ```
  initialization
  while( condition-is-true ) {
    code block
    update control variable
  }
  ```

In many high level languages, there are two common types of loops: the "for" loop, and the "while" loop.

Each loop has the five components mentioned before, and they are functionally equivalent, although they may differ in syntax.

In the "for" loop, the initialization, stopping, and updating conditions are specified at the beginning. In the "while" loop, the initialization is separate from the stopping condition, and the updating of the control variables is in the body of the loop.

The primary difference between while and for loops, is the stopping condition. The for loop states the condition when the loop should stop, whereas the while loop states the condition when the loop should continue.
Example Translation: For Loop

Here is one example translation for a for loop from C/C++ to assembly.

In this example, the start expression sets the value of the variable a to 0. The update expression is to increment the value of a by 1. The end condition is met when a is greater than or equal to 5. Essentially this loop will iterate over the code block 5 times.

In assembly we can first see the value of the EAX register being set to zero. Then the end condition is checked, and a conditional jump is taken if EAX is not less than (is greater than or equal to) the value 5. If the jump is not taken, the code block is executed.

After the code block is executed, the value of the EAX register is incremented by 1 and an unconditional jump is made to the CheckIfStop label, which checks the end condition.
Example Translation:
While Loop

Here is an example translation for a while loop into assembly.

Prior to starting the while loop, the variable a is initialized to 0. The loop then checks to see if the variable a is less than 5. If the variable a is less than 5, the code block is executed. At the end of the code block the variable a is incremented by 1.

You'll notice that the assembly translation is identical to the for loop. This is because they are both functionally equivalent. You may notice that the end condition is checked with a jge (jump if greater than or equal to) as opposed to a jnl (jump if not less than). They are actually identical, and are the same opcode (instruction).
Uses for Loops

- Encrypt/decrypt network traffic
  - Loop over each character in the string to send
- Attempt to connect to a list of IRC servers
  - Loop over list of IRC servers
- Perform a port scan
  - Try to connect to ports 1...65535
- Perform a DDOS attack
  - Keep sending malicious packet
- Log keystrokes
  - Check state for each key code 0...92

Looping is fundamental to computing, and found in numerous places throughout software, especially malicious code.

For example, looping may be used in the encryption and decryption routines, that encrypt data before sending it over the network. Malicious code might use a loop to iterate over each character in a string, modifying (encrypting) it somehow.

One common control mechanism for bots is to connect to an IRC server. Some bots will try to connect to multiple IRC servers, and might use a loop to iterate over a list of IRC servers to connect to.

Alternatively, a port scanner might repeatedly call a "attempt to connect" function for each port between 1 and 65535. Rather than having 65536 call instructions, a loop could be used, saving quite a bit of space.

DDOS attacks are classic examples of looping. It is not uncommon to see a loop used to repeatedly send malicious packets.

Finally, key stroke loggers work by check the state for each key code, to see if it has been depressed.
Exercise 3: Identify Loop Components

- Examine loops
  - Goal: Identify different loop components
- We'll examine clean up routine in primary thread of sdbot.c
  - sdbot.exe

For this exercise, you will examine loops. The goal is for you to be able to identify the different components of a loop.

You will examine the same specimen as before, except this time, the routine will be the "clean up" routine in the primary thread.
Question
Identify Loop Components

1. Loop between 0x4015F5 and 0x401612
   1. Identify addresses and instructions of stopping condition
   2. Identify address and instruction that updates control variable
   3. Identify memory address used as control variable (base-displacement). This may look like "[EBP + var]" where IDA Pro uses "var" to represent a negative number.
   4. Identify address and instruction of loop initialization
   5. Identify address range for loop body

There is a loop between the addresses 0x4015F5 and 0x401612. Answer the following five questions:

1. Identify the addresses and instructions of the stopping condition.
2. Identify the addresses and instructions that update the control variables.
3. Identify the memory addresses used as a control variable (in base-displacement form.) In IDA Pro, this may look like "[EBP + var]", where IDA Pro uses "var" to represent a negative number.
4. Identify the address and instruction of the loop initialization.
5. Identify the address range for the body of the loop.
Hints

Identify Loop Components

- Loops execute code many times
  - Look for backwards jumps
- Stopping condition typically conditional jump
  - May include a comparison (cmp or test)
- Control variable update instruction sets bits in flags register
  - cmp, test, and, or, add, sub, dec, inc, etc.
  - Tells you the memory location of the control variable
  - Typically close to (and before) stopping condition
- When you know end of loop and control variable, look for initialization
  - E.g. mov [ebx+4], 6453

Here are some hints, in case you get stuck.

Remember that loops execute code many times, and this means there will be a jump to a previous address. The jump will be a conditional jump, which means there may be a comparison instruction (such as cmp or test) near the conditional jump.

In addition, the instructions that update the control variables typically set bits in the flags register. This means that besides just cmp and test instructions, you may want to look for arithmetic instructions (e.g., add, sub, dec, inc, and, or, etc.) The stopping condition and update are typically close to each other, and once you identify the stopping condition and update instructions, you will know the memory location of the control variable.

Once you know the stopping conditions and the control variable, you can look for the loop initialization. This will typically be a mov or push/pop instructions.
Relevant Code Section

```
.text:004015F5  loc_4015F5:
.text:004015F5
.text:004015FC
.text:004015FC loc_4015FC:
.text:004015FC
.text:004015FF
.text:00401606
.text:00401608
.text:0040160E
.text:00401612

mov [ebp+var_4], 0
mov edi, [ebp+var_4]
push ds:[edi+4] ; s
call closesocket
inc [ebp+var_4]
cmp [ebp+var_4], 40h
jl short loc_4015FC
```

You can locate the code by loading sdbot.exe in OllyDbg or IDA Pro. Here's what you will see if you look at this code block in IDA Pro.
Answers (1)

Identify Loop Components

1. Identify addresses and instructions of stopping condition
   - cmp [ebp-4], 0x40 at 0x40160E
   - jl at 0x401612
2. Identify address and instruction that updates control variable
   - inc [ebp-4] at 0x40160B
3. Identify memory address used as control variable
   - ebp-4
4. Identify address and instruction of loop initialization
   - mov [ebp-4], 0x0 at 0x4015F5
5. Identify address range for loop body
   - Loop body at 0x4015FC – 0x401606
   - Note: [ebp-4] is the same as [ebp+var_4] in IDA Pro

Here are the answers to the exercise.

The stopping condition has two parts to it. The comparison is at 0x40160E, cmp [ebp-4]. The conditional jump is at 0x401612, jl (jump if less than).

The instruction to update the control variable is at 0x40160B, inc [ebp-4]. The memory address used as a control variable is ebp-4.

The loop initialization is at 0x4015F5, and is a mov [ebp-4], 0x0. The address range for the body of the loop is between 0x4015FC and 0x401606.

In IDA Pro, [ebp+var_4] is a symbolic name for [ebp-4]. To view this in IDA Pro, click on the text name and hit the letter 'q'.
Answers (2)

Identify Loop Components

Here are the answers overlaid on a screen shot (from IDA Pro) of the relevant code section.
Compound Expressions

- Simple expressions only evaluate a single condition
  - if \( x < 4 \) \{ block of code... \}
- Compound expressions evaluate multiple conditions
  - if \( x < 4 \) AND \( x > 1 \) \{ block of code... \}
- AND (short circuit)
  - Negate logic of each condition
  - \( x < 4 \) becomes \( x \geq 4 \), \( x > 1 \) becomes \( x \leq 1 \)
  - Jump to end of block if true
- OR (short circuit)
  - Test each condition
  - Jump to code block if true
  - Negate logic of last condition, jump to end of block if true

Up until now our examples have had simple expressions. A single condition (such as if a variable is less than 4) was evaluated.

Compound (sometimes called complex) expressions evaluate multiple conditions. For instance, we might wish to check if a variable is within a range of values. A compound condition could be if the variable is greater than 1, but less than 4.

There are several ways to translate compound expressions into assembly, especially given different optimizations that a compiler can implement. Many compilers use a technique called "short circuit evaluation" which skips a code block for the first Boolean AND that fails. Short circuit evaluation also executes a code block as soon as the first Boolean OR succeeds.

One way to translate an AND (which requires both expressions to be true) is to negate the logic of each condition, and if the negated condition evaluates to be true, jump to the end of the block. One way to translate a Boolean OR (which requires either one or both of the expressions to be true) is to test each condition and jump to the code block if any of them evaluate to be true. Then negate the logic of the last condition and jump to the end if the negated condition is true.

When you negate a logical condition, you change the condition so that the reverse outcome is true. For instance, negating the expression \( x < 4 \) yields \( x \geq 4 \), as the latter is the reverse of the former. Similarly, the negation of \( x > 1 \) yields \( x \leq 1 \).
Example Translation:
Compound Expressions (AND)

\[
\begin{align*}
\text{if}( (a < 4) \land (a > 1)) & \\
& \begin{cases}
\text{cmp eax, 4} \\
\text{jne End}
\end{cases} \\
& \begin{cases}
\text{cmp eax, 1} \\
\text{jeq End}
\end{cases} \\
& \begin{cases}
\text{code block} \\
\text{Block1:}
\end{cases} \\
& \begin{cases}
\text{code block} \\
\text{End:}
\end{cases} \\
& \begin{cases}
\text{Continue here after if...} \\
\text{Continue here after if...}
\end{cases} \\
\end{align*}
\]

In this example, the compound expression checks to see if the variable \(a\) is less than 4 and greater than 1. Effectively if the variable is either 2 or 3.

Notice on the assembly translation the logic has been inverted. The first condition that is checked is \(jnl\) (jump if not less than). The conditional jump branches to the end of the block. The second condition is also inverted, \(jng\) (jump if not greater than). If the conditional jump is followed it too branches to the end of the block.

If neither conditional jump is taken, the code block is executed.
Example Translation:
Compound Expressions (OR)

In this example, the compound expression checks to see if the variable a is less than 4 or greater than 10. Effectively if the variable is not 4, 5, 6, 7, 8, 9, or 10.

Notice on the assembly translation the logic for the first comparison has not been inverted. The first condition that is checked is jl (jump if less than). The conditional jump branches to the code block, because we want to execute the code block if the value is less than 4. The second condition however, is inverted, jng (jump if not greater than). If the conditional jump is followed it branches to the end of the block.

Similar to the AND compound expression, if neither conditional jump is taken, the code block is executed.
Exercise 4: Reverse Compound Expression

- Analyze compound expression
  - Goal: Understand / interpret a specific compound expression
- Examine system information module in Sdbot
  - sdbot.exe

For this exercise you will analyze a compound expression. The goal is for you to become comfortable interpreting and understanding compound expressions.

You will examine the same specimen as before, except this time you will examine code from the system information gather function.
Questions
Reverse Compound Expression

- "If-Else" statement between 0x406465 and 0x40648B
  1. Identify the number of variables in the condition
  2. Identify the memory addresses evaluated in the condition (base – displacement)
  3. Identify the start of the second code block (for the "else")
  4. Identify the addresses of the code block executed if the condition is met
  5. Identify the ending address of the "if-else" statement
  6. Identify the addresses of the code block executed if the condition is not met
  7. Describe the conditions when the first and second code blocks will be executed

There is an "if-else" statement between 0x406465 and 0x40648B. Answer the following seven questions:

1. Identify the number of variables in the condition.
2. Identify the memory addresses that are evaluated in the condition (in base-displacement format.)
3. Identify the start of the second code block (the code block executed if the condition is not met.)
4. Identify the addresses of the code block that is executed if the condition is met.
5. Identify the ending address of the "if-else" statement.
6. Identify the both the starting and ending address of the second code block.
7. Describe, in your own words, the conditions when the first and second code blocks will be executed.
Hints
Reverse Compound Expressions

- Remember the format for an "if-else" statement
  ```
  if( condition ) {
    1st code block
  } else {
    2nd code block
  }
  ```
- Look for jumps to same address
  - Suggests start or end of code block
- If condition is not met second code block is executed
  - Jump to start of second code block
- Second code block starts where first one finishes
  - So first one has to jump over second
- AND conditions invert logic, jump to second code block if true
- OR conditions jump to first code block
  - Invert last condition, jump to second if true

Here are some hints to help you, in case you get stuck.

Remember the format for an "if-else" statement. If the condition is met (true), then the first code block is executed, otherwise the second code block is executed.

You may want to look for jumps to the same address, as these suggest either the start or end of a code block. If the first code block is executed, there must be a jump over the second code block, so both are not executed. This implies that the second code block starts where the first code block finishes.

Don't forget a typical case of AND and OR conditions. AND conditions invert the logic of the condition, and jump to the second code block, if the (inverted) conditions are met. OR conditions jump to the first code block, if the conditions are met. The exception is that in an OR condition, the last condition is inverted, and jumps to the second code block, if the (inverted) condition is met.
You can locate the code by loading sdbot.exe in OllyDbg or IDA Pro. Here's what you will see if you look at this code block in IDA Pro.
Here are the answers to the exercise.

There are two variables evaluated in the condition, ebp-0x90 and ebp-0x8C. In IDA Pro, \[\text{ebp}+\text{VersionInformation.dwMajorVersion}\] and \[\text{ebp}+\text{VersionInformation.dwMinorVersion}\] are symbolic names for [ebp-0x90] and [ebp-0x8C] respectively. To view this in IDA Pro, click on the text name and hit the letter 'q'.

The start of the second code block is at 0x406485. The code block that is executed if the condition is met (true) is between 0x406477 and 0x406483.

The end of the "if-else" statement is at 0x406491. The second code block is between 0x406485 and 0x40648B.

The if condition can be described as: The first code block is executed if [ebp-0x90] equals 5 AND [ebp-0x8C] equals 1. Otherwise the second code block is executed.
Here are the answers overlaid on a screen shot (from IDA Pro) of the relevant code section.
Functions

- Commonly used code in one place
  - Optimization for size
  - Slight overhead to transfer control
- 3 Pieces
  - Input (getting data into function)
    - Parameters, arguments, operands, etc.
  - Body (computation / etc.)
  - Output (returning data from function)
    - Return value, result, etc.

A function is a block of code that is commonly used throughout a program. Typically, functions are meant to be somewhat independent of their environment, although this is not a strict requirement. The concept of a function in computer programming has several benefits, such as optimization for size, increased reusability, etc. There is however some slight overhead when transferring control in and out of a function.

When looking at functions, there are three components.
- Input: how data gets into a function
- Body: what the function actually does
- Output: how data gets out of a function
Calling Functions

\[
\text{return} = \text{function(} \text{arg1, arg2, \ldots } \text{)}
\]

- Actions when calling a function
  - Pass in parameters (stack / register)
  - Save return pointer
  - Transfer control to function
- Actions when returning from a function
  - Set up return value (typically EAX)
  - Clean up stack, restore registers
  - Transfer control to saved return pointer

The basic format for a function in C/C++ is:

\[
\text{return} = \text{function(} \text{argument1, argument2, \ldots } \text{)}
\]

Since there is a transfer of control when calling a function, certain housekeeping activities must be performed prior to actually entering the body of the function. First parameters must be set up to be passed into the function. This typically happens by placing the variables in registers or pushing them onto the stack. The address of the next instruction after the function call must also be saved. This is typically also put on the stack. Once this is done, the function can start executing.

When returning from a function, there are also housekeeping activities that must be done. The return value must be set up. Typically the EAX register holds the return value from a function. Also any local variables that were allocated need to be removed from the stack, and any registers that were used in the body of the function must also be restored. After this, control can be transferred to the return pointer (the address of the instruction after the original function call). The return pointer was set up by the block of code that called the function.
Calling Conventions (1)

- How to get data in and out of functions
  - Can vary per compiler
- cdecl
  - Arguments on stack, right to left
  - Return value in EAX
  - Caller cleans up stack
  - Most common
- stdcall
  - Similar to cdecl, callee cleans up stack
  - Used in WIN32 API

The term calling convention refers to how data gets in and out of functions, as well as who (the code block that calls a function, or the function itself) is responsible for performing which housekeeping activities. Unfortunately, not all compilers implement the calling conventions the same way. Luckily there are a few calling conventions that are fairly standard.

The first, and most common, calling convention is called cdecl. With this calling convention, any arguments passed into the function are pushed onto the stack from right to left (reverse order). The return value from the function is in the EAX register. It is up to the code block that calls the function to clean up the stack (remove parameters passed into the function). By requiring the calling code block to clean up the stack, a variable number of arguments can be passed into the function. One example of a function that takes a variable number of arguments is the printf() function.

Another common calling convention is called stdcall. This is used when calling functions in the WIN32 API. It is similar to cdecl, with the exception that the function itself is responsible for removing the arguments passed in from the stack. This also results in a smaller code size, since stack clean up instructions are not required in the code block calling the function.
Calling Conventions (2)

- fastcall
  - Parameters in registers
  - Extra parameters on stack
  - Caller cleans up
- thiscall
  - Used in C++ code (member functions)
  - Similar to cdecl, caller cleans up stack
  - ECX typically holds "this" pointer (Microsoft)
  - "this" pointer pushed onto stack last (GNU)

Another calling convention is fastcall. In this convention, parameters are first passed in by register, and then if extra parameters are required they are pushed onto the stack. Similar to cdecl, the block of code that calls the function is responsible for cleaning up the stack. The return value is also in the EAX register. Both Microsoft and GNU compilers use the ECX and EDX registers to pass parameters for fastcall.

C++ programs have another calling convention, called thiscall. In C++ an object can reference itself by accessing the "this" pointer. To facilitate this, the pointer to the object is passed in the ECX register for Microsoft compilers, and is pushed onto the stack last by GNU compilers. In other aspects (clean up and return value) thiscall is the same as cdecl.
Example Calling Conventions: cdecl and stdcall (Sdbot)

cdecl function (note the stack cleanup with add esp, 8)

push offset byte_40F898 ; Source
push offset byte_40E044 ; Dest
call strcpy
add esp, 8
push offset LibFileName ; "kernel32.dll"
call LoadLibraryA
mov [ebp+hLibModule], eax

stdcall function (no stack cleanup, WIN32 API function)

This is an example of two calling conventions from a variant of Sdbot.

The first function call (strcpy) is a cdecl function call. You will notice two arguments are pushed onto the stack. Also, the stack is cleaned up by the block of code that calls the function with the "add esp, 8" instruction. This cleans up the stack by incrementing the value in ESP by 8, which is equivalent to two POP instructions.

The second function call (LoadLibraryA) is called with the stdcall convention. While arguments are pushed onto the stack, there is no clean up done by the code block that calls the function. Also, LoadLibraryA is a WIN32 API function.
Example Calling Conventions: fastcall and thiscall (Sdbot)

```
mov [ebp+var_20], eax
mov edx, [ebp+var_20]
mov ecx, [ebp+var_14]
call sub_411299
```

fastcall function (note the use of edx and ecx registers)

```
mov ecx, [ebp+var_8]
call sub_41100A
```

thiscall function (ecx holds address of 'self')

In this example, the first function call (sub_411299) is made with the fastcall convention. In this case the EDX and ECX registers are used and there is no stack clean up.

The second function call (sub_41100A) is made with the thiscall convention. In this case ECX holds the address of the object.
Virtual Function Table

- Object oriented languages support developer-centric features
  - Inheritance, polymorphism, ...
- Common to resolve a function call at runtime
  - Allows for more "powerful" language constructs
  - Common to use a vtable (a.k.a. vtable)
- vtable is a list of possible functions
  - Have to select appropriate one at run time
  - "call 0x8048XXXX" translates to "call [eax]"
  - Makes reversing more difficult

Throughout the course of your examinations, you will likely encounter malicious code that has been written in object oriented (OO) languages, such as C++, Delphi, etc.

The features of object oriented languages, such as inheritance, polymorphism, encapsulation, etc. are all powerful language constructs that can facilitate software development.

One very common OO capability is to determine which function to call at run time. In C++ this is used to implement virtual methods. The most common approach used to implement virtual methods in C++ is to use a structure called a virtual function table (abbreviated vtable or vtable).

A virtual function table is essentially a list of addresses of functions that can be called. At run time, there is code that selects the appropriate address from the list, and then calls the function.

What this means to the reverse engineer is that you will see calls such as "call [eax]" instead of the more familiar "call 0x8048XXXX". This also makes static code analysis more difficult, since determining what function actually gets called is determined at run time. As a result, reversing tools will usually not be able to recreate the proper flow of execution.
This is an example of a virtual method call in the DataSpy Network X bot (version 0.4 beta). In this example, the virtual function table is at address 0x4101F8, and the function that eventually gets called is at address 0x4034D0.

The first thing that happens is the EDX register is set so that it points at the beginning of the virtual function table. In this example, EDX will contain the value 0x4101F8. Next the ECX register is set to point at the class to which the function belongs. This is an example of the "thiscall" calling convention. Finally, the call instruction calls the 6th function in the virtual function table. To determine the function that actually gets called, we have to examine both the call instruction, and the virtual function table.

The call to [EDX+0x14] calls the function who's address is 0x14 (20) bytes after where EDX points. In this example EDX points to 0x4101F8 (the virtual function table). 0x4101F8 + 0x14 equal 0x41020C. At this address (0x41020C) is the value 0x4034D0. This value (0x4034D0) is the address of the function that finally gets called. In this example, this function (at address 0x4034D0) is used by DSNX to process commands received from an IRC channel.
Function Anatomy

- Some commonality between functions
  - Primarily setup and restore activities
- Function prolog
  - Done at start of function
  - Allocate variables, save registers, etc.
  - mov edi, edi
    - 2 byte NOP equivalent for patching code at run time (hot patching)
    - Overwrite with a 2 byte relative jump, to a 5 byte long jump
- Function epilog
  - Done at end of function
  - Clean up stack, restore registers, etc.

When examining the body of a function, there is some commonality across various different functions. Primarily the setup and restoration activities associated with entering and exiting the function.

The function prolog is code at the beginning of a function that sets up the environment for the function. Activities that occur in the prolog would be allocating local variables, saving registers, etc.

The function epilog is code at the end of the function that restores the environment after the function is finished. Activities that occur in the epilog typically undo what was done in the prolog. This includes cleaning up the stack, restoring registers, etc.

One common thing you will see in the prolog of code compiled with Microsoft compilers is the instruction "mov edi, edi". This is essentially a two byte equivalent for a NOP instruction. These two bytes are used to hot patch an executable when it is running in memory, without need to stop and restart the program.

The way hotpatching works, is that the two byte "mov edi, edi" is overwritten with a two byte relative jump. The relative jump transfers control to another jump that can jump anywhere in memory. This long jump instruction is typically located five bytes prior to the start of the function being patched. When the function is not patched, the five bytes that hold a relative jump will be 5 NOP instructions. The reason a two-byte instruction ("mov edi, edi") was chosen was to ensure that the instruction pointer (from a multi-threaded process) won't be pointing to the middle of an instruction during the hotpatching process.
Stack Usage

- ESP points to next item on stack
  - Changes with each push/pop
- Use EBP as unchanging reference
  - mov ebp, esp in function prolog
  - Need to save EBP (push EBP) first
  - Called "frame pointer"
- EBP - value = local variable
  - Registers are also used for local variables
- EBP + value = parameter
  - Don’t forget about SFP
  - Clean up the stack
  - Add to ESP, pop values, ret <value>
    - LEAVE mov esp, ebp and pops EBP

The stack is typically used to store local variables, as well as parameters passed in to the function. The problem is that the value in the ESP register may change, during the function’s execution. Referencing values on the stack becomes rather difficult and complex. To alleviate this, the EBP register is used as an unchanging reference to a specific part of the stack. You will sometimes see the EBP register referred to as the “frame pointer”. One of the activities that occurs in the function prolog is to copy the current value of the stack pointer (ESP) to the EBP register. However, prior to copying ESP to EBP, the EBP register must first be saved onto the stack (so it can be restored later.) The copy of EBP that was saved on the stack is called the Saved Frame Pointer (SFP).

By setting up EBP in the function prolog in this manner, it means that when you see code reference EBP minus some value (e.g. [EBP-8]) it is accessing a local variable. When you see code reference EBP plus some value (e.g. [EBP+8]) it is referencing a parameter that was passed in. In addition to the stack, registers may also be used for local variables.

When cleaning up the stack, there are a few tricks compilers use. Compilers may pop variables off of the stack. It is also common to see a value added to ESP, the use of the ret instruction (which can also pop bytes off of the stack), and the leave instruction. The leave instruction essentially undoes the stack prolog.
Here is a diagram that shows how parameters and local variables are laid out on the stack. Lower memory addresses (e.g. 0x00) are on the top, and higher memory addresses (e.g. 0x4000) are on the bottom. Parameters that were passed into the function are labeled "arg0", "arg1", and "arg2". Local variables are labeled "var0", "var1", and "var2". The return address is labeled "ret".

Since both function parameters and local variables are on the stack, it may be tempting to reference them relative to the ESP register. However, the value of the ESP register changes with each push and pop instruction. Calculating distances from ESP at runtime can be tricky. To simplify this, many compilers will use the EBP register as an unchanging reference into the stack. However, to use the EBP register, the existing value must first be saved onto the stack. You can see this in the diagram with the box labeled "sfp" (saved frame pointer). After the value in EBP has been saved, the value in ESP can be copied into EBP. EBP will now remain unchanged (as will the relative distances from EBP to parameters and local variables) while ESP can vary with each push/pop/etc.

Once the stack has been configured as described, local variables (which are allocated after EBP has been set) will have smaller addresses (since the stack “grows” up towards smaller addresses). As a result, these will often be written as EBP minus a value. Function parameters which were push onto the stack prior to EBP being set will have larger addresses. Consequently, these will often be referenced as EBP plus a value.

When using IDA Pro, the disassembler will provide text based names for parameters and variables. The names for parameters are arg_XXX, where XXX is the relative distance from EBP. The names for variables are var_XXX, where XXX is the relative distance from the first local variable. So [ebp-arg_8] is usually the first parameter, and [ebp+var_0] is usually the first local variable.
Stack Usage Example

<table>
<thead>
<tr>
<th>push</th>
<th>ebp ←</th>
<th>Save EBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>ebp, esp ←</td>
<td>Save ESP in EBP</td>
</tr>
<tr>
<td>push</td>
<td>ecx ←</td>
<td>Allocate space for local variable</td>
</tr>
<tr>
<td>mov</td>
<td>eax, [ebp+8] ←</td>
<td>Three parameters passed into the function</td>
</tr>
<tr>
<td>add</td>
<td>eax, [ebp+0Ch] ←</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>eax, [ebp+10h] ←</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>[ebp-4], eax ←</td>
<td>Save result in local variable</td>
</tr>
<tr>
<td>mov</td>
<td>eax, [ebp-4] ←</td>
<td>Copy result to EAX (return)</td>
</tr>
<tr>
<td>mov</td>
<td>esp, ebp ←</td>
<td>Restore ESP</td>
</tr>
<tr>
<td>pop</td>
<td>ebp ←</td>
<td>Restore EBP</td>
</tr>
<tr>
<td>retn</td>
<td>←</td>
<td>“leave”</td>
</tr>
</tbody>
</table>

Here is an example function and how it uses the stack.

The first thing that happens is that the EBP register is saved by pushing it onto the stack. Then the ESP register is copied into the EBP register. Then space is allocated on the stack for a single local variable. The space is allocated by the “push ecx” instruction, which increments the ESP register by 4. An “add esp, 4” would be functional, but require more space. Next there are three references to parameters passed into the function. Namely [EBP+0x8], [EBP+0xC], and [EBP+0x10]. If you were to view this example in IDA Pro on your own system, you would likely see [EBP+arg_8], [EBP+arg_C], and [EBP+arg_10] respectively.

The first parameter [EBP+0x8] is copied into the EAX register. Then the other two parameters are added to the EAX register. After the additions the result (in EAX) is copied into a local variable, [EBP-0x4]. The local variable is then copied back into EAX. (This was done as a result of turning optimizations off). If you were to view this on your own system, you would likely see [EBP+arg_0] instead of [EBP-0x4]. IDA Pro does this to facilitate understanding of what the variables are likely used for.

After the return value has been set up, the ESP register is restored. The EBP register is also restored, by popping it off of the stack. Since these two sets of instructions occur so frequently in a function’s epilogue, there’s also a dedicated instruction to implement them: “leave”, which you sometimes see in place of “mov esp, ebp” and “pop ebp”. Finally the function returns.

Looking at this code, can you tell what it does?
int somefunc(int a, int b, int c) {
    int d;
    d = a + b + c;
    return(d);
}

This is the source code for the function that was on the previous slide. As you can see, it takes three parameters (a, b and c) and adds them together, storing the result in a local variable (d). The return value of this function is the result of the additions, which is stored in the local variable d.

Were you able to figure out the previous slide? If you haven't already done so, take a moment to try and identify which instructions on the previous page correlate to instructions on this page.
Exercise 5:
Identify Function Components

• Examine two small functions
  – Goal: identify different components and stack usage
• Examine routines to send irc messages in Sdbot
  – sdbot.exe

For this exercise you will examine two small functions. The goal is for you to become comfortable identifying the different components of functions, and how the stack is used.

You will examine the same specimen as before, except this time you will examine code in the functions used to send traffic to IRC servers.
Questions
Identify Function Components

- Functions between:
  - 0x405572 – 0x4055CC
  - 0x405682 – 0x4056DF
  - Neither function returns anything
- Function prolog:
  - Identify number of parameters, local variables on the stack, and registers used
  - Identify addresses and instructions used to save old frame pointer, and allocate a new frame pointer
  - Identify addresses and instructions used to save registers used in the function
- Identify addresses of function body
- Function epilog:
  - Identify addresses of instructions used to restore registers used
  - Identify addresses and instructions used to restore the frame pointer

The first function is between 0x405572 and 0x4055CC. The second function is between 0x405682 and 0x4056DF. Neither function returns anything, so the value in the EAX register is ignored. For each function, answer the following questions:

In the function prolog:
- Identify the number of parameters, local variables on the stack, and registers used.
- Identify the addresses and instructions used to save the old frame pointer, and allocate a new frame pointer.
- Identify the addresses and instructions used to save registers that are used in the function.

Identify the addresses of the body of the function.

In the function epilog:
- Identify the addresses of the instructions used to restore registers
- Identify addresses and instructions used to restore the frame pointer
Hints
Identify Function Components

- Parameters are EBP+value
- Local variables on stack are EBP-value
- Registers are saved to the stack
- EBP is the frame pointer
  - Save EBP on stack
  - Allocate new frame pointer (from ESP)
- Restore registers from stack
- Restore frame pointer from stack
- Push and pop save and restore from stack

Here are some hints, in case you get stuck.

Parameters passed into the function, are referenced as a positive value from the frame pointer (EBP+value). Local variables on the stack are referenced as a negative value from the frame pointer (EBP-value). Registers that need to be saved, are saved to the stack.

The EBP register is used as the frame pointer, so it too must be saved on the stack. To allocate a new frame pointer, the value from the ESP register is used.

In the function epilog, registers, including the frame pointer, are restored from the stack.

The push and pop instructions save and restore values to and from the stack.
You can locate the code by loading subbot.exe in OllyDbg or IDA Pro. Here's what you will see if you look at this code block in IDA Pro. (The first function.)
You can locate the code by loading sbot.exe in OllyDbg or IDA Pro. Here's what you will see if you look at this code block in IDA Pro. (The second function.)
Answers (1)
Identify Function Components

- For first function
  - 2 parameters (ebp+0x8, ebp+0xC)
    * [ebp+s] and [ebp+arg_4]
  - 1 local var, on stack (ebp-0x200)
    * [ebp+buf]
  - 2 registers used (edi, eax)
- To save frame pointer:
  - push ebp (0x405572)
  - mov ebp, esp (0x405573)
- To save registers: push edi (0x40557B) (eax not saved)
- Function body: 0x40557C – 0x4055C5
- To restore registers: pop edi (0x4055CA)
- To restore frame pointer: leave (0x4055CB)

Here are the answers to the exercise.

For the first function, there are two parameters (ebp+0x8 and ebp+0xC), one local variable on the stack (ebp-0x200), and two registers that are used (edi and eax). It is often easier to determine the number of parameters passed to a function by examining the code that calls the function. In IDA Pro, [ebp+s], [ebp+arg_4] and [ebp+buf] are symbolic names for [ebp+0x8], [ebp+0xC] and [ebp-0x200] respectively. To view this in IDA Pro, click on the text name and hit the letter 'q'.

The frame pointer is saved with two instructions. The first instruction is at 0x405572 (push ebp), and the second instruction is at 0x405573 (mov ebp, esp). The edi register is saved at 0x40557B (push edi). The eax register is not saved.

The body of the function is between 0x40557C and 0x4055C5.

The edi register is restored at 0x4055CA (pop edi). The frame pointer is restored at 0x4055CB (leave). The leave instruction performs two tasks. First the ESP register is restored (the value in the EBP register is copied into ESP). Second, the EBP value is restored by popping it off of the stack.
You can locate the code by loading sdbot.exe in OllyDbg or IDA Pro. Here's what you will see if you look at this code block in IDA Pro, with the answers overlaid. (The first function.)
Answers (2)
Identify Function Components

- For second function
  - 3 parameters: (ebp+0x8, ebp+0xC, ebp+0x10)
    - [ebp+s], [ebp+arg_4] and [ebp+arg_8]
  - 1 local var. on stack: (ebp-0x200)
    - [ebp+buf]
  - 2 registers used: (edi, eax)
- To save frame pointer:
  - push ebp (0x405682)
  - mov ebp, esp (0x405683)
- To save registers: push edi (0x40568B) (eax not saved)
- Function body: 0x40568C - 0x4056D8
- To restore registers: pop edi (0x4056DD)
- To restore frame pointer: leave (0x4056DE)

For the second function, there are three parameters (ebp+0x8, ebp+0xC, and ebp+0x10), one local variable on the stack (ebp-0x200), and two registers are used (edi and eax). In IDA Pro, [ebp+s], [ebp-arg_4], [ebp-arg_8] and [ebp+buf] are symbolic names for [ebp+0x8], [ebp+0xC], [ebp+0x10] and [ebp-0x200] respectively. To view this in IDA Pro, click on the text name and hit the letter 'q'.

There are two instructions to save the frame pointer. The first instruction is at 0x405682 (push ebp). The second instruction is at 0x405683 (mov ebp, esp). The edax register is saved at 0x40568B (push edi). The eax register is not saved.

The body of the function is between 0x40568C and 0x4056D8. The edi register is restored at 0x4056DD. The frame pointer is restored at 0x4056DE (leave). The leave instruction performs two tasks. First the ESP register is restored (the value in the EBP register is copied into ESP). Second, the EBP value is restored by popping it off of the stack.
You can locate the code by loading sdbot.exe in OllyDbg or IDA Pro. Here's what you will see if you look at this code block in IDA Pro, with the answers overlaid. (The second function.)
Common Optimizations

- Dead code elimination
  - Don't include what isn't used
- Use registers instead of local variables
- Loop unrolling
  - Instead of looping over code block 12 times, loop twice over 6 copies of the code block
  - Space vs. time trade-off
- Reuse variables on the stack
  - If two functions take the same argument(s) no need to clean up

There are several optimizations that a compiler can take when transforming source code into the target language. This is an active area of research in the compiler and theoretical computer science community.

One example of a compiler optimization is to remove dead code. Essentially this means that if there is source code that isn't called by the program, the compiler doesn't include it in the output. Some compilers will include dead code, so don't be surprised if you see code that is never accessed.

Another common compiler optimization is to use only registers for local variables instead of the stack. This works if there are a few number of local variables.

The technique of loop unrolling takes advantage of the classic space vs. time tradeoff. In this optimization, a code block is copied a number of times and then the number of loops is reduced. Instead of looping over a block of code 12 times, make 6 copies and loop twice. This saves on the overhead of having to transfer control 12 times.

Compilers have also been known to reuse variables on the stack. If two functions take the same arguments, there is no need to clean up the stack.
Clearing and Swapping

- Any value XOR'd with itself is 0
- Efficient way to zero (clear) a register
- Faster than mov reg, 0x0
- XCHG exchanges two registers
  - Swap the values
- Faster than a move

The XOR instruction performs a bitwise exclusive-or. One neat mathematical fact is that any value XOR'd with itself is always zero. Compilers take advantage of this fact and use it as a way to zero (clear) a register. It's actually faster than a "mov reg, 0" instruction. You'll also see this technique used in shellcode development. When developing shellcode it's normally advisable to avoid the null byte (0x00) since many standard C string functions use 0x00 to mark the end of a string.

The XCHG instruction exchanges the values of two registers. In essence it swaps the values of the registers, without needing a third register. It's faster than doing a plain move (mov). Compilers will also use this on occasion.
Multiply and Divide with SHR and SHL

- Multiplying and dividing by two is fairly common
- SHR: shift bits right
  - Equivalent to dividing by $2^x$
  - shr eax, 3: Divide EAX by 8
- SHL: shift bits left
  - Equivalent to multiplying by $2^x$
  - shl eax, 3: Multiply EAX by 8

The shift right (shr) and shift left (shl) instructions can be used for bit manipulation purposes, but they can also be used for certain types of multiplication and division.

For every bit shifted to the right, it is equivalent to dividing by 2 to the power of how many bits were shifted. For example the instruction "shr eax, 3" which shifts the contents of the EAX register to the right by 3 bits is equivalent to dividing by $2^3$ or 8.

Similarly, for every bit shifted to the left, it is equivalent to multiplying by 2 to the power of how many bits were shifted. So the instruction "shl eax, 3" which shifts the contents of the EAX register to the left by 3 bits is equivalent to multiplying by $2^3$ or 8.

Since SHR and SHL can be used for both arithmetic as well as bit manipulation, it isn't always apparent what type (arithmetic or bit manipulation) was intended. Instead you'll have to look at the context, how the data is used, to reach a conclusion.
The LEA instruction (load effective address) is commonly used throughout code, yet it can be one of the most difficult instructions to fully understand. Simply stated, LEA calculates an effective address, and stores the address in a register. Recall that an effective address is a computed address. The catch is that LEA doesn't care if the result is a valid memory address or not.

It's common to see the use of LEA with storing the address of a local variable. For instance "lea eax, [ebp-8]" stores the address of the first local variable in the EAX register. This is faster than the two instructions it would normally require ("mov ebp, eax" and "sub eax, 8").

Since LEA doesn't care if the effective address is a valid memory address or not, LEA can also be used for efficient calculations. For instance the instruction "lea ecx, [eax+ebx+2]" performs two additions and a move in a single instruction.
Advanced Math

- Simple calculations can create large instructions
  - Faster than doing idiv/imul
- Difficult to reverse
  - Lots of code to follow
  - Try treating it as a "black box"
- Division by Invariant Integers Using Multiplication
  - http://citeseer.ist.psu.edu/granlund94division.html

There are some very complex calculations that come as a result of compiler optimizations. Some very simple calculations can create a large number of instructions.

Unfortunately, it can become very difficult to reverse this type of code, as it has many instructions to follow, and no documentation (e.g. no system calls, etc.) One solution is to treat the segment of code as a "black box." Examine what goes in, see what comes out and come up with a theory as to what the code segment does. Once you have a theory, you still have to test it. So try running a few different values through the code segment and see if they match your theoretical result.

As an example of how confusing some of the advanced math optimizations can become, see the paper titled "Division by Invariant Integers Using Multiplication." It talks about optimizations a compiler can make if the source code is doing division by a constant value. In this case, the paper describes how to transform division into a series of multiplication instructions!
64-Bit Code

- x86-64 builds upon x86
  - Code has to be compiled for 64-bit
- General purpose registers expanded to 64 bit
  - RAX, RBX, RCX, RDX, ...
- New registers
  - r8, r9, r10, r11, r12, r13, r14, r15
- Instructions (usually) have 'q' (e.g. movq)
  - Stack operations choose implicitly
- New addressing (RIP + displacement)
  - Position independent code is easier to write
  - Common for DLLs

One question often asked is how 64-bit code impacts reverse engineering. The x86-64 architecture builds upon the x86 architecture. Meaning 32-bit code will still work (and run) on 64-bit processors. While code must be specifically compiled for 64-bit architectures, there are some differences to note. Additionally, even though 64-bit malware is not the norm, it does exist.

The first major difference between x86 and x86-64 is that the size of the general purpose registers have been expanded to 64-bits. The new 64-bit version of EAX, EBX, ECX, EDX, etc. can be accessed as RAX, RBX, RCX, RDX, etc. There are also 8 new registers, r8 through r15 (inclusive).

The 64-bit version of several instructions have the letter "q" (for quadword) appended. The exception is stack related instructions (e.g. ret, leave, etc.) because size is implied.

The change that you will likely see most often is the use of a new addressing mode, called “RIP-relative addressing”. The RIP register (64-bit version of EIP) can now be used to reference memory locations.

This addressing mode provides a serious advantage to position independent code (PIC). PIC is commonly used in DLLs, which may not always be loaded at their desired address. To deal with the dynamic nature of DLL layout, the OS loader includes information about the location of runtime variables, and instructions that access those variables. Using RIP-relative addressing, much of this information is no longer necessary, reducing both load and run times.
Tying it All Together: Code Examinations

- There is no "standard" methodology for examining code
  - Just like there is no "standard" methodology for examining network traffic, performing host forensics, etc.
- It's an iterative process
  - Start with what you know and work from there
- You will find functions you don't know
  - For system calls, there is MSDN, Google, etc.

Realize that there is no "standard" methodology for examining code. Just as there is no standard methodology for performing a forensic examination, examining network traffic, etc.

Instead, like any other form of analysis, code examinations tend to be an iterative process. Start with what you know, and work from there. As you understand the role of new functions and segments of code, you will be able to build a better picture of what the specimen does.

Part of reversing involves research. There will be many times when you encounter a system or library call, and you won't know what the function does. In this case, don't hesitate to use available resources such as Google, the Microsoft Developers Network (MSDN), compiler documentation, etc.

When you start reversing a new specimen, you should probably have (at least) one browser window open at the website of your favorite search engine.
Tips and Tricks (1)

- Look at list of libraries/functions imported
  - Might give a hint as to capabilities
- Examine embedded strings
  - Easy to fake, so be wary
- Start with simple (small) functions that make a few system calls
  - Can be easier to understand
  - Or analyze parts of functions
  - Enough small parts make a whole

Even though there is no standard methodology for reversing, there are a number of tips and tricks that can make your life easier.

One of the first things I do when examining a new specimen is to look at the list of libraries and functions that are imported. This can often times give a hint as to the capabilities of the specimen. If you see the specimen imports a networking library, there is a good chance it has network functionality.

You can also examine the embedded strings. Although it is a trivial exercise to fake strings and hide the true strings, so be wary of the amount of trust you place in strings. I tend to use embedded strings as more of an "investigative lead" than a "definite conclusion."

It can be easier to start by analyzing small functions that make a few system calls, rather than very large functions. This is because system calls tend to be fairly well documented. Alternatively, you can also analyze functions in pieces, breaking a larger problem into a series of smaller problems. As you understand some of the pieces, it tends to become easier to understand other pieces. Remember, enough small pieces make a whole.
Tips and Tricks (2)

- Stepping through with a debugger can be very useful
  - Easier to see what is where, skip over blocks of code, etc.
- Maintain good notes
  - Many debuggers / disassemblers let you annotate the assembly code
  - When you figure out what a function does, rename it

Sometimes it is very helpful to step through code with a debugger. It allows you to see what type of data goes into and out of various pieces of code. It can also be easier to recognize loops, simple inlined functions (e.g. strcpy, strlen, etc.) and so forth.

One important point is to maintain good notes. I maintain both paper AND electronic notes. Many disassemblers and debuggers let you annotate the assembly, or at least insert comments. I like to annotate the specimen I'm examining with pseudo-C code. By the time I finish examining a specimen I can sometimes practically recreate the source code in C.

One thing that helps is to give variables and functions meaningful names. When you understand what a function does, rename it to something meaningful. It's easier to understand the function name "SendAndEncrypt" than "sub_4034889".

112
Tips and Tricks (3)

- There is no "easy" way to figure out the purpose of a block of code
  - Sometimes function calls help
  - Think about context (where did it come from, incident details, etc.)
  - Use what you know from behavioral analysis
- If you get stuck, don't give up
  - Examine another part of the code
  - Work on a small part of a function
  - Look at the data that goes in and out in a debugger

Understand that there is no magic when reversing code. There is no easy way to figure out what a block of code does, from a high level. Sometimes function calls may help, but they are not always available.

It may also help to think about the context surrounding the specimen. For instance, where did the specimen come from? Are there any relevant incident details? Do you have any network traces that you can examine? Don't forget to use the knowledge you gained during a behavioral analysis.

I want to stress this next point... If you get stuck, don't give up! It can be frustrating at times, trying to reverse a rather complex piece of code. As mentioned before, there is no magic wand to reading code. If you do get stuck, try examining another piece of code, or work on a small part of the function. Also look at the data that is used by the code, by stepping through the code with a debugger. Also, don't be afraid to ask for help. Sometimes another set of eyes can catch something that might be staring you straight in the face.
Next Steps

- We've covered a lot of concepts to reverse code
  - For more x86 instructions, see appendix
- The next step is to start examining common techniques used by malware

Believe it or not, we've covered a majority of the reversing concepts that you're likely to encounter. While we didn't spend much time on individual instructions (we could be here for weeks) there is an appendix to today's book with explanations of a number of individual instructions.

Even though it may not seem like it, many of the concepts we've covered relate directly to analyzing malware. We're now going to switch gears and examine some of the different ways (from a code perspective) malicious code (such as rootkits) works.
Let’s continue by examining some of the anti-disassembling techniques malware authors may include in their specimens. We’re drill deeper into this topic in the FOR610.4 section of the course.
Anti-Disassembling

We're now going to examine some of the techniques used by malicious code to perform various activities. Remember: there are many different ways to accomplish a given task. We'll focus on the most common ways.

The core of this section is on "malicious activities" such as DLL injection, API hooking, sniffing, etc. Malicious code such as bots combine multiple techniques into one program.

The first type of malicious activity we will examine is anti-disassembling.
Disasmblers

- Disassemblers translate from binary machine code to readable assembly
- Used in many reversing tools
  - E.g. IDA Pro, OllyDbg, ImmDbg, etc.
- Biggest problem is (correctly) identifying code
  - Self modifying code, non standard instruction usage, etc.
- Two primary categories
  - Linear sweep
  - Recursive traversal
- Heuristic approaches to identify code vs. data

Assemblers are tools that translate human readable assembly (mnemonics) into binary machine code. Disassemblers do the opposite, that is they translate binary machine code into human readable assembly mnemonics.

Disassemblers are a common staple of many reversing and analysis tools. Even debuggers, such as OllyDbg, Immunity Debugger, WinDbg all use disassemblers at one stage or another.

While the disassembly process may seem straightforward, the largest problem that disassemblers encounter is correctly determining what parts of the program are actually code. This problem surfaces in a number of scenarios, including self-modifying code, non-standard instruction usage, etc.

In order to understand anti-disassembly tactics, it is helpful to understand the two basic categories of disassemblers, linear sweep and recursive traversal. A linear sweep disassembler treats everything in a code segment as code. The process is fairly straightforward, start at the beginning of each code section and treat everything as an instruction.

The primary drawback of linear sweep disassemblers is that they don't account for branch instructions. In contrast, recursive traversal disassemblers will follow code branches. While it may seem that recursive traversal disassemblers are the solution, they have their own problems. Most notably, recursive traversal disassemblers have problems dealing with branches calculated at runtime.

In addition, some analysis tools (e.g. IDA Pro and OllyDbg) include a hybrid approach of both linear and traversal disassembling. Another possibility is to include a heuristic based approach to identifying code.
Anti-Disassembling

- Anti-disassembling aims to hinder disassemblers and static analysis
  - Debugging can help
- Several different ways
  - Jumping into the middle of an instruction
  - Jumping via return
  - Modifying PE headers

The term anti-disassembling refers to steps that a code author may take to hinder the work of disassemblers, or a static code analysis. Use of a debugger can help alleviate several problems, although there are also anti-debugging tactics that can hinder debugging sessions as well.

There are several different ways to hinder disassembly. We'll take a look at some of the more common techniques.
Junk Code

- Insert series of useless instructions between real instructions
- Surround useless instructions with pusha/pop that
  - Save and restore registers
- Can really slow down an analyst

One technique that malicious code authors use to slow down a static analysis is to insert series of useless instructions, between real instructions. The useless instructions (also known as "junk code") can distract an analyst performing a static analysis.

In order for the junk code not to have a functional side effect on the real program, the registers and memory locations modified inside of the junk code shouldn't be used outside of the junk code. In many circumstances this can be quite limiting.

One way to alleviate this restriction is to save all of the registers, perform a series of useless instructions, restore the registers, and then execute a real instruction. One technique to do this is to use the pusha and pop that instructions. The code calls pusha before the start of the junk code to save of the registers, and then calls popa after all of the junk code to restore the registers.
Here is an example of junk code. This was taken from the Honeynet scan of the month challenge 33.

At the top, the pusha instruction is called. This will save the contents of the registers to the stack. Then 726 bytes later (roughly 150 instructions), the popa instruction is called. This restores the contents of the registers from the stack. After the popa instruction is the real instruction (mov eax, [eax+2]).

After the real instruction, there is another pusha instruction, which starts more junk code.
Branch into Instruction

- Branch into a multi-byte instruction
  - Can use jmp/call/etc.
- Fools many disassemblers
  - Bytes considered code changes at runtime
  - OllyDbg sometimes halts
  - IDA Pro prompts about EIP inside instruction (debugging)
- Common with many executable packers
- Used in shellcode to get current location

Another common tactic used to fool disassemblers is to jump into the middle of an instruction that is multiple bytes in length. This tactic will fool many disassemblers, as it changes which bytes are actually code, at run time.

Different tools approach this tactic differently. Depending on the circumstances, OllyDbg will sometimes halt with an error. In IDA Pro, during a debugging session, the analyst will be notified by a pop up box, that EIP now points to a part of memory that was not previously considered to be an instruction.

There are a couple of instances where this tactic is fairly common. The code that is inserted by several executable packers has been known to use this technique. In addition, shellcode developers often use this tactic to determine their current location in memory.
Branch into Instruction Example

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x417000</td>
<td>E8 FF FF FF FF call near ptr shellcode+4</td>
<td>Before call instruction</td>
</tr>
<tr>
<td>0x417095</td>
<td>C3 retn</td>
<td></td>
</tr>
<tr>
<td>0x417095</td>
<td>; END OF FUNCTION CHUNK FOR</td>
<td></td>
</tr>
<tr>
<td>0x417096</td>
<td>pop eax</td>
<td></td>
</tr>
<tr>
<td>0x417000</td>
<td>E8 shellcode db 0E8h</td>
<td>After call instruction</td>
</tr>
<tr>
<td>0x417001</td>
<td>FF db 0FFh</td>
<td>EIP points here</td>
</tr>
<tr>
<td>0x417002</td>
<td>FF db 0FFh</td>
<td></td>
</tr>
<tr>
<td>0x417003</td>
<td>FF db 0FFh</td>
<td></td>
</tr>
<tr>
<td>0x417004</td>
<td>; -------------</td>
<td></td>
</tr>
<tr>
<td>0x417004</td>
<td>FF C3 inc ebx</td>
<td></td>
</tr>
<tr>
<td>0x417004</td>
<td>; END OF FUNCTION</td>
<td></td>
</tr>
<tr>
<td>0x417006</td>
<td>5B pop eax</td>
<td></td>
</tr>
</tbody>
</table>

Here is an example of a call into the middle of an instruction.

The first screen capture is from the IDA Pro debugger prior to executing the call instruction. As you can see, the call instruction jumps into the last byte of itself (at offset 0x417004). The bytes that represent the call instruction are 0xE8FFFFFFF. The next instruction is a retn, which is represented by the byte 0xC3.

The second screen capture is what is shown after executing the call instruction. The first four bytes of the original call instruction (0xE8FFFFFFF) are now interpreted as data. The last byte of the call instruction (0xFF) is now combined with the next byte (0xC3) to form the instruction inc ebx.

The code shown on this slide is an example of how exploit developers can get the current value of the EIP register. When the call instruction executes, it pushes a copy of EIP onto the stack. After the call, the pop instruction (at address 0x417006) will pop the previously stored value of EIP in the EAX register.
Jumping via Ret

- ret returns from a function
  - Pop current value on stack into EIP
- Manually push a value onto stack
- Transfer control to new location
  - Effectively an unconditional jump
- Confuses disassemblers, since ret normally ends a function
  - E.g. 2 rets

Another tactic used to confuse disassemblers is to use a return instruction as a jump. A return instruction (ret or retn) is commonly used to return from a function (indicating the function has finished). Essentially, a return instruction pops the last value on the stack (pointed to by ESP) into the EIP register.

The idea behind this tactic is to manually push a value onto the stack, and then execute a return instruction. This transfers control to the value that was pushed onto the stack, effectively executing an unconditional jump.

This is confusing to disassemblers as return instructions typically mark the end of a function. There are also variants of this, where multiple returns are executed (e.g. two returns.)
This is a screenshot from OllyDbg examining a jump effected by the return instruction.

As you can see, OllyDbg shows two functions (one from 0x401000 through 0x40100B, and the second from 0x401011 through 0x40101E) and some data (the ASCII value "hide").

Stepping through the code, the call instruction at address 0x401001 calls the next instruction (pop ebx). The call instruction pushes EIP onto the stack, and then transfers control to the next instruction. The pop ebx instruction pops the value of EIP (which is on the stack) into ebx. The ebx register now holds the address 0x401006.

The "add ebx, 6" instruction changes the value in EBX to 0x40100C (0x401006 + 0x6 = 0x40100C). This is the address of the first byte of the ASCII string "hide" (immediately following the retn instruction.)

The new value of EBX (0x40100C) is pushed onto the stack, and finally the retn instruction is executed. When the retn instruction is executed, the last value on the stack (0x40100C) is popped into EIP.

The ASCII string "hide" is equivalent to the instruction pop 0x00656469, and this is what the processor executes. The rest of the instructions (0x401011 through 0x40101E) are executed. The real end of the function is the last retn instruction at address 0x40101E.
Modify PE Headers

- PE specification defines several fields to describe executable file
  - E.g. size of each segment, entry point address, imports, etc.
- Windows loader is fairly flexible
  - Many disassemblers/debuggers aren't
- Make (portions of) a file "unreadable"
- Can be confusing to an analyst used to common/normal values

The Portable Executable (PE) specification defines a number of fields that are used to load and interpret executable files. For example, the size of a segment, the first instruction to execute (entry point), functions to import, etc.

The Windows loader is fairly flexible in the interpretation of many fields, however, many disassemblers and debuggers aren't as flexible. This has the effect of making portions of the file "unreadable" to the analysis tool.

A side effect of using non-standard values in the PE headers, is that it can be confusing to an analyst who is used to the common/normal values.
Modify PE Headers Example  
(Honeynet SOTM 33)

<table>
<thead>
<tr>
<th>Address of Entry Point</th>
<th>00000400h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base of Code</td>
<td>000001000h</td>
</tr>
<tr>
<td>Base of Data</td>
<td>000002000h</td>
</tr>
<tr>
<td>Image Base</td>
<td>00000000h</td>
</tr>
<tr>
<td>Size of Stack Commit</td>
<td>000002000h</td>
</tr>
<tr>
<td>Size of Heap Reserve</td>
<td>000000000h</td>
</tr>
<tr>
<td>Size of Heap Commit</td>
<td>000001000h</td>
</tr>
<tr>
<td>Loader Flags</td>
<td>ABDBFFDDhh</td>
</tr>
<tr>
<td>Number of Data Directories</td>
<td>DFFDDDDhh</td>
</tr>
</tbody>
</table>

This example is from the Honeynet scan of the month challenge 33.

Looking at the entry point address, we can see a non-standard value of 0x1E:2000. The typical entry point for a portable executable is in the range 0x40000. While this doesn't really stop most reversing tools, it can confuse an analyst.

The values for the loader flags (an obsolete field) and the number of data directories are both abnormal. Per the specification, the loader flags field must be zero, and the number of data directories is unusually large. These values combined will crash older versions of Soft ICE.

Examining the section headers, there is a section named "NicolasB" which has a raw data size of 0xEFED0D8F. This would imply that the size of this section on disk is 0xEFED0D8F bytes in size. Attempting to open this executable file crashes IDA Pro v5.2. To avoid this, either update the field in the executable file, or don't load this section in IDA Pro.

To manually load sections in IDA Pro, when opening an executable file, select the "Manual Load" checkbox in the options area of the "Load a new file" dialog window.
### FOR610.3 Roadmap

- Core Reversing Concepts
- Assembly Primer
- Anti-Disassembling
- Exception Handling
- Hands-On Exercises (Throughout)

1st half of FOR610.3

... then 2nd half of FOR610.3

Now, let's examine how malware may use Structured Exception Handling to its advantage.
Structured Exception Handling

This section examines structured exception handling.
Structured Exception Handling

- Windows error handling mechanism
  - Call programmer defined functions when exceptions occur
- Frame based exception handling
  - Exception handler information stored on the stack
  - Use of __try and __except keywords in MS VC++
  - Commonly referred to as SEH
- Vectored exception handling
  - Non stack based
  - Doesn't require keywords
- Frame based exception handling is often abused
  - Exploit code (bypasses return pointer protection)
  - Code packers (trigger exception, exception handler is part of the code)
  - We will see an example of this in FOR610.4

Structured exception handling (SEH) is a Microsoft Windows error handling mechanism. SEH allows programmers to define functions that are called when an exception occurs. A developer can use SEH to handle both software and hardware errors.

There are two types of exception handling that Microsoft offers developers, frame based and vectored. Frame based exception handling stores information about the exception handlers on the stack frame. Compilers must implement their own keywords to provide access to SEH from code. In Microsoft Visual C++ the keywords are __try and __except. Frame based exception handling has been around for many years, and is what is meant by the term "Structured Exception Handling".

Vectored exception handling is an extension to SEH, which unlike frame based exception handling, does not use the stack frame. Vectored exception handling is implemented as library code, and does not require compiler defined keywords.

SEH is often abused by malicious code. Overwriting the exception registration record on the stack is one way exploit writers can bypass return pointer protection mechanisms. Executable packers (e.g. PECompact, Armadillo, etc) also abuse SEH mechanisms as a form of anti-debugging and anti-disassembling. For instance, the exception handler may be used as a non-standard branching mechanism, making the exception handler a part of the flow of execution. When an exception is triggered, execution continues in the exception handler.

We will see an example of abusing the SEH mechanism in FOR610.4.
There are two primary components of structured exception handling. The first is the _EXCEPTION_REGISTRATION structure. This structure sits on the stack, and is used by the operating system, to determine where the exception handling function sits in memory.

The _EXCEPTION_REGISTRATION structure has two components, a pointer to an exception handler function, and a pointer to the previous _EXCEPTION_REGISTRATION record. In this manner, multiple exception handlers form a linked list, and exceptions can be propagated up the exception handling chain.

When an exception handler is called, the function prototype is passed four arguments. The first argument is a pointer to an ExceptionRecord, which describes the exception that occurred. The second parameter is a pointer called EstablisherFrame. This parameter points to the _EXCEPTION_REGISTRATION structure that was used to call the exception handler. The third parameter is a pointer to a CONTEXT structure, which contains information about the state of various registers. The last argument points to a structure that contains information about the exception dispatcher.

The EstablisherFrame parameter is of interest to exploit developers, since it holds a pointer to the _EXCEPTION_REGISTRATION structure that was used to locate the exception handler. When the exception handler is called, this argument is located at ESP+8. This field is useful to exploit developers, as they are able to control the contents of the _EXCEPTION_REGISTRATION during a stack based buffer overflow attack.
Let's examine how an exploit developer can overwrite the EXCEPTIONREGISTRATION structure, and gain control of execution, without overwriting the return pointer. It's important to note that in all of the stack illustrations, smaller addresses are at the top, larger addresses are at the bottom. This means that for each item pushed onto the stack, ESP moves higher up the picture (the addresses become smaller). Likewise, for each item popped off the stack, ESP moves lower down the picture (the addresses become larger).

On the left, is a picture of a stack prior to a stack based buffer overflow attack. On the right, is a picture of the same stack, after the buffer overflow attack. The EXCEPTIONREGISTRATION structure depicted on this stack, comes from the Microsoft Visual C++ implementation. This compiler uses an extended EXCEPTIONREGISTRATION structure, which is why there are a number of additional fields. For purposes of the exploit, they won't get in the way.

There are three relevant components that the attacker placed on the stack, shellcode, a relative jump into the shellcode, and the address of a pop+pop+ret instruction sequence.
After the attacker has overwritten values on the stack, they need to trigger an exception. After the buffer overflow, the local variables on the stack will have new values. It is quite possible that during normal function execution, that these new values will be invalid, and result in an exception. Another possibility is for the attacker to keep writing values, off the edge of the memory page.

Once an exception is triggered, the exception handling function takes control. The diagram on the left shows what the stack for the exception handling function looks like. The ESP register points to the return address. The body of the exception handling function is the pop+pop+ret instruction sequence.

Remember that the EstablisherFrame argument points to the "pointer to previous _EXCEPTION_REGISTRATION structure" field from the EXCEPTION_REGISTRATION structure that the attacker overwrote. However, instead of having a valid pointer at this address, the attacker placed a relative jump into the shellcode.
Once the exception handling function starts executing, the first instruction is a pop. This adds 4 to the value in ESP. The ESP register now points to the ExceptionRecord argument.
The second instruction to execute is another pop instruction. This increases the value of the ESP register by 4. The ESP register now points to the EstablisherFrame argument. Remember that the value of this argument (EstablisherFrame) is the address of the structure the attacker overwrote, on the original stack.
The third instruction is a return (ret). This moves the value pointed at by ESP into the EIP register. Since the ESP register points to the EstablisherFrame argument, which in turn points to the relative jump, the value in EIP is now the address of the relative jump instructions that the attacker placed on the stack.

When the relative jump instruction executes, the EIP register points at the shellcode, completing the exploit.
Now that you have seen the theory behind SEH based exploits, let's see what one looks like during an analysis session. For this example, the shellcode that gets executed is from the Metasploit framework, and is 317 bytes in length.

On the top is a stack prior to being overfl owed. The EBP register points to the saved frame pointer. At EBP-12 (0x12FD78) is the address of the current exception handler. At EBP-16 (0x12FD74) is the address of the next _EXCEPTION_REGISTRATION structure.

On the bottom is the same stack, after being overflowed. At address 0x12FD78 is the address of a pop+pop+ret instruction sequence in little endian (0x7FF51037). Address 0x12FD74 contains three instructions, a relative jump backwards by 7 bytes (0xEBF9) and two NOPs (0x9090). Address 0x12FD0 contains a relative jump backwards by 328 bytes (0x8FFFEF).

The second jump is needed, because the first relative jump is a near jump, which can only jump backwards by 128 bytes, or forwards by 127 bytes. This first jump is a near jump, because there are only four bytes available, and a relative long jump requires 5 bytes.
Once an exception has been triggered, the exception handler takes control. In the upper left corner are the three instructions that were pointed to by the overflowed stack. On the right is the stack for the exception handler. As you can see, at ESP+8, is the address of the short relative jump, in little endian (0x0012FD74).

The first two pop instructions increment the value of the ESP register (by 8) to 0x12F864. The ret instruction moves this value (0x0012FD74) into the EIP register.

The EIP register now points to the relative jump instruction. Recall that this relative jump instruction is part of the stack that was overflowed on the previous page. The relative jump instruction transfers control to the second relative jump instruction. The second relative jump instruction jumps into the shellcode, which executes. You can see from the resulting netstat output, that there is indeed a listener on port 4444.
FOR610.3 Roadmap

... done with 1st half of FOR610.3

- User-Mode Rootkits
  - Keyloggers
  - Sniffers, Spoofing, Downloaders
  - HTTP Channels
  - Hands-On Exercises (Throughout)

2nd half of FOR610.3

In the second part of FOR610.3, we'll examine several common malware characteristics at the assembly level, learning to recognize common patterns by examining the use of Windows API calls.
User Mode Rootkits

We'll now examine different components of user mode rootkits.
Rootkits

- Purpose of a rootkit is to attain, maintain, and hide attacker access
  - Include stealth techniques
- Netstat, Task Manager, etc. call functions to list processes, ports, ...
  - If we can modify the data going back we're golden
- User mode rootkits do this in user space
  - Individual processes
- Kernel mode rootkits do this in kernel space
  - Modify parts of the kernel in memory or on disk

A rootkit is a collection of tools that are used to attain, maintain, and hide access by the attacker. Rootkits typically use stealth techniques to hide the attackers presence.

One way that administrators, incident responders, and investigators alike monitor activity on a system is with user space tools such as netstat, task manager, Process Explorer, etc. Each of these tools displays information about various system resources, such as what processes are running, open ports, etc.

If an attacker can modify the data that is returned by these tools, they can effectively hide their tracks. Alternatively, an attacker may be able to modify the functionality of a program (such as login) to allow illegitimate access.

If a rootkit operates completely in user space (that is, it doesn't modify parts of the kernel) it is said to be a user mode rootkit. These rootkits infect individual processes and tools. By contrast, a kernel mode rootkit modifies parts of the kernel, either in memory or on disk. These rootkits do not infect individual processes and tools, since they have already modified (parts of) the kernel.
Here is an example of a typical process. The code in user space needs to get some information from the kernel, let's say the code wants to enumerate a key in the registry.

The user code would first call a library function (Step 1). The library (in this case ntdll.dll) resides in user space (it is loaded when the program is first loaded.) The library would then make a system call to the kernel (Step 2). The kernel would then return the information to the library function (Step 3). The library function would then do any additional processing and return the result to the user code (Step 4).
Here's one way a rootkit can go through and modify the results that are returned to the user code.

When the rootkit code gets inserted into the user space, it can modify the first few bytes of the function in the library ntdll.dll (in memory, not on disk). This is possible because the library code itself resides in user space. The modification that most rootkits make is a jump into the rootkit code.

When the user code calls the modified function (Step 1), control is then transferred to the rootkit (Step 2). The rootkit then restores the first few bytes of the library function and calls that function (Step 3). The function then performs a system call to request the information from the kernel (Step 4). The kernel returns the information to the library function (Step 5). The library function then returns the information to the rootkit (which called the library function) (Step 6). The rootkit then re-modifies the first few instructions of the library function. Once this has been completed, the rootkit can update/modify the information returned from the library function and return it to the original user code (Step 7).

As you can see, there are a number of additional steps that take place when rootkits are used. This is just one method used by rootkits to intercept API functions.
Two Infection Steps

1. Get rootkit code into victim process
   a. DLL Injection
2. Intercept function
   a. API Hooking

In order for a rootkit to create the scenario we saw on the previous slide, there are two steps that need to be taken.

First, the rootkit code has to be injected into the victim process. One way this is normally done is using DLL injection.

Once the rootkit code has been injected into the victim, it has to place itself between the user code and the library function. This activity is typically called API hooking.
DLL Injection

- Injects malicious code (.dll) into another process (while running)
- Several ways to do this
  - SetWindowsHookEx
  - CreateRemoteThread/LoadLibrary

The first step a rootkit needs to take is to place malicious code in the address space of another process. The easiest (and best documented) technique to do this is by injecting a dynamically linked library (DLL) file into the address space of another process.

This method has been documented in a number of places, and has legitimate use. As is the case with many segments of code (reading files, sending data across a network, writing to files) the activity itself is not necessarily malicious. Instead, the intent and purpose of the code is what makes it malicious.

For purposes of this class, we'll look at the two most common DLL injection techniques, the first uses SetWindowsHookEx, and the second uses a combination of CreateRemoteThread and LoadLibrary.
Windows Hooks

- Message sent to user process
- User process dispatches message to appropriate hook chain
- Filter function in hook chain executes
  - Execute arbitrary code
  - Hook API functions, monitor key strokes, etc.
  - Known as a shatter attack

The first type of DLL injection we'll discuss is using a technique involving hooks.

In the graphical Windows environment, when an event happens (e.g., a mouse button is clicked, a key is pressed, a window is resized, etc.) at message is sent to the corresponding application. A hook is a mechanism for developers to intercept these messages.

When the application reads the message from its queue, it dispatches the message to a series of hook filters. Each filter has previously registered itself with the application, describing which types of message it should receive. There can be more than one hook filter that is registered to receive any event, and they are chained together, creating a hook chain.

When a filter function receives a message, it can execute arbitrary code, such as hooking API functions, monitoring keystrokes, etc.

This type of attack is known as a "shatter attack", and was described by Foon in the paper "Exploiting design flaws in the Win32 API for privilege escalation."
Attacks can use this hooking mechanism to inject DLLs into a remote process.

First, the attacking process has to load the DLL into its own memory space, and determine the address of the function that will be the filter function in the remote process. Next, the attacking process calls the SetWindowsHookEx() function to set a hook in the remote process. When doing so, the attacking process specifies the DLL and function that will act as the filter function.

Third, a message has to be sent to the victim process. This can happen intentionally, with the attacking process sending a message (e.g. using the BroadcastSystemMessage() function), or the attacking process can wait for an external message to be generated (e.g. a key press).

Once a message has been sent to the victim process, Windows loads the malicious DLL into the victim process's address space. The message is then passed to the malicious filter function, which executes the attacker's code.
Find Address of Filter Function (BWLoader)

```
mov   eax, [ebp+lpLibFileName]
push  eax          ; lpLibFileName
call  ds:LoadLibraryA        ; Load DLL into attacking process space

push  offset ProcName; "CHIProc"
mov   eax, [ebp+hmod]
push  eax          ; hModule
call  ds:GetProcAddress
```

Here is an example of DLL injection via SetWindowsHookEx() from a tool called BWLoader.

The first call for DLL injection is to LoadLibraryA to load the DLL into the attacking process's address space. Next, the GetProcAddress is called to retrieve the address of the filter function.
Install Hook
(BWLoader)

```
push 0  ; lpdwProcessId
mov eax, [ebp+hWnd]
push eax  ; hWnd
call ds:GetWindowThreadProcessId  Get ID of target thread
cmp esi, esp
call sub_41ACA3
mov esi, esp
push eax  ; dwThreadId
mov ecx, [ebp+hmod]
push ecx  ; hmod
mov edx, [ebp+lpfn]
push edx  ; lpfn
push [HH_CBT]  ; idHook
call ds:SetWindowsHookExA  Installs the filter function
```

Next, the ID of the target thread is determined by calling the GetWindowThreadProcessID. This may be different than the process ID.

Finally a call to the SetWindowsHookExA function, installs the filter function in the hook chain of the remote process.
Another type of DLL injection is to use the CreateRemoteThread() function. Here are the steps for this type of DLL injection.

First, the remote process must be opened so that it can be worked with. To do this, malware will typically use the OpenProcess() function call.

The end goal is to call LoadLibrary in the victim process. The argument that LoadLibrary takes is a string name of a DLL. This means memory must be allocated in the remote process to hold the string. To do this, malware will typically use the VirtualAllocEx function call.

Once the memory in the remote process has been allocated, the name of the DLL must be written to that location in memory. Since this memory is in another process, the WriteProcessMemory function call is typically used.
DLL Injection with
CreateRemoteThread, continued

Once the name of the DLL is written to the remote process, the address of the LoadLibrary function (in the remote process) must be located. LoadLibrary is located in the kernel32.dll file. Due to the way Windows loads processes into memory, kernel32.dll is always at the same address on the same machine. This means if we can locate LoadLibrary in the current process, we will know the address of LoadLibrary in the remote process.

To locate LoadLibrary, the code must first locate the address of the dll that contains LoadLibrary. In our case this is kernel32.dll. To locate kernel32.dll, a specimen will typically use the GetModuleHandle function call. Once the kernel32.dll file has been located in memory, a specimen will typically then call GetProcAddress to locate the address of the LoadLibrary function within the kernel32.dll file.

Once the name of the DLL is written to the remote process, and the address of LoadLibrary has been located, a specimen will call CreateRemoteThread. CreateRemoteThread will create a new thread in the remote process, and one of the arguments that this function takes is the address of a function that the new thread should run once it's been created. In this case, it will be LoadLibrary. The argument to LoadLibrary is, of course, the name of the DLL (in the remote process).
Open Process, Allocate Memory  
(kinject)

| push  | eax | ; dwProcessId |
| push  | 1   | ; bInheritHandle |
| push  | 1F0FFFFh | ; dwDesiredAccess |
| call  | OpenProcess | — Open remote process |

| mov    | eax, [ebp+nSize] |
| push   | eax | ; dwSize |
| push    | 0  | ; lpAddress |
| mov    | eax, [ebp+hProcess] |
| push   | eax | ; hProcess Allocate memory in remote process |
| call   | VirtualAllocEx | — remote process |

We will now go over an example piece of code, kinject. The kinject tool allows the user to inject arbitrary DLLs into another process.

Above, we can see the first call to OpenProcess. In this case the process id of the remote process was stored in the EAX register (and then pushed onto the stack).

Next we see call to VirtualAllocEx, to allocate memory for the name of the DLL in the remote process.
Write Memory (kinject)

```
push  0  ; lpNumberOfBytesWritten
mov   eax, [ebp+nSize]
push  eax  ; nSize
mov   eax, [ebp+lpBuffer]
push  eax  ; lpBuffer
mov   eax, [ebp+lpParameter]
push  eax  ; lpBaseAddress
mov   eax, [ebp+hProcess]
push  eax  ; hProcess
call  WriteProcessMemory  memory in remote
     process
```

Here we can see the call to WriteProcessMemory. In this case, the name of the DLL is being written to the memory in the remote process (which was allocated by the VirtualAllocEx function on the previous slide.)
Get Address (kinject)

```assembly
push    offset aLoadlibraya ; "LoadLibraryA"
add  esp, 0FFFFFFF4h  
push    offset ModuleName ; "kernel32.dll"
call   GetModuleHandleA  ; Find address of kernel32.dll
add  esp, 0ch
mov  eax, eax
push  eax  ; hModule
call   GetProcAddress  ; Find address of LoadLibraryA
add  esp, 8
mov  [ebp+lpStartAddress], eax
```

Here we see the call to GetModuleHandleA. This function is given the string "kernel32.dll" as an argument.

The call to GetProcAddress is used to find the address of LoadLibraryA. There is a small trick that the compiler used. You'll notice that the first line of code pushes address of the string "LoadLibraryA" onto the stack. Then the value 0xC is subtracted from the ESP register. After the GetModuleHandleA function call, the value 0xC is then added back to the ESP register. This effectively makes the last item on the stack, the address of the string "LoadLibraryA".

The address of the kernel32.dll library is then pushed onto the stack, and the GetProcAddress call is performed. The return value is the address of LoadLibraryA within the kernel32.dll library.
Create Remote Thread (kinject)

```
push    0                   ; lpThreadId
push    0                   ; dwCreationFlags
mov     eax, [ebp+lpParameter]
push    eax                  ; lpParameter
mov     eax, [ebp+lpStartAddress]
push    eax                  ; lpStartAddress
push    0                    ; dwStackSize
push    0                    ; lpThreadAttributes
mov     eax, [ebp+hProcess]   ; hProcess
call    CreateRemoteThread  ; Create remote thread to run
                            ; LoadLibraryA
```

Finally we see the call to CreateRemoteThread. When this call is executed, the following activities will take place:

- A new thread will be created in the remote process
- LoadLibraryA will be run in the newly created thread
- LoadLibraryA will load the malicious DLL into the remote address space
Additional Notes on DLL Injection

- **SetWindowsHookEx**
  - Only works for GUI apps
  - Inject to all apps in single call
- **CreateRemoteThread/LoadLibrary**
  - Works for all apps (not just GUI)
  - Only works on NT/XP based systems
- **Vista**
  - UIPI prevents SetWindowsHookEx to higher integrity level
  - Can't get correct permissions for CreateRemoteThread on protected processes

There are some interesting pros and cons of using different DLL injection methods.

First, the SetWindowsHookEx() function call, only works with graphical applications. Although, all of the graphical apps can be injected with a single call to SetWindowsHookEx(), by setting the target thread ID parameter to 0.

The CreateRemoteThread() technique will work for any application, not just graphical ones. One restriction is that it only works on NT based system, not Windows 95/98/ME.

With the introduction of Windows Vista, there are some changes that stop many classic DLL injection attacks. First, user processes and high-privileged system processes run at different sessions, preventing messages from being exchanged. In addition, a feature called User Integrity Privilege Isolation was implemented. This prevented a lower integrity process from calling SetWindowsHookEx() on a process with a higher integrity level.

Also, an unprivileged process can't get the correct permissions needed to call the CreateRemoteThread() function on a protected process.
While Windows 7 introduces several new security enhancements, DLL (code) injection is still possible.

The AppInit_DLLs registry key, which automatically loads specified DLLs still exists. There is a new (associated) registry key that can specify to auto-load only signed DLLs. This key is disabled by default in Windows 7, but enabled in Server 2008.

The largest changes to DLL injection on Windows 7 come from User Interface Privilege Isolation (UIPI). UIPI focuses on preventing processes at lower integrity levels (ILs) from writing to processes at higher integrity levels.

This means that processes can still inject DLLs (code) into other processes at the same (or lower) ILs. For example, Explorer.exe runs in a medium IL, which is the same as a non-privileged / non-administrative session. Consequently most programs a user runs can inject code into the Explorer.exe process. It is also possible to elevate ILs by instantiating certain COM objects that "auto-elevate".

Finally, there are the traditional methods for getting a user to elevate a program's privileges, for example social engineering. Malicious code could be placed in a Windows installer program, creating a trojan horse scenario.
Reflective DLL Injection

- Libraries loaded with LoadLibrary are visible in module list
- Reflective DLL injection doesn't use LoadLibrary
  - Includes its own (minimal) loader
  - Loaded libraries no longer visible
- Detection is more difficult
  - Look at virtual memory ranges for suspicious characteristics

Traditional DLL injection techniques rely on calling the LoadLibrary() function at some point in time. A drawback of using LoadLibrary() is that the loaded DLLs are visible in the module list. An alternative technique is to write custom code that manually loads the DLL. This general approach has existed under different names, such as "manual mapping" and most recently "reflective DLL injection".

Reflective DLL injection is now in use in the Metasploit framework, notably when using the VNC server and Meterpreter payloads. Essentially, a custom DLL loader is included that manually loads a DLL, directly from RAM, into the address space of the running program. The advantage of this approach (for the attacker) is that the loaded libraries are no longer visible.

As a result of the manual loading process, detecting DLL injection is much more difficult to detect. One approach to detecting DLLs loaded with reflective DLL injection is to examine virtual memory ranges (described by virtual address descriptors, VADS) for suspicious characteristics.
Exercise 6: Examine DLL Injection

- Analyze DLL injection
  Goal: Become familiar with analyzing DLL injection
- Examine kInject
  - Injects an arbitrary DLL into an arbitrary process
  - kinject.exe
- kNTIillusion.dll
  - Part of the NTIillusion rootkit (user mode)
  - Hides network connections, reg keys, processes, etc.

For this exercise, you will analyze a tool that performs DLL injection. The goal is for you to become comfortable with the steps involved with this type of DLL injection, as well as to become comfortable analyzing code that performs DLL injection.

The specimen you will examine is called kInject. kInject allows you to inject an arbitrary DLL into an arbitrary process. The name of the specimen is kinject.exe. It, and the associated files, are located in the kinject.zip archive on the DVD you received for this course.

The DLL you will be injecting (kNTIillusion.dll) is part of the NTIillusion rootkit. The NTIillusion rootkit is a user mode rootkit, which performs many traditional rootkit functions, such as hiding network connections, registry keys, processes, etc. For details about the NTIillusion rootkit, take a look at the following Phrack article about it: http://www.phrack.org/issues.html?issue=62&id=12.
Questions
Examine DLL Injection

- Identify the addresses of the following activities:
  - Memory allocated in the victim process
  - The dll name written to the victim process
  - Address for LoadLibraryA is determined
  - The DLL gets loaded into the victim process
- Don’t worry about victim process being opened
  - We'll have kinject start a new copy of notepad
  - kinject gets a handle to new process at creation
- Verify DLL injection with listdlls
  - listdlls notepad.exe
  - Look for ntilusion.dll file
  - Must be run from within a command shell

In the specimen (kinject.exe), identify the addresses of the following activities:
- When memory is allocated in the victim process
- When the name of the DLL is written to the victim process
- When the address for LoadLibraryA is determined
- When the DLL gets loaded into the victim process

Don’t worry about identifying when the victim process is being opened. We’ll use kinject to start a new copy of notepad.exe. Kinject will be able to get a handle to the process when it creates the new notepad process.

You can verify that the DLL injection actually occurred using the listdlls tool. To run listdlls, you need to run it from the cmd.exe you started earlier. Type:

listdlls notepad.exe

Look for the ntilusion.dll file.
Setup (1)
Examine DLL Injection

- Make the directory c:\malware
- Copy 3 files to c:\malware
  - kInject.exe, kNTIllusion.dll, listdlls.exe
- Start up a copy of cmd.exe and cd into \malware
  - cd \malware
- Load kInject.exe into OllyDbg
  - File > Open > Navigate to c:\malware
  - Click on kinject.exe
  - Need to provide command line options
  - notepad.exe c:\malware\kntillusion.dll --create

To examine this specimen, first create the directory c:\malware. If this directory already exists on your system, that’s ok. Next copy the files kinject.exe, kNTIllusion.dll and listdlls.exe to the c:\malware directory. After you have copied the files, start up a copy of cmd.exe and change into the c:\malware directory.

To load kinject.exe into OllyDbg, start OllyDbg and go to File > Open and then navigate to the c:\malware directory. Click on the file kinject.exe. In order to run correctly, you will need to provide kinject.exe with the appropriate command line arguments. The options are:

notepad.exe c:\malware\kntillusion.dll --create

The first argument (notepad.exe) is the name of the process to inject the DLL into. The second argument (c:\malware\kntillusion.dll) is the name of the DLL to inject. The third argument (--create) tells kinject to automatically start (create) the first argument (notepad.exe).
This is a screen shot of the OllyDbg window to open the file. Notice that under the "File name" box the executable file to debug (c:\malware\kinject.exe) is entered.

In the "Arguments" box, the arguments (notepad.exe c:\malware\kntillusion.dll --create) are entered. Note that before the "create" argument, you need to supply two dashes, not a single dash.
One way to locate DLL injection functionality within the malicious executable is to start with the Names window in OllyDbg. You can bring it up by pressing "Ctrl+N" after loading kinject into the debugger. This will show you the names of the external API calls that the program makes. Examine the list to identify the API calls associated with the DLL injection functionality. (See the reminder of these calls on the next slide.)

Once you identify a promising-looking call (such as VirtualAllocEx) in the Names window, highlight it by clicking on it once, then press Enter (do not double-click). Once you press Enter, the References window will come up, showing you the references to that API call. Click on each reference, then press Enter, and OllyDbg will take you to the area of the code that references that instance of the API call. This will allow you to examine the context within which the call is made, so you can determine whether it corresponds to the DLL injection characteristic you had in mind.
<table>
<thead>
<tr>
<th>Hints (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examine DLL Injection</td>
</tr>
<tr>
<td>• Set break points on system calls used for injection</td>
</tr>
<tr>
<td>- OllyDbg: F2 to set a break point</td>
</tr>
<tr>
<td>• Look for the system calls</td>
</tr>
<tr>
<td>- VirtualAllocEx</td>
</tr>
<tr>
<td>- WriteProcessMemory</td>
</tr>
<tr>
<td>- GetModuleHandleA, GetProcAddress</td>
</tr>
<tr>
<td>- CreateRemoteThread</td>
</tr>
</tbody>
</table>

Here are some additional hints, in case you get stuck.

Set break points on the system calls used for DLL injection. In OllyDbg press F2 to set a break point.

Look for the system calls that are associated with DLL injection. The system calls are:

• OpenProcess
• VirtualAllocEx
• WriteProcessMemory
• GetModuleHandleA and GetProcAddress
• CreateRemoteThread
The victim process is opened by calling `OpenProcess` at address `0x401A95`. Memory is allocated in the victim process by calling `VirtualAllocEx` at address `0x4014C1`. The name of the DLL to inject is written to the victim process by calling `WriteProcessMemory` at address `0x4014F1`.
The address for LoadLibraryA is determined by first calling GetModuleHandleA at address 0x401F9. The result of GetModuleHandleA is then passed as input (along with the ASCII string "LoadLibraryA") to GetProcAddress on line 0x401504.

Finally, the DLL is loaded into the victim process by calling CreateRemoteThread at address 0x40153E.
Answers (3)
Examine DLL Injection

- The victim process being opened:
  - 0x401A95 (OpenProcess)
- Memory allocated in the victim process:
  - 0x4014C1 (VirtualAllocEx)
- The dll name written to the victim process:
  - 0x4014E1 (WriteProcessMemory)
- Address for LoadLibraryA is determined:
  - 0x4014F9 (GetModuleHandleA)
  - 0x401504 (GetProcAddress)
- The DLL gets loaded into the victim process:
  - 0x40153E (CreateRemoteThread)

Here are the answers to the exercise:

The victim process is opened at 0x401A95 (OpenProcess). Memory is allocated in the victim process at 0x4014C1 (VirtualAllocEx). The name of the DLL is written to the victim process at 0x4014E1 (WriteProcessMemory).

The address for LoadLibraryA is determined in two locations. First at 0x4014F9 (GetModuleHandleA) and then at 0x401504 (GetProcAddress).

Finally, the DLL is loaded into the victim process at 0x40153E (CreateRemoteThread).
This slide shows how to use listdlls to verify that the dll kntillusion.dll was successfully injected into a running copy of notepad.exe.

To run listdlls, in your running copy of cmd.exe, make sure you have switched into the c:\malware directory. Then run the following command:

```
listdlls -d kntillusion.dll
```

The arguments "-d kntillusion.dll" directs listdlls to only list processes that have the dll kntillusion.dll loaded.

As you can see from the slide, the process notepad.exe has the dll file kntillusion.dll loaded.
API Hooking

- The rootkit needs to intercept the API calls
  - System calls wrapped in library code
- Many methods
  - We'll examine a common method
  - Modifying the first few bytes of the library function

Recall that the rootkit still needs to intercept the API calls, which are really system calls wrapped in library code.

There are many methods to do this, and we'll examine a common method where the first few bytes of the library function are modified.
Here is a quick refresher. We've already gotten the rootkit code injected into the victim process with DLL injection.

The next step is to overwrite the first few bytes of the victim function. This will allow step 2 (redirection to the rootkit code) to take place.
API Hooking Steps

- Locate address of function to hook
  - GetProcAddress()
- Set memory permissions to read/write
  - VirtualProtect()
- Save first few bytes of victim
  - Manual copy, ReadProcessMemory()
- Compute new instructions
  - Typically a jump
- Overwrite first few bytes of victim function
  - Manual copy, WriteProcessMemory()
- Restore memory permissions to original value
  - VirtualProtect()

Here are the steps to go through to patch the first few bytes of a victim function.

First the address of the victim function has to be determined. A common function to do this is the GetProcAddress call. Then the permissions for the memory segment that contains the code need to be set to read and write (and execute). This is commonly done with the VirtualProtect function.

After the permissions have been changed, the first few bytes of the victim function need to be saved (so they can be restored later). This can be done by manually copying the bytes, or by calling the ReadProcessMemory function. The new instructions (that will replace the first few in the victim function) must be computed. Typically, this is a jump into the rootkit code.

Once the new instruction has been computed, it needs to be written to the victim function. This can be done with a manual copy, or via the WriteProcessMemory function. Finally the memory permissions for the code segment the victim function is in can be restored. Again, this is typically done by calling the VirtualProtect function.
Find Address, Set Permissions
(Vanquish)

mov     eax, [ebp+lpProcName]
push    eax ; lpProcName
mov     ecx, [ebp+hModule]
push    ecx ; hModule

Find address of victim function

call    ds:GetProcAddress

mov     [ebp+flNewProtect], eax
mov     eax, [ebp+flNewProtect]
lea     ecx, [ebp+fOldProtect]
push    ecx ; lpfOldProtect
mov     edx, [ebp+flNewProtect]
push    edx ; flNewProtect
push     5 ; dwSize
mov     eax, [ebp+lpBaseAddress]
push    eax ; lpAddress

PAGE_EXECUTE_READWRITE

Set memory permissions

call    ds:VirtualProtect

This is an example of API hooking from the Vanquish rootkit.

The upper screen capture shows the call to GetProcAddress. This is done to find the address of the victim function.

The next screen capture shows the call to VirtualProtect to change the memory permissions. The value 0x40 is the numeric constant for PAGE_EXECUTE_READWRITE.
Copy Bytes from Victim Function, Compute Instruction (Vanquish)

```assembly
push 5
mov eax, [ebp+lpBaseAddress]
push eax
mov ecx, [ebp+arg_8]
push ecx
call sub_1AE1E80  ; This does the copy

mov [ebp+var_8], edx
mov eax, [ebp+arg_C]
mov byte ptr [eax], 0C9h  ; First byte of a relative jump
mov ecx, [ebp+var_8]
and ecx, 0FFh
```

The next screen capture show the first 5 bytes being copied from the victim function. The function sub_1AE1E80 is what actually performs the copy.

Then the new instruction is computed. The byte 0xE9 is the opcode for a relative jump (jump to a location, relative to the current instruction).
Write Instruction, Reset Perms (Vanquish)

```assembly
push 5
mov ecx, [ebp+arg_C]
push ecx
mov edx, [ebp+lpBaseAddress]
push edx
call sub_1AE1EBB ; This overwrites bytes

lea ecx, [ebp+flNewProtect]
push ecx ; lpf101dProtect
mov edx, [ebp+lpf101dProtect]
push edx ; flNewProtect
push 5 ; dwSize
mov eax, [ebp+lpBaseAddress]
push eax ; lpAddress
call ds:VirtualProtect ← Reset memory permissions
```

Then the computed instruction is copied over the first five bytes of the victim function. Notice that the same function that copied the bytes from the victim function is also being used to copy bytes to the victim function. It's actually a generic memory copy style function. Also notice that location of [ebp+lpBaseAddress] is reversed, compared to the previous slide.

Finally, the memory permissions are reset with another call to VirtualProtect. The victim function has now been patched.
Hooked Function Steps

- Restore original bytes
  - Set permissions, overwrite bytes w/original, restore permissions
- Execute original function
- Re-hook victim function
  - Set permissions, overwrite bytes w/jump, restore permissions

The rootkit code that is jumped to by the victim function must also take certain steps. In this case it must first restore the original victim function bytes, using a very similar procedure as before. In this case, the original bytes are copied over, and a new instruction does not need to be computed.

Then the original function needs to be executed, and then re-hooked, using the same procedure as before.
Hooked Function (Vanquish)

```
push offset ?UFindNextFileW@WGYHPAXPAU_WIN32_DL
mov eax, dword_1AE9218
push eax
; lpBaseAddress
call sub_1AE2251
; Restore the victim bytes

call dword_1AE9218
; Address of library function
mov [ebp+var_4], eax

push offset ?UFindNextFileW@WGYHPAXPAU_WIN32_DL
mov ecx, dword_1AE9218
push ecx
; lpBaseAddress
call sub_1AE20A1
; This re-hooks the API
```

Here is an example of the first parts of the rootkit function to filter the output from the FindNextFile function.

The first screen capture shows the call to a function, sub_1AE2251, which restores the victim bytes. The next screen capture shows the call to the "healed" victim function.

Finally, the call to sub_1AE20A1 re-hooks the victim API function. This is the function that the previous screen shots (to show the hooking of the victim API function) were taken from.
Kernel Mode Rootkits

We will now examine some techniques used by kernel mode rootkits.
Direct Kernel Object Manipulation

- Many rootkit techniques focus on filtering data returned by a function (or system) call
  - Instead modify the data that the kernel uses
- More difficult to detect
  - Don't need to patch/hook userspace
  - Can unload rootkit code after modifying objects
- There are some drawbacks
  - Can't perform all rootkit functionality
  - Very delicate
  - Structure layouts change per OS version
  - Paradox: it's gotta be there somewhere...

Many rootkit techniques focus on filtering data returned by a user space or kernel (system) function call. Regardless of where the filtering takes place, the kernel has to examine lists of objects (e.g. running processes, threads, etc.). Instead of modifying the returned data, direct kernel object manipulation (DKOM) focuses on modifying the kernel objects in memory.

Since the kernel objects are manipulated, it is much more difficult to detect DKOM-based rootkits. Once the objects have been modified, the rootkit code can be unloaded from memory, leaving less of a footprint.

Despite the reasons why an attacker may want to use DKOM-based tactics, there are some things to consider. First, DKOM techniques are only usable for objects that the kernel keeps lists of. For instance, the kernel keeps track of all running processes, but not all files on the file system. As a result, DKOM techniques can't be used for file hiding.

Also, DKOM techniques are very delicate, with the layouts for various data structures changing between each OS version. Overwriting one byte too many can lead to a blue screen of death (BSOD).

Finally, even though DKOM techniques are effective at hiding processes there exists a "rootkit paradox" (originally described by Jesse Kornblum at http://jessekornblum.com/publications/jde06.html). The paradox is that even though a rootkit wants to remain hidden, in order for the rootkit code to execute, it has to be scheduled by the kernel, and is consequently kept track of by the kernel.
DKOM: Hiding Processes

- Windows kernel keeps track of running processes and threads
  - EPROCESS and ETHREAD structures
- Each entry points to (both) previous and next entries
  - Doubly linked list
- To hide a process adjust surrounding entries to skip over an EPROCESS structure
  - Have to do this in kernel space (device drivers)
  - Unlinked processes still run because threads are scheduled

The Windows kernel keeps track of running processes and threads with lists of structures called EPROCESS and ETHREAD respectively. The list of EPROCESS structures is a doubly linked list. Each entry in the EPROCESS list points to both the entry before and the entry after in the list.

When Windows enumerates a list of running processes, it walks down the list of EPROCESS structures. To hide a running process a rootkit needs to remove (unlink) the corresponding EPROCESS structure from the EPROCESS list. This can be done by adjusting the pointers in the entries before and after the target EPROCESS structure.

Since all of these structures are maintained by the kernel, DKOM-based techniques have to be executed by code located in kernel space. This is most commonly done by writing the rootkit code into a malicious device driver.

You may be wondering how a process that is unlinked from the EPROCESS list could still run, since it is no longer in the kernel list. The answer lies in the fact that the Windows kernel schedules threads to run, as opposed to individual processes.
This is an example of a list of EPROCESS structures. On the slide above, there are 3 EPROCESS structures labeled A, B, and C. Recall that each EPROCESS structure points to the structures before and after it.

The Flink field of structure A points to structure B. The Flink field of structure B points to structure C. This forms one half of the doubly-linked list.

The Blink field of structure C points to structure B. The Blink field of structure B points to structure A. This forms the second half of the doubly-linked list.
This slide demonstrates how to unlink the EPROCESS structure labeled B.

The desired result is that the doubly linked list exclude the B structure. To do this, the Flink field of structure A is set to the value of the Flink field of structure B (meaning A’s Flink field now points to C). The Blink field of structure C is set to the value of the Blink field of structure B (meaning B’s Blink field now points to A).

The Flink and Blink fields of structure B also have to be updated, to point back to structure B. Doing this will help prevent random crashes when the program that structure B represents exits.

After the links have been updated, the doubly linked EPROCESS list goes from structure A to structure C, bypassing structure B. If Task Manager (or other various process-listing programs) enumerated the list of running processes, the program associated with structure B would not be shown.
Steps to Unlink

• Locate EPROCESS list
• Find target EPROCESS
  – Search by name or PID
• Update links
  – Both neighbors and target

There are three high level steps to unlink (remove) an EPROCESS structure from the doubly-linked list maintained by the kernel.

First, the EPROCESS list has to be located. Second, the target EPROCESS structure has to be identified from within the list. The identification is usually by name or process ID (PID). Finally, the links of the target and neighbor EPROCESS structures have to be updated.
The first high level step is to locate the EPROCESS list. One common method for doing this is to call the IoGetCurrentProcess() method. This method returns a pointer to the current EPROCESS structure.

Internally, IoGetCurrentProcess() starts by looking at the CurrentThread field of the Kernel Processor Control Block (KPRCB). This is a structure that the Windows kernel uses to keep track of scheduling information, including the current thread of execution.

The CurrentThread field of the KPRCB points to an ETHREAD structure, which describes the current thread of execution. The ETHREAD structure is a structure used by the kernel to keep track of information about a thread of execution. It is analogous to a process's EPROCESS block (but for threads). The ApcState field of the ETHREAD structure points to the EPROCESS structure of the program that the thread belongs to.

Once the EPROCESS structure has been located, the EPROCESS list can be enumerated using the Flink and Blink fields.
Find Target EPROCESS structure

- Search by PID
  - Not persistent across reboot/restart
  - UniqueProcessId
- Search by name
  - 16 character name
  - ImageFileName
- Offset for field varies by OS

Once the EPROCESS list has been located, the target EPROCESS structure to unlink (remove from the list) has to be located. Generally, there are two ways to identify the target EPROCESS structure, by process ID (PID) or by name. Regardless of how the target EPROCESS structure is located, the offsets of the fields vary by operating system version, and are usually located in a dynamic fashion.

One disadvantage of locating an EPROCESS structure by PID is that PIDs are not persistent, they change across reboots, or even when a program is restarted. The field that describes the PID is the UniqueProcessId field. Since the offset (distance from the start of the EPROCESS structure) for the UniqueProcessId field varies per operating system, it is not uncommon for malicious code to have several different offsets hard coded. At run-time, the malicious code determines the version of the operating system and selects the appropriate offset.

Another mechanism to identify the target EPROCESS structure is by name. A disadvantage of this technique is that the name is only 16 characters long, and may not account for the path that a particular executable is run from. The field that describes the name of the program associated with the EPROCESS structure is ImageFileName. Since the offset for the ImageFileName field varies per operating system, malicious code has to determine the offset at run-time. A technique that has worked fairly consistently in the past, is for the rootkit to determine this offset when it is first loaded into memory.

Since most kernel mode rootkits operate as Windows device drivers (so they can have the appropriate access levels) they are loaded by the Service Control Manager. The name for the Service Control Manager is "System". When a kernel mode rootkit starts up, it can locate its own EPROCESS structure (using the technique described on the previous slide) and then look for the string "System". (Incidentally, this is a technique that has been used in several Sysinternals tools).
The last step to unlinking (removing) and EPROCESS structure from the doubly-linked list is to update the various Flink and Blink fields.

To remove the EPROCESS structure, the neighboring Flink and Blink fields have to be updated to the values that the target EPROCESS structure (the one to hide) points to. That is, the structure before the target has to point to the structure after the target. Similarly, the structure after the target has to point to the structure before the target.

The target EPROCESS structure is then updated so it’s own Flink and Blink fields point to itself. This is done so to help prevent random blue-screen-of-deaths (BSOD) when the hidden process terminates.
This is an example of DKOM from the FU rootkit.

This slide shows two high level steps, locating the EPROCESS list, and finding the target EPROCESS structure by PID.

First IoGetCurrentProcess() is called to get a pointer to the current EPROCESS structure. Then ECX is set to the offset of the UniqueProcessId (PID) field. The value of the PID field for the current EPROCESS structure is moved into EDX and then compared against the target PID. If a match is found, the function exits. Otherwise, EDX is set to the offset of the Flink field.

The middle portion is a check to see if the entire list has been enumerated, and the target PID has not been found. If this is the case, the function exits (with an error). This is to prevent entering an infinite loop.

The last portion retrieves the next EPROCESS structure. This is done by following the Flink field of the current EPROCESS structure. The Flink field points to the Flink field of the next EPROCESS structure (not the start of the next EPROCESS structure). To accommodate for this, the offset of the Flink field (in the EDX register) is subtracted.

After the start of the next EPROCESS structure is located, the new PID is extracted and checked.
This slide shows the process of updating neighbor links for the EPROCESS structures.

When entering this portion of code, the EAX register points to the target EPROCESS structure to hide. ECX is set to the offset of the Flink field, and then EAX is updated to point to the target EPROCESS structure’s Flink field.

ECX is then set to the previous EPROCESS structure, and EDX is set to the next EPROCESS structure. Then the previous entry’s Flink is set to point to the entry after the target structure.

ECX is then set to point to the next entry, and EAX points to the previous entry. The Blink field of the entry after the target structure is set to the entry before the target structure.

The FU rootkit does not set the Flink and Blink pointers of target structure to itself.
Next, we'll look at the way keyloggers work.
We'll now examine how keyloggers work.
Keystroke Loggers

- Users type many interesting things
  - Credit card info, passwords, SSNs, etc.
- Monitor user key strokes
- Many bots and worms do this
  - Send the logs to an attacker
- Two most common methods
  - Install hook for keyboard events
  - Poll keyboard state with GetAsyncKeyState()

Users type many things that are of interest to the attackers. Credit card information, social security numbers, passwords, banking information, etc. all have potential value to the criminal underground.

Since this information is typed at the keyboard, different malware specimens will monitor a user's keystrokes. Many bots and worms have this type of functionality built into them, and they send the logs of user key strokes to the attacker, sometimes through e-mail, sometimes through a web request, or even through proprietary protocols.

There are two commonly used methods to log keystrokes. The first method is to install a hook for keyboard related events. The second method is to poll the state of each key on the keyboard using the GetAsyncKeyState function.
Keystroke Logging with Hooks
(Trojan-Spy.Win32.Keylogger.be)

```
SetKeyHook proc near
    push    0 ; dwThreadId
    mov     eax, ds:Module
    push    eax
    mov     eax, offset fn
    push    eax
    call    WH_KEYBOARD ; idHook
    mov     ds:hhk, eax
    push    0 ; dwThreadId
    mov     eax, ds:Module
    push    eax
    mov     eax, offset sub_mhcb
    push    eax
    call    SetWindowsHookExA ; idHook
    mov     ds:dword_4054E0, eax
    ret
SetKeyHook endp
```

Here is an example of a keystroke logger using the SetWindowsHookExA() method of dll injection. This is from the wdll.dll component of the Trojan-Spy.Win32.Keylogger.be tool.

Conveniently, the function that performs the hooking was exported with the name "SetKeyHook".

You can see this function sets up to intercepts keyboard messages, by calling SetWindowsHookExA() with the WH_KEYBOARD parameter. This function also sets up to intercept mouse messages by calling SetWindowsHookExA() with the WH_MOUSE parameter. Notice the similarity between this function, and the DLL injection technique which also used hooks. In this case, the WH_KEYBOARD and WH_MOUSE messages were intercepted. The previous example of using hooks intercepted the WH_CBT message.
Keystrokes with GetAsyncKeyState

KeyLogger
Every 50 ms:
for(x=1;x<256;x++) {
GetAsyncKeyState(x)
}

User Program

Note: Steps 2 and 3 are repeated every 50 ms

This slide shows how a keystroke logger that uses the GetAsyncKeyState function works.

The first thing that happens (Step 1) is that the user starts typing on the keyboard.

The rootkit code is in a loop. The code periodically (e.g. every 50 milliseconds) polls the state of each key on the keyboard by calling GetAsyncKeyState. This loop (call GetAsyncKeyState for each key) is repeated every 50 milliseconds (or whatever the malware author decides.)
Key Logging with GetAsyncKeyState (Spybot)

```assembly
push    10h
    ; uVirtualKey ← Check if shift is depressed
    call   GetKeyState
    mov    edi, eax
    movsx  edi, di
    mov    [ebp+var_8], edi
    mov    edi, [ebp+var_4]
    mov    ebx, uKey[edi=4]
    push   ebx
    call   GetAsyncKeyState ← Check to see if a key is depressed
    mov    edi, eax
    movsx  edi, di
    inc    [ebp+var_4]
    cmp    [ebp+var_4], 5Ch ← Only looks for 92 (0x5C) keys
    jle    loc_4B32B9
```

This example comes from the Spybot bot software.

The first screen capture shows GetKeyState being called to check if the shift key is depressed. Then the GetAsyncKeyState is called in a loop, examining the status of each key. Since GetKeyState and GetAsyncKeyState return the state of a keyboard key (and not an ASCII character code), the keystroke logger checks to see if the shift key was also depressed. This way it is possible to determine the difference between an upper and lower case letters, as well as the numbers 1234567890 and the symbols !@#$%^&*().

The second screen capture shows the loop that is wrapped around the GetAsyncKeyState function call. Examining the loop reveals that this specimen is only interested in 92 (0x5C) key codes. The [ebp+var_4] variable is used as an index into an array of key codes to monitor.
Exercise 7: Keylogger

- Examine keylogger
  - Goal: Become familiar with analyzing keylogger code
- Keylogger built into SpyBot
  - spybot.exe

For this exercise you will examine a keylogger. The goal is for you to become analyzing code that monitors user keystrokes.

The specimen you will be analyzing is a bot called SpyBot. The name of the file is spybot.exe. You will find it in the spybot.zip archive on the DVD you received for this course.
Questions
Keylogger

- Identify addresses where keys are examined
- Extra credit:
  - Identify addresses of the polling loop
  - Determine how often keys are polled

For this exercise, you should identify the addresses of the functions where the keys are examined.

For extra credit, identify the addresses of the loop that performs the polling, and how often the keys are polled.
Hints
Keylogger

- Look for system calls
  - Inside function at 0x4030E0
- Extra credit
  - Remember loops jump backwards
  - Look for a system call that suggests some type of pause
  - It takes milliseconds as the argument

Here are some hints, in case you get stuck.

To find the addresses of the instructions that examine the keys, look for the system calls mentioned before. As a further hint, the instructions are somewhere inside the function located at 0x4030E0.

To help find the polling loop, remember that loops have to jump to earlier instructions. To find out how often the keys are polled, look for a system call that suggests some type of pause. As a further hint, the function takes milliseconds as the argument.
The state of the keys are examined in four locations. The GetKeyState function is called at addresses 0x4032DB, 0x403306 and 0x403332. The GetAsyncKeyState function is called at address 0x4032F3.
Here you can see the start and end of the polling loop. The polling loop starts at address 0x40320D, and calls the Sleep function with a parameter of 8. This means the loop sleeps for 8 milliseconds.

The end of the polling loop compares the variable at address [EBP-0x9CC] with the value 0. If the two are equal, the loop restarts, as shown by the conditional jump at address 0x40358D.
Here are the answers to the exercise.

The keys are examined in four locations. The GetKeyState function is called at 0x4032DB, 0x403306, and 0x403332. The GetAsyncKeyState function is called at 0x4032F3.

The polling loop starts at 0x40320D and ends at the conditional jump at 0x40358D. There is a call to Sleep() at the top of the loop. The loop sleeps for 8 milliseconds before each iteration.
Our next section examines several malware characteristics, including network sniffing, packet spoofing, and file downloading.
Sniffers

We will now examine techniques used by malicious software that monitor network connections.
Monitoring the Network

- Monitor network traffic
  - Most common is to put interface into promiscuous mode
  - Used by admins and attackers alike
- Other mechanisms
  - Intercepting network related calls
  - Intercept higher level functions (InternetUrlOpen, etc.)
- Installing BHOs

Sniffers are tools that monitor network traffic. Sniffers have been around for many years and are useful to systems administrators as well as attackers.

The traditional sniffer places the interface into promiscuous mode. By doing this, the network card will then pass all packets, even ones addressed to other Mac addresses, to the operating system.

There are other mechanisms to monitor network traffic, including performing API hooking on network related calls such as send and recv. In addition, malicious code can also hook higher level functions such as InternetUrlOpen, etc.

Yet another possibility is to install a browser helper object (BHO) for the users browser, and monitor traffic (web based).
Steps to Sniff

- Create a raw socket
  - WSASocket() or socket()
- Bind socket to an interface
  - bind()
- Put interface into promiscuous mode
  - WSAIoctl() or ioctlsocket()

Sniffing on an interface with promiscuous mode is a fairly straightforward process.

First, a raw socket needs to be created. This is commonly done with calls to the WSASocket or socket functions. After the raw socket has been created, it must be bound to a specific interface to listen on. This is commonly done with the bind function call. Finally the interface must be put into promiscuous mode. Enabling promiscuous mode is typically performed through calls to the WSAIoctl or ioctlsocket functions.
Create socket and bind (RKSniffer)

```
push 1                        ; dwFlags
push eax                       ; g
push eax                       ; IpProtoColInfo
push eax                       ; protocol
push SOCK RAW                  ; type
push AF_INET                   ; af
call ds:WSASocketA             ← Create a raw socket

lea eax, [ebp+name]
push 10h                      ; name_len
push eax                       ; name
push dword ptr [esi+4] ; 5
call ds:bind                   ← Bind to an interface
```

This example is from the RKSniffer tool. The first screen shot shows the creation of a raw socket via the WSASocketA function call. Note that the type is SOCK_RAW.

The second screen shot shows a call to the bind function, binding the socket to a specific interface.
The last step is to put the interface into promiscuous mode.

In this example, the function call WSAIoctl is provided with the SIO_RECVALL parameter, telling the operating system to put the network card into promiscuous mode.

The SIO_RECVALL is a numeric constant, that IDA Pro was able to replace with the equivalent text name. In OllyDbg you may see a numeric value instead of the name SIO_RECVALL.
Exercise 8: RKSniff

- Examine a sniffer
  - Goal: Become comfortable analyzing code to put interface into promiscuous mode
- RKSniff by RedKod Team
  - rksniffer.exe
- Identify addresses of the following activities:
  - Raw socket being created
  - Socket being bound
  - Interface put into promiscuous mode
- Hint: Look for system calls (3 slides ago)

For this exercise, you will examine a sniffer. The goal is for you to become comfortable analyzing code that puts network interfaces into promiscuous mode.

The specimen you will analyze is RKSniff by RedKod Team. The name of the file is rksniffer.exe. You will find it in the rksniff.zip archive on the DVD you received for this course.

For this exercise, identify the addresses of the following activities:
- When the raw socket is created.
- When the socket is bound to an interface.
- When the interface is put into promiscuous mode.

If you get stuck, look for the system calls (they are three slides prior).
The raw socket is created by calling WSASocketA at address 0x401194. The raw socket is then bound to an interface by calling the bind function at address 0x4012C6.
The interface is put into promiscuous mode by calling the WSAIoctl function at address 0x4012F8.
Here are the answers to the exercise:

The raw socket is created at 0x401194 (WSASocket). The socket is bound to an interface at 0x4012C6 (bind).

Finally, the interface is put into promiscuous mode at 0x4012F8 (WSAIoctl).
Packet Spoofing

We'll now examine techniques used to send spoofed packets onto the network.
Send Spoofed Packets

- Spoofing network traffic can be useful
  - E.g. DOS tools, ARP cache poisoning, etc.
- On Windows, using winpcap is common
  - Build packet in code, and send it out
  - Raw sockets were disabled in XP SP2
- Steps
  - Open adapter
  - Allocate memory for packet structure
  - Make raw data with bogus information
  - Initialize packet with raw data
  - Send raw packet to the network

The ability to send spoofed network traffic can be immensely useful for an attacker. It is a commonly used technique in denial of service tools, to help make tracing and stopping the attack much more difficult. ARP cache poisoning, which is commonly used to sniff network traffic on switched networks also requires sending out spoofed network traffic.

The most commonly used method in malicious code to send spoofed network traffic is to use the winpcap libraries. Essentially, the malicious code builds up a packet in memory, and sends the packet onto the wire. The use of raw sockets was also very popular until Windows XP SP2 disabled the use of raw sockets. It is still possible to send raw network traffic in Windows, just not using raw sockets. Raw sockets are the most common technique that Linux based malicious code uses to send spoofed network traffic.

The steps to send a spoofed network packet are fairly straightforward. First the network adapter has to be opened, and then memory for a pcap specific packet structure is allocated. Raw network data is then created with bogus information (E.g. false source or destination IP addresses.) Then the packet structure is initialized with the raw data, and is sent out onto the network.
Open Adapter, Allocate Memory
(nmap)

<table>
<thead>
<tr>
<th>call</th>
<th>PacketOpenAdapter ← Open the network adapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>esp, 4</td>
</tr>
<tr>
<td>test</td>
<td>eax, eax</td>
</tr>
<tr>
<td>mov</td>
<td>[esi], eax</td>
</tr>
<tr>
<td>jz</td>
<td>short loc_4B262C</td>
</tr>
<tr>
<td>cmp</td>
<td>dword ptr [eax], 0FF</td>
</tr>
<tr>
<td>jz</td>
<td>short loc_4B262C</td>
</tr>
<tr>
<td>push</td>
<td>7D 0000h</td>
</tr>
<tr>
<td>push</td>
<td>eax</td>
</tr>
<tr>
<td>call</td>
<td>PacketSetBuff</td>
</tr>
<tr>
<td>add</td>
<td>esp, 8</td>
</tr>
<tr>
<td>call</td>
<td>PacketAllocatePacket ← Allocate memory for the packet structure</td>
</tr>
</tbody>
</table>

This example comes from the Windows version of the wildly popular tool nmap. The tool was called with the command line options: -c eth0 -S 1.2.3.4 172.16.26.1

These command line options directed the tool to perform a port scan against the IP address 172.16.26.1, with a source IP address of 1.2.3.4.

The first step was for the tool to open the adapter by calling PacketOpenAdapter(). Then memory for the packet structure was allocated using the PacketAllocatePacket() function.
Make Raw Data, Initialize, Send
(nmap)

```
MOV [esi+8Ch], ecx
MOV edx, [ebp+0]
MOV [esi+10h], edx
JZ short loc 47BE:
MOV eax, dword ptr [esi]
push edx
push eax
```

The packet is then built up in memory. As you can see from the registers windows, the ECX register contains the source address (1.2.3.4) in little endian. The EDX register contains the destination address (172.16.26.1) also in little endian.

After this, the packet structure is initialized by calling PacketInitPacket(). Finally, the raw packet is sent onto the network by calling the PacketSendPacket() function.
We'll now examine a tool that has risen in popularity over the past few years... Downloaders.
Downloader is a piece of code that simply downloads a file to disk, and then executes it. Due to their simple nature, downloaders tend to be very small. This smaller size allows them to fit into more places and makes them more likely to avoid detection.

The protocol used to download files is typically either HTTP or FTP, although there have been downloaders that download over TFTP, and even custom protocols.

Downloaders are so common (and popular) that their functionality is included as a payload in the Metasploit framework.

Another similar type of malware is called a dropper. Instead of downloading a file, a dropper has the dropped file embedded in itself. This helps get by some types of automated analysis.
Downloader Steps

- Download file and save to disk
  - URLDownloadToFile()
- Execute newly downloaded file
  - ShellExecute(), WinExec(), etc.

A downloader has two basic steps.

First the downloader has to retrieve a file. The most common function used for this purpose is URLDownloadToFile.

Once the file has been downloaded, it must be executed. There are a number of ways of doing this, using function calls such as ShellExecute, WinExec, etc.
This specimen is the Trojan-Downloader.Win32.MultiDL.23. The function you see is actually the main part of the program.

First the URLDownloadToFileA function is called with a url and a file name.

Second, the WinExec function is called to execute the downloaded file. Finally the ExitProcess function is called with an argument of 0.
Our next section examines the use of HTTP control channels by malware.
HTTP Control Channels

Internet Relay Chat (IRC) is a common communication protocol for malicious code such as bots. Recently, alternative protocols, such as HTTP, have also been used. We'll now examine how malicious code can use HTTP for command and control.
Alternative Control Channels

- IRC is a popular communication mechanism
  - Not as common in business
  - Fairly easy to detect
- Alternative communication channels
  - HTTP, P2P, etc.
  - Black Energy, Storm, Stration
- We'll examine HTTP

Internet Relay Chat (IRC) is a popular protocol for controlling malicious code, and has been in use for several years. Fortunately, it is easy to detect such traffic, especially in corporate environments, since IRC is seldom used during the course of normal business.

To circumvent this situation, malicious code authors have used alternative protocols for communicating and controlling their malicious code. Protocols such as HTTP, P2P, and even instant messaging can be used to communicate from controller to bot, and vice versa. Bots such as Black Energy, Storm (and its variants), Stration, etc. all use mechanisms other than IRC for communication.

In this section, we'll examine how attackers can use HTTP to communicate with malicious code.
HTTP Control Channels

- Two approaches
  - Regular commands over port 80
    - E.g. reverse shell over port 80
  - Embedded in HTTP traffic
    - Looks like user is browsing a page
    - Exists in tools such as Metasploit framework, Reverse WWW Shell, etc.

- Malicious code polls server at intervals
  - Receive commands, send results
  - Some will work with local proxy configuration
  - We'll examine this type of malicious code

When communicating over HTTP, there are two general approaches. The first is to tunnel regular traffic over port 80. An example of this would be using a reverse shell (e.g. from the Metasploit framework) over port 80. While this is a fairly simple approach, it is likely that this sort of traffic would not pass through a proxy. In addition, certain intrusion detection systems can detect this type of anomalous traffic.

The other mechanism is to embed the malicious communication in HTTP traffic. The network connection looks like a user is browsing a regular web page, since the traffic goes through well formed HTTP requests and responses. This technology has existed for some time, in tools such as the Metasploit framework (PassiveX payload), and Reverse WWW Shell by THC.

The way the second approach works, is that malicious code on a victim system periodically polls a web server, receiving commands, and sending results through the HTTP connection. Some malicious code is designed to read local proxy settings from Microsoft Windows systems.
Here is a graphical representation of the connection process for HTTP controlled malware.

The first thing the malware does, is initiate an HTTP request to a web server, for a specific web page. Typically this will be a web page that interfaces to a custom (attack controlled) web application.

The second step is that the web page requested will return a command to the bot. Finally, the bot processes the command it received from the web page. It should be noted that the first two steps happen over the same connection. Since this connection was initiated by the victim, this approach will generally work through NATs, firewalls, proxies, etc.

This process of request, receive, and process is repeated a regular intervals, determined by the malicious code author.
HTTP Control Steps

- Create HTTP connection
  - InternetOpen(), InternetConnect()
- Build HTTP request
  - HttpOpenRequest(), HttpAddRequestHeaders() (optional)
- Send HTTP request
  - HttpSendRequest
- Read response
  - InternetReadFile

In order for malicious code to communicate over an HTTP connection, there are several steps that must occur.

First, the malicious code must create an HTTP connection. One way of doing this is to call the InternetOpen() function to create a network connection, and the InternetConnect() function to initiate a connection to the remote server. As a side note, the InternetConnect() function doesn't actually create the TCP/IP connection for HTTP connections until an HTTP request is actually sent. Instead, it merely builds the in-memory data structures that contain the connection information.

After an HTTP connection has been created, the malicious code needs to build the HTTP request. This can be done by calling the HttpOpenRequest() function to create an actual request. An optional step is to add custom headers. This can be done by calling the HttpAddRequestHeaders() function.

Once the request has been built, data can be sent to the remote system by calling the HttpSendRequest() function. Windows will create a TCP/IP connection to the remote system, and send an HTTP request. The response from the server (which typically contains commands for the malicious code) can be read by calling the InternetReadFile() function.
Create HTTP Connection
(Black Energy)

```
push 40000000h
push 0
push 0
push 1
push offset aMozilla4
call ds::InternetOpenA ← Create network connection
push 0
push 0
push 3
push arg
push 0
push 50h
lea edx, [ebp+var_110]
push edx
mov eax, [ebp+var_8]
push eax
call ds::InternetConnectA ← Create HTTP session to server
```

Here is an example of the HTTP control mechanism from the Black Energy bot. This bot appears to have been developed by Russian hackers.

We can see the bot creating the HTTP connection by first calling the InternetOpenA() function to create a network connection. Next, the bot creates an HTTP session to the remote web server by calling the InternetConnectA() function. It should be noted that the actual TCP/IP connection isn't created until the HTTP request is sent.
Build and send HTTP Request  
(Black Energy)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push 0</td>
<td></td>
</tr>
<tr>
<td>push 0</td>
<td></td>
</tr>
<tr>
<td>push 0</td>
<td></td>
</tr>
<tr>
<td>push 0</td>
<td></td>
</tr>
<tr>
<td>push 0</td>
<td></td>
</tr>
<tr>
<td>lea ecx, [ebp+var_228]</td>
<td></td>
</tr>
<tr>
<td>push ecx</td>
<td></td>
</tr>
<tr>
<td>push offset aPost</td>
<td></td>
</tr>
<tr>
<td>mov edx, [ebp+var_11C]</td>
<td></td>
</tr>
<tr>
<td>push edx</td>
<td></td>
</tr>
<tr>
<td>call ds:HttpOpenRequestA</td>
<td>Build HTTP request</td>
</tr>
<tr>
<td>push eax</td>
<td></td>
</tr>
<tr>
<td>mov eax, [ebp+var_4]</td>
<td></td>
</tr>
<tr>
<td>push eax</td>
<td></td>
</tr>
<tr>
<td>mov ecx, [ebp+var_C]</td>
<td></td>
</tr>
<tr>
<td>push ecx</td>
<td></td>
</tr>
<tr>
<td>call ds:HttpSendRequestA</td>
<td>Send HTTP request</td>
</tr>
</tbody>
</table>

Here we can see the bot build an HTTP request by calling the HttpOpenRequestA() function. In this specific variant, the bot does not add any additional headers by calling the HttpAddRequestHeadersA() function.

Once the HTTP request has been built, it is sent by calling the HttpSendRequestA() function.
Read Response (Black Energy)

```
push    edx
mov     eax, [ebp+arg_C]
push    eax
mov     ecx, [ebp+arg_B]
push    ecx
mov     edx, [ebp+var_C]
push    edx
call    ds:internetReadFile ← Read server response
```

Finally, the response from the server (which typically contains a command for the bot) is read by calling the InternetReadFile() function.
Exercise 9: Blinky

- Examine an HTTP bot
  - Goal is to become comfortable with http controlled bots
- blinky.exe (Main polling loop at address 0x40157B)
- Identify addresses of the following:
  - Function that is the HTTP control connection
  - Creating an Internet connection
  - Building the HTTP request
  - Sending the HTTP request
  - Reading server responses
- Extra credit: Identify commands supported by the bot

For this exercise, you will examine a bot which uses an HTTP connection for control. The goal is for you to become comfortable analyzing bots that are controlled via HTTP.

The specimen you will analyze is called “blinky.exe”. The main loop that polls the web server starts at address 0x40157B.

For this exercise, identify the addresses of:
- The function that performs the HTTP communication
- The code that creates the Internet connection
- The code that builds the HTTP request
- The code that sends the HTTP request
- The code that reads the server responses

For extra credit, identify the commands that are supported by the bot.
Here are some hints to help you during the exercise.

To find the HTTP control connection, remember that it is called from the main loop (at address 0x40157B). You may also want to look for the system calls described 5 slides ago. When looking for the HTTP control connection, there are two functions that are similar. One function is used by the bot for HTTP control, and the other is used by the “http flood” command. To help differentiate between the two functions, the HTTP control function includes a call to the `HttpAddRequestHeadersA()` function.

To help with answering the extra credit questions, in the main loop, look for strings compared to the data that is returned by the server. In this specific specimen, several common string functions have been inlined. The function at address 0x401C3A is the `strtok()` function. The `strtok()` function splits a string, given a list of character tokens. The function at address 0x401C2F is the `atoi()` function. The `atoi()` function converts an ASCII string representation of a number (e.g. “1234”) to a binary value (e.g. 1234). The function at address 0x401B20 is the `strcmp()` function. Recall that the `strcmp()` function is used to compare two strings together.
Here are the answers to the exercise.

The function at address 0x401400 is the function that is the HTTP control function. You can see this function is called in the main loop (at address 0x40159D). A hint is that the string "/index.php" is pushed onto the stack, right before calling the HTTP control function.

Inside the HTTP control function, the network connection is created by calling the InternetOpenA() function at address 0x40142A. The connection to the remote server is then created by calling the InternetConnectA() function at address 0x401431.
Here you can see the bot building the HTTP request by first calling the \texttt{HttpOpenRequestA()} function at address 0x401476. After this, the bot adds additional HTTP headers by calling the \texttt{HttpAddRequestHeadersA()} function at address 0x4014AC.
The HTTP request is sent, by calling the HttpSendRequestA() function at address 0x4014C1.

Finally, the results (commands from the server) are read by calling the InternetReadFile() function at address 0x4014EA.
Here are the answers for the exercise.

The function at address 0x401400 is used by the bot for HTTP control. Do not confuse this with the function at address 0x40120D, which is used by the bot during the http_flood command.

Inside the HTTP control function, the Internet connection is created by first calling the InternetOpenA() function at address 0x40142A. The connection to the remote server is created by calling the InternetConnectA() function at address 0x401451.

Next, the HTTP request is built by first calling the HttpOpenRequestA() function at address 0x401476. Additional HTTP headers are added by calling the HttpAddRequestHeadersA() function at address 0x4014AC.

The HTTP request is sent by calling the HttpSendRequestA() function at address 0x4014C1.

Finally, the response from the server is read by calling the InternetReadFile() function at address 0x4014EA.
Here we can see some of the commands supported by the bot. To do this, we can look at the strings that are compared to the data sent by the remote server. The address of the start of the data sent by the remote server is stored in the EBX register.

At address 0x4015D3, we see a compare to the string “sysinfo”. Examining the associated function, it turns out that this command instructs the bot to return basic information about the victim system.

At address 0x4015F6, we see a compare to the string “download”. Examining the associated function, it turns out that this command instructs the bot to download a file from a remote website.

At address 0x401626, we see a compare to the string “http_flood”. Examining the associated function, it turns out that this command instructs the bot to launch an HTTP GET flood against a remote web server.

At address 0x40167C, we see a compare to the string “icmp_flood”. Examining the associated function, it turns out that this command instructs the bot to launch an ICMP flood (send a series of ICMP packets) against a remote server.
At address 0x4016AF, we see a compare to the string "execute". Examining the associated function, it turns out this command instructs the bot to execute a file on the victim system.

At address 0x4016D8, we see a compare to the string "wait". Examining the associated function, it turns out this command instructs the bot to pause for a specified period of time.

Finally, at address 0x401707, we see a compare to the string "die". Examining the associated function, it turns out this command instructs the bot to exit.
Here are the answers to the extra credit question.

The bot supports the following commands:

- The "sysinfo" command, which can be seen by a string compare at address 0x4015D3. This command instructs the bot to send information about the victim system to the remote web server.
- The "download" command, which can be seen by a string compare at address 0x4015F6. This command instructs the bot to download a file from a remote web site.
- The "http_flood" command, which can be seen by a string compare at address 0x401626. This command instructs the bot to launch an HTTP flood against a remote server.
- The "icmp_flood" command, which can be seen by a string compare at address 0x40167C. This command instructs the bot to launch an ICMP flood against a remote server.
- The "execute" command, which can be seen by a string compare at address 0x4016AF. This command instructs the bot to execute a file from the local system.
- The "wait" command, which can be seen by a string compare at address 0x4016D8. This command instructs the bot to pause for a specified period of time.
- The "die" command, which can be seen by a string compare at address 0x401707. This command instructs the bot to exit.
Conclusions

This brings us almost to the end of this section. We covered a lot of material, and looked at several malware specimens.
Where to go from Here?

- There is a plethora of malicious code to explore
  - See the websites
- Start researching new techniques used for DLL injection, hooking, etc.
- Remember: if you get stuck examining a specimen, don't give up!

At this point, you may be wondering what options are available to you. There is a plethora of malicious code, that you can download and examine.

Malicious code is constantly evolving, as the authors learn new tactics. It is very beneficial to start researching new techniques and tactics used by the computer underground.

Above all, remember: If you get stuck examining a specimen, don't give up!
Here are some websites that contain good information about reverse engineering.

The site offensivecomputing.net also contains malicious code specimens available for download.
Books

Rootkits, Spyware/Adware, Keyloggers and Backdoors – Zaytsev  
Microsoft Windows Internals – Russinovich & Solomon  
Reversing: Secrets of Reverse Engineering – Eliam  
Assembly Language for Intel-Based Computers – Irvine  
Hacker Debugging Uncovered – Kaspersky  
Hacker Disassembling Uncovered – Kaspersky  
Rootkits, Subverting the Windows Kernel – Hoglund & Butler  
Virus Research and Defense – Szor  
Disassembling Code IDA Pro and SoftICE – Pirogov  
The IDA PRO Book – Eagle

Here are some books that have material that is relevant to reverse engineering.
Additional Exercises (Optional)

- Here are additional exercises to reinforce the topics we covered.
- If you'd like even more practice: A few additional specimens for your exploration in extra-malware.zip.

If you'd like more practice, take a look at following exercises when you get the chance. For even more opportunities to practice reverse-engineering, you will find additional malicious executables in the extra-malware.zip file on the DVD you received for this course.
Exercise 10: Reverse Blinky

- Understand meaning of a function
- Function is at address 0x401188
- Determine flow of execution
- Describe what function does
  - In your own words
  - Include error checking or weaknesses
- Extra credit: write pseudo code for function

The goal of this exercise is to understand the meaning of a function. This goes beyond just tracing the flow of a program, and recognizing function calls.

During the course of a reverse engineering session, identifying the limits of a malicious code specimen can be very useful. For instance, being able to identify a list of servers a specimen contacts can be used to develop IDS alerts. Alternatively, identifying weaknesses in communication protocols used by malicious code can be used to help minimize and control a malicious code infestation.

You will be analyzing the same specimen as before, Blinky. This time you will reverse a function at address 0x401188. Your task is to determine the flow of execution for the function, and to describe in your own words, what the function does. When describing what the function does, include any error checking or weaknesses of the function.

For extra credit, write up short pseudocode that describes the function's operation.
Here are some hints to help you understand the purpose of the function.

First, look at any system calls that the function makes. Also look at strings that are referenced. Be careful though, it is easy to plant false strings, or modify strings. As a result, strings should usually be used for guidance, rather than for definitive conclusions.

Also look how the function is called, what other parts of the program call the function? When do they call it?

Try to identify any variables that are used, and their purpose. You will have to do this by examining how the variables are used.

Finally, ask yourself "what is going on?" Sometimes just answering this question is enough to understand the purpose of the function.
Hints (2)

- Return values for URLDownloadToFile
  - 0x0 is S_OK
    - Success
  - 0x8007000E is E_OUTOFMEMORY
    - Insufficient memory to complete operation
  - 0x800C0008 is
    INET_E_DOWNLOAD_FAILURE
    - If URL to download is invalid
- Look at last exercise to determine what function at address 0x401400 does

Here are some additional hints to help you out.

The URLDownloadToFile function has three different return values. They are:

- If the call is successful, the value 0x0 (symbolic constant S_OK) is returned.
- If there is insufficient memory to complete the operation, the value 0x8007000E
  (symbolic constant E_OUTOFMEMORY) is returned.
- If the URL used to download the file is invalid, the value 0x800C0008 (symbolic constant
  INET_E_DOWNLOAD_FAILURE) is returned.

The function at address 0x401400 is called at the end of the function you are examining. Refer to the
last exercise (Exercise 9) to determine what the function does.
Here are the answers to the exercise.

The function downloads a file using the `URLDownloadToFile` function. It checks the result of the function call for three values.

If the return value is `E_OUTOFMEMORY`, the function sends the string “Result: out of memory”. If the return value is `INET_E_DOWNLOAD_FAILURE`, the function sends the string “Result: download failure”. If the return value is `S_OK`, the function sends the string “Result: ok”. Otherwise, the function sends the string “Result: ” followed by the numeric value of the result (in hex).

To summarize, the function downloads a file, checks for various error conditions, and sends the results back to the attacker.
Answers (2) Extra Credit
Reverse Blinky

download_file(arg1, url, dest_file) {
    result = URLLDownloadToFile(0, url, dest_file, 0, 0)
    if result is E_OUTOFMEMORY
        send "Result: out of memory"
    else if result is INET_E_DOWNLOAD_FAILURE
        send "Result: download failure"
    else if result is S_OK
        send "Result: ok"
    otherwise
        send "Result: " (followed by numeric result in hex)
}
Exercise 11: Reverse Spybot

- Reverse other portions of spybot.exe
  - Same specimen as exercise 7
- Multi-functional bot
  - Built-in HTTP server
  - Spreading capabilities
- Determine what various functions do
  - See if you can build network or host signatures to identify specimen
- Spend rest of class
  - Continue on your own

The last exercise of the day is for you to reverse portions of Spybot. You will be analyzing the same specimen used in exercise 7.

Spybot is a multi-functional bot, with many capabilities including a built-in HTTP server, and the ability to spread to multiple different systems.

Your task is to choose functions and determine what they do. One common goal in reverse engineering is to develop network and host based signatures to identify the specimen activity in an organization. While reverse engineering this specimen, see if you could identify signatures that could be used in a corporate environment to find the specimen.

You will have the rest of class time to analyze the specimen. You can also continue to analyze the specimen outside of class.
End of FOR610.3

- You have now completed FOR610.3.

You have reached the end of FOR610.3. Course materials in the subsequent sections will build upon the concepts and skills we covered in the prior pages.
This appendix presents a reference for commonly-used Intel x86 assembly instructions, which you are likely to encounter when analyzing malicious software. It is an appendix to FOR610 course materials, and is particularly relevant for the topics discussed in FOR610.3. This appendix was written by Michael Murr.
List of Instructions

- List of commonly used 80x86 assembly instructions
  - Not an extensive list, as this is not meant to be a course in assembly programming
  - See references at the end of this appendix for further information

Assembly is an architecture-dependent programming language (although there is a common subset amongst each processor family). We'll stick to just instructions, skipping topics such as the different ways of addressing memory, or identifying high-level logic structures. For more information there are various resources on the Internet, some of which are listed at the end of this appendix.

Throughout this appendix the code samples are in Courier font. Comments (things ignored by the assembler) are preceded with a semicolon (";"). An assembly instruction can be thought of as having two primary parts, the assembly instruction (sometimes referred to as an opcode or operation code), which is the instruction itself, as well as zero, one, or two arguments which are called operands.
<table>
<thead>
<tr>
<th>Arithmetic instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ADD</td>
</tr>
<tr>
<td>• INC</td>
</tr>
<tr>
<td>• MUL</td>
</tr>
<tr>
<td>• SHL/SHR</td>
</tr>
</tbody>
</table>

Let's take a look at some of the basic arithmetic instructions.
ADD dest, src

- Adds source to destination
- Result is stored in destination

\[
\begin{align*}
\text{MOV EAX, 0x08} & \quad ; \ EAX = 0x08 \ (8 \text{ decimal}) \\
\text{MOV EBX, 0x02} & \quad ; \ EBX = 0x02 \ (2 \text{ decimal}) \\
\text{ADD EAX, EBX} & \\
\end{align*}
\]

After the ADD, EAX equals 0x0A (10 decimal), EBX is still 0x02 (2 decimal).

The ADD instruction adds two operands (destination and source) together using integer arithmetic. The result is stored in the destination operand.

\[
\begin{align*}
\text{MOV EAX, 0x08} & \quad ; \ EAX = 0x08 \ (8 \text{ decimal}) \\
\text{MOV EBX, 0x02} & \quad ; \ EBX = 0x02 \ (2 \text{ decimal}) \\
\text{ADD EAX, EBX} & \\
\end{align*}
\]

In the example on this slide, after the ADD operation, EAX equals 0x0A (10 decimal), EBX is still 0x02 (2 decimal).
SUB dest, src

- Subtracts the source from the destination
- Result is stored in destination

MOV EAX, 0x08 ; EAX = 0x08 (8 decimal)
MOV EBX, 0x02 ; EBX = 0x02 (2 decimal)
SUB EAX, EBX

After the SUB, EAX equals 0x06 (6 decimal), EBX is still 0x02 (2 decimal).

The SUB instruction subtracts the source operand from the destination operand. The result of the subtraction is stored in the destination operand.

MOV EAX, 0x08 ; EAX = 0x08 (8 decimal)
MOV EBX, 0x02 ; EBX = 0x02 (2 decimal)
SUB EAX, EBX

In the example on this slide, after the SUB operation, EAX equals 0x06 (6 decimal), EBX is still 0x02 (2 decimal).
INC dest

- Increments the destination by 1

```assembly
MOV EAX, 0x08 ; EAX = 0x08 (8 decimal)
INC EAX
```

After the INC, EAX equals 0x09 (9 decimal).

The INC instruction increments the destination operand by 1. This is the same thing as doing an ADD instruction with the same destination, and using the source of 1.

```assembly
MOV EAX, 0x08 ; EAX = 0x08 (8 decimal)
INC EAX
```

In the example on this slide, after the INC operation, EAX equals 0x09 (9 decimal).
DEC dest

- Decrements the destination by 1

\[
\text{MOV EAX, 0x08} \quad ; \text{EAX = 0x08 (8 decimal)}
\]
\[
\text{DEC EAX}
\]

After the DEC, EAX equals 0x07 (7 decimal).

The DEC instruction decrements the destination operand by 1. This is the same thing as doing a SUB instruction with the same destination, and using a source of 1.

\[
\text{MOV EAX, 0x08} \quad ; \text{EAX = 0x08 (8 decimal)}
\]
\[
\text{DEC EAX}
\]

After the DEC, EAX equals 0x07 (7 decimal).
MUL src

- Multiplies AL/AX/EAX by source (Unsigned integer multiplication)
- Result is stored in AL/DX:AX/EDX:EAX

    MOV EAX, 0x08 ; EAX = 0x08 (8 decimal)
    MOV EBX, 0x02 ; EBX = 0x02 (2 decimal)
    MUL EBX

    After the MUL, EDX:EAX equals 0x10 (16 decimal), EBX is still 0x02 (2 decimal).

The MUL instruction multiplies either AL, AX, or EAX by the operand. Which register gets multiplied is determined by the size (in number of bits) of the source operand. If the source operand is 8 bits, AL is used. If the source operand is 16 bits, AX is used. If the source operand is 32 bits, EAX is used.

The location of the result depends on the size of the source operand. Since multiplication of two numbers can result in a larger sized result, a bigger destination is needed. If the source operand is 8 bits, the result is stored in AX (16 bits). If the source operand is 16 bits, the result is stored in DX:AX (32 bits total). If the source operand is 32 bits, the result is stored in EDX:EAX (64 bits total).

<table>
<thead>
<tr>
<th>Multiplier (operand)</th>
<th>Multiplicand</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>AL</td>
<td>AX</td>
</tr>
<tr>
<td>16 bits</td>
<td>AX</td>
<td>DX:AX</td>
</tr>
<tr>
<td>32 bits</td>
<td>EAX</td>
<td>EDX:EAX</td>
</tr>
</tbody>
</table>
DIV src

- Divides AX/DX:AX/EAX:EDX by source (Unsigned integer division)
- Quotient is stored in AL/AX/EAX and remainder in AH/DX/EDX

MOV EAX, 0x8003 ; EAX = 0x8003 (32771 decimal)
MOV EBX, 0x100 ; EBX = 0x100 (256 decimal)
DIV EBX

After the DIV, EAX equals 0x80 (128 decimal), EDX equals 0x03 (3 decimal).

The DIV instruction divides AX, DX:AX, or EAX:EDX by the source operand. The result is stored in either AL, AX, or EAX, and the remainder is stored in either AH, DX, or EDX. The size of the source operand determines what is used for division, and where the quotient and remainder are stored. If the source operand is 8 bits, AX is divided by the operand, and the result is stored in AL, with the remainder in AH. If the source operand is 16 bits, DX:AX is divided by the operand, and the result is stored in AX with the remainder in DX. If the source operand is 32 bits, EDX:EAX is divided by the operand, and the quotient is stored in EAX with the remainder in EDX.

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor (operand)</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>8 bits</td>
<td>AL</td>
<td>AH</td>
</tr>
<tr>
<td>DX:AX</td>
<td>16 bits</td>
<td>AX</td>
<td>DX</td>
</tr>
<tr>
<td>EDX:EAX</td>
<td>32 bits</td>
<td>EAX</td>
<td>EDX</td>
</tr>
</tbody>
</table>
Shifting (SHL/SHR)

- SHL dest, src
- SHR dest, src
  - Shifts the destination to the left (SHL) or to the right (SHR) by source bits (fill with 0)

The SHL and SHR instructions shift the destination operand to the left or right (respectively) source number of bits. The destination is zero-padded, meaning as new bits are needed, a zero is used. During a SHL the leftmost bit (or the rightmost bit in the case of a SHR) is moved into the carry flag (CF).

MOV EAX, 0x03 ; EAX = 0x03 (3 decimal)
SHL EAX, 1

After the SHL operation, EAX equals 0x06 (6 decimal)

MOV EAX 0x03 ; EAX = 0x03 (3 decimal)
SHR EAX, 1

After the SHR operation, EAX equals 0x01 (1 decimal)
Rotating (ROL/ROR)

- ROL dest, src
- ROR dest, src
  - Rotates the destination to the left (SHL) or to the right (SHR) by source bits

Rotating bits either to the left (ROL) or right (ROR) is very similar to shifting, except in the case of ROL, the leftmost bit is copied into both the carry flag (CF) and the rightmost bit. In the case of ROR, the rightmost bit is copied into both the carry flag (CF) and the leftmost bit.

```assembly
MOV AL, 0x01 ; AL = 0x01 (1 decimal)
ROL AL, 2

After the ROL operation, AL equals 0x04 (4 decimal)

MOV AL, 0x01 ; AL = 0x01 (1 decimal)
ROR AL, 2

After the ROL operation, AL equals 0x40 (64 decimal)
```
## Boolean instructions

- Flags
- OR
- NEG
- TEST
- AND
- NOT
- XOR
- CMP

Now, let's take a look at some of the instructions that deal with boolean operations.
Flags

- 16 bit register where each bit has a different meaning
  - Control bits control CPU operation
  - Status bits indicate status of an operation

Let's talk about registers for a moment. A register is really just a named, on-chip storage location. Since the registers are on-chip, their access time is quite fast. There are some general purpose registers (EAX, EBX, ECX, EDX, etc.) and some status registers (Flags, etc.) The flags register is important to us because the individual bits in the flag register represent the side effects of various instructions (e.g. ADD/SUB/MUL, etc.)

A few of the individual flags we are interested in:

O: Overflow – Set if the result of a signed arithmetic instruction generates a number which is either too large or too small for the destination.

D: Directional – This bit controls the direction that string instructions (e.g. SCAS) operate (either 1 for high memory addresses to low memory addresses, or 0 for low memory addresses to high memory addresses.).

S: Sign – This bit is set if the instruction resulted in a number that is signed

Z: Zero – This bit is set if the instruction resulted in a zero (the result is zero).

C: Carry – This bit is set if the result of an unsigned arithmetic instruction is too large to fit in the destination.
AND dest, src

- Performs a bitwise AND of the destination and source
- Result stored in destination

MOV EAX, 0xF023 ; EAX = 0xF023 (61475 decimal)
MOV EBX, 0x03AC ; EBX = 0x03AC (940 decimal)
AND EAX, EBX

After the AND, EAX equals 0x20 (32 decimal), EBX is still 0x03AC (940 decimal).

The AND instruction performs a bitwise logical “AND” of the destination and source operands. The result of this operation is stored in the destination.

MOV EAX, 0xF023 ; EAX = 0xF023 (61475 decimal)
MOV EBX, 0x03AC ; EBX = 0x03AC (940 decimal)
AND EAX, EBX

After the AND operation, EAX equals 0x20 (32 decimal), EBX is still 0x03AC (940 decimal).

```
1111 0000 0010 0011 (EAX)
0000 0011 1010 1100 (EBX)
------------------------ (AND)
0000 0000 0010 0000 (Result)

0000 0000 0010 0000 == 0x20 (32 decimal)
```
OR dest, src

- Performs a bitwise OR of the destination and source
- Result stored in destination

\begin{verbatim}
MOV EAX, 0xF023 ; EAX = 0xF023 (61475 decimal)
MOV EBX, 0x03AC ; EBX = 0x03AC (940 decimal)
OR EAX, EBX

After the OR, EAX equals 0xF3AF (62383 decimal), EBX is still 0x03AC (940 decimal).
\end{verbatim}

The OR instruction performs a bitwise logical "OR" of the destination and source operands. The result of this operation is stored in the destination.

\begin{verbatim}
MOV EAX, 0xF023 ; EAX = 0xF023 (61475 decimal)
MOV EBX, 0x03AC ; EBX = 0x03AC (940 decimal)
OR RAX, RBX

After the OR operation, EAX equals 0xF3AF (62383 decimal), EBX is still 0x03AC (940 decimal).
\end{verbatim}

1111 0000 0010 0011 (EAX)
0000 0011 1010 1100 (EBX)
------------------------ (OR)
1111 0011 1010 1111 (Result)

1111 0011 1010 1111 == 0xF3AF (62383 decimal)
NOT dest

- Inverts all of the bits in destination
- Also called the one's complement

```
MOV EAX, 0xF023 ; EAX = 0xF023 (61475 decimal)
NOT EAX

After the NOT, EAX equals 0xFDC (4060 decimal).
```

The NOT instruction inverts all of the bits in the destination. This is also known as one's complement. The result of this operation is stored in the destination.

```
MOV EAX, 0xF023 ; EAX = 0xF023 (61475 decimal)
NOT EAX

After the NOT operation, EAX equals 0xFDC (4060 decimal).

1111 0000 0010 0011 (EAX)
----------------------- (NOT)
0000 1111 1101 1100 (Result)

0000 1111 1101 1100 == 0xFDC (4060 decimal)
```
NEG dest

- Negates all of the bits in destination
  - Invert all of the bits, and then add 1
- Also called the two's complement

MOV EAX, 0xF023 ; EAX = 0xF023 (61475 decimal)
NEG EAX

After the NEG, EAX equals 0x0FDD (4061 decimal).

The NEG instruction negates the destination operand. It negates the operand using two's complement (essentially it inverts all of the bits in the destination operand, and then adds 1). The result of this operation is stored in the destination.

MOV EAX, 0xF023 ; EAX = 0xF023 (61475 decimal)
NEG EAX

After the NEG operation, EAX equals 0x0FDD (4061 decimal).

1111 0000 0010 0011 (EAX)
---------------------- (NEG)
0000 1111 1101 1101 (Result)

0000 1111 1101 1101 == 0x0FDD (4061 decimal)
XOR dest, src

- Performs a bitwise XOR of the destination and source
- Result stored in destination

```
MOV EAX, 0xF023  ; EAX = 0xF023 (61475 decimal)
MOV EBX, 0x03AC  ; EBX = 0x03AC (940 decimal)
XOR EAX, EBX
```

After the XOR, EAX equals 0xF38F (62351 decimal), EBX is still 0x03AC (940 decimal).

The XOR instruction performs a bitwise logical exclusive-or (XOR) of the destination and source operands. The result of this operation is stored in the destination.

```
MOV EAX, 0xF023  ; EAX = 0xF023 (61475 decimal)
MOV EBX, 0x03AC  ; EBX = 0x03AC (940 decimal)
XOR EAX, EBX
```

After the OR operation, EAX equals 0xF38F (62383 decimal), EBX is still 0x03AC (940 decimal).

```
1111 0000 0010 0011 (EAX)
0000 0011 1010 1100 (EBX)
------------------------ (XOR)
1111 0011 1000 1111 (Result)
```

```
1111 0011 1000 1111 == 0xF38F (62351 decimal)
```
TEST dest, src

- Performs a bitwise AND of the destination and source
- Only changes the status flags

```assembly
MOV EAX, 0xF023 ; EAX = 0xF023 (61475 decimal)
MOV EBX, 0x03AC ; EBX = 0x03AC (940 decimal)
TEST EAX, EBX
```

After the TEST, EAX is still 0xF023 (61475 decimal), EBX is still 0x03AC (940 decimal).

The TEST instruction performs an implied AND between the source and destination operands. The result isn’t stored anywhere, instead the various flags within the flags register are set.

You may be asking what is the point of this? Simple: by examining the various flags, we can create conditional jumps. That is, we can have conditional branches within our code. The instructions such as JNAE, JZ, JC, etc. are all conditional jumps that examine the flags register to determine to jump or not.
CMP dest, src

- Subtracts source from destination
- Only changes the status flags

EAX = 0xf023 (61475 decimal)
EBX = 0x03AC (940 decimal)
CMP EAX, EBX

After the CMP, EAX is still 0xF023 (61475 decimal), EBX is still 0x03AC (940 decimal).

The CMP instruction is similar to the TEST instruction. The CMP instruction performs an implied subtraction (SUB) of the source operand from the destination operand. The result isn’t stored anywhere, instead the various flags within the flags register are set. The reasons for this are the same as with the TEST instruction.
Now let's take a look at some of the instructions that control the flow of a program.

<table>
<thead>
<tr>
<th>Control Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
</tr>
<tr>
<td>LOOP</td>
</tr>
<tr>
<td>POP</td>
</tr>
<tr>
<td>RET/RETN</td>
</tr>
<tr>
<td>SCAS</td>
</tr>
<tr>
<td>LEA</td>
</tr>
<tr>
<td>JMP/JXX</td>
</tr>
<tr>
<td>PUSH</td>
</tr>
<tr>
<td>CALL</td>
</tr>
<tr>
<td>REP/REPXX</td>
</tr>
<tr>
<td>NOP</td>
</tr>
</tbody>
</table>
MOV dest, src

- Copies source into destination

MOV EAX, 0xF023 ; EAX = 0xF023 (61475 decimal)
MOV EBX, 0x03AC ; EBX = 0x03AC (940 decimal)
MOV EAX, EBX

After the MOV, EAX and EBX are both equal to 0x03AC (940 decimal).

The MOV instruction copies the source operand into the destination operand.

MOV EAX, 0xF023 ; EAX = 0xF023 (61475 decimal)
MOV EBX, 0x03AC ; EBX = 0x03AC (940 decimal)
MOV EAX, EBX

After the MOV operation, EAX and EBX are both equal to 0x03AC (940 decimal).
JMP dest

- Transfers control to destination ("jumps" to destination)
- Always jumps to destination—there is no special condition that needs to be true

JMP 0x804A8084

After the JMP, the processor will start executing instructions at memory location 0x804A8084.

The JMP instruction allows you to jump to another portion of code. JMP will always jump to the instruction, there are no special conditions (special flag settings) necessary.

JMP 0x804A8084

After the JMP operation, the processor will start executing instructions at memory location 0x804A8084.
Conditional JXX (1)

- JXX dest
- Conditionally transfers control to destination ("jumps" to destination)
- Only jumps to destination if a specific condition has been met

The other type of JMPs are conditional jumps. I refer to these as the JXX family. These instructions are in the form of J<condition>, where <condition> is a two- or three-letter shorthand description of the condition required to jump to another point in code. The next few slides will describe the various JXX instructions.

JXX instructions are normally preceded by a CMP instruction. Remember: the CMP instruction stands for Compare, it compares two operands by performing an implied subtraction of the source operand from the destination operand. Since CMP performs an implied subtraction, neither operand is modified. However, the various flags in the flags register (Zero, Carry, etc.) are set. A condition is determined to be met or not based off of the individual flags that are set in the flags register.
The first class of conditional JXX instructions are jumps with conditions that depend on general comparisons. The instructions are:

**Jump if Equal (JE) / Jump if Zero (JZ)**

Jumps if the two operands are equal, or if they are zero. This condition is met if the Zero Flag (ZF) in the flags register is set to 1.

**Jump if Not Equal (JNE) / Jump if Not Zero (JNZ)**

Jumps if the two operands are different, or not zero. This condition is met if the Zero Flag (ZF) in the flags register is set to 0.

**Jump if Carry (JC)**

Jumps if there was a carry. This condition is met if the Carry Flag (CF) is set to 1.

**Jump if No Carry (JNC)**

Jumps if there was not a carry. This condition is met if the Carry flag is set to 0.

**Jump if CX is Zero (JCXZ) / Jump if ECX is Zero (JECXZ)**

Jumps if CX or ECX is set to zero.

**Jump if Parity (JP)**

Jumps if the parity is even. This condition is met if the parity flag is set to 1.

**Jump if Not Parity (JNP)**

Jumps if the parity is odd. This condition is met if the parity flag is set to 0.
Conditional JXX (3)

- Unsigned comparison jumps
  - JA / JNBE
  - JAE / JNB
  - JB / JNAE
  - JBE / JNA

The next family of JXX instructions are jumps with conditions that depend on comparisons where both operands are treated as unsigned numbers.

Jump if Above (JA) / Jump if Not Below or Equal (JNBE)
These instructions jump if the first operand is bigger than the second operand (operand1 > operand2). This condition is met if both the Carry Flag (CF) and Zero Flag (ZF) are set to 0.

Jump if Above or Equal (JAE) / Jump if Not Below (JNB)
These instructions jump if the first operand is equal to or bigger than the second operand (operand1 >= operand2). This condition is met if the Carry Flag (CF) is set to 0.

Jump if Below (JB) / Jump if Not Above or Equal (JNAE)
These instructions jump if the first operand is less than the second operand (operand1 < operand2). This condition is met if the Carry Flag (CF) is set to 1.

Jump if Below or Equal (JBE) / Jump if Not Above (JNA)
These instructions jump if the first operand is less than or equal to the second operand (operand1 <= operand2). This condition is met if the Carry Flag (CF) or the Zero Flag (ZF) are set to 1.
Conditional JXX (4)

- Signed comparison jumps
  - JG / JNLE
  - JNG / JLE
  - JGE / JNL
  - JS / JNS
  - JL / JNGE
  - JO / JNO

The last set of JXX instructions are jumps with conditions that depend on signed comparisons.

**Jump if Greater than (JG) / Jump if Not Less than or Equal (JNLE)**
These instructions jump if the first operand is greater than second operand (operand1 > operand2). This condition is met if both the Sign Flag (SF) and Zero Flag (ZF) are set to 0.

**Jump if Not Greater than (JNG) / Jump if Less than or Equal (JLE)**
These instructions jump if the first operand is less than or equal to the second operand (operand1 ≤ operand2). This condition is met if the Zero Flag (ZF) equals 1 or the Sign Flag (SF) does not equal the ZF.

**Jump if Greater than or Equal (JGE) / Jump if Not Less than (JNL)**
These instructions jump if the first operand is greater than or equal to the second operand (operand1 ≥ operand2). This condition is met if the Sign Flag (SF) equals the Zero Flag (ZF).

**Jump if Signed (JS) / Jump if Not Signed (JNS)**
The JS instruction jumps if the number is signed. This condition is met if the Sign Flag (SF) is set to 1. The JNS instruction jumps if the number is not signed. This condition is met if the Sign Flag (SF) is set to 0.

**Jump if Less than (JL) / Jump if Not Greater than or Equal (JNGE)**
These instructions jump if the first operand is less than the second operand (operand1 < operand2). This condition is met if the Sign Flag (SF) does not equal the Zero Flag (ZF).

**Jump if Overflow (JO) / Jump if No Overflow (JNO)**
The JO instruction jumps if there was an overflow. This condition is met if the Overflow Flag (OF) is set to 1. The JNO instruction jumps if there was not an overflow. This condition is met if the Overflow Flag (OF) is set to 0.
LOOP dest

- Repeats a segment of code (between dest and the LOOP) CX number of times

MOV CX, 0x08
0x8043A8048:
  <some code>
LOOP 0x8043A8048

Will loop from 0x8043A048 to the LOOP 0x08 (8 decimal) times.

The LOOP instruction repeats a section of code between the destination operand (an address in memory) and the address of the LOOP instruction. The number of times that the loop is executed is determined by the number in the CX register. Each time the processor loops through the code, it decrements the CX register by one. When the CX register is decremented to 0, the processor executes the instruction immediately following the LOOP instruction.

MOV CX, 0x08  ; CX == 0x08 (8 decimal)
0x8043A8048:  ; This is address 0x8043A048
  <some code>
LOOP 0x8043A8048

Will loop from 0x8043A048 to the LOOP 0x08 (8 decimal) times.
PUSH src

- Pushes the source onto the stack
  - For later retrieval by POP

```assembly
MOV EAX, 0x26 ; EAX = 0x26 (decimal 38)
PUSH EAX
```

After the PUSH, the value of EAX (0x26 or 38 decimal) will be copied onto the stack.

The PUSH instruction copies the source operand onto the system stack. The system stack is an area of memory that is used to temporarily hold data. The stack is a Last In First Out (LIFO) structure which means that the last item pushed onto the stack, is the first item popped off.

```assembly
MOV EAX, 0x26 ; EAX = 0x26 (decimal 38)
PUSH EAX
```

After the PUSH operation, the value of EAX (0x26 or 38 decimal) will be copied onto the stack.
POP dest

- POPs value from stack, and stores in destination
- Retrieves last value PUSHed onto the stack

; STACK = 0x26 (decimal 38)
MOV EAX, 0x123 ; EAX = 0x123 (decimal 291)
POP EAX

After the POP, the value of EAX will be 0x26 (decimal 38).

The POP instruction copies the value sitting at the top of the system stack into the destination operand. As stated on the previous page, the system stack is an area of memory that is used to temporarily hold data. The stack is a Last In First Out (LIFO) structure which means that the last item pushed onto the stack, is the first item popped off.

; STACK = 0x26 (decimal 38)
MOV EAX, 0x123 ; EAX = 0x123 (decimal 291)
POP EAX

After the POP operation, the value of EAX will be 0x26 (decimal 38).
CALL dest

- Calls a function at the memory address destination
- Saves a copy of the current location on the stack so you can return later

CALL 0x8043A084

After the CALL, the CPU will begin executing instructions at memory address 0x8043A084.

The CALL instruction calls a function located at the memory address specified by the destination operand. What happens (behind the scenes) is a copy of the address of the current instruction is PUSHed onto the stack, and then control is transferred to the memory location specified by the destination operand. The address of the current instruction is PUSHed onto the stack so that when the function has finished executing, the program knows where to return control. This is the return address that is overwritten in many buffer overflow attacks.

CALL 0x8043A084

After the CALL operation, the CPU will begin executing instructions at memory address 0x8043A084
RETN

- Returns from a function call
- Returns to address that was previously saved on the stack by a CALL command

The RETN instruction normally indicates the end of a function. Think of it as returning from a function. What happens is that the address that was previously PUSHed onto the stack by the CALL instruction is now POPed off, so that the program can continue executing from where the most recent CALL instruction was. Many classic buffer overflow attacks would overwrite this address so when RETN is executed, execution is transferred not to the original CALL command, but somewhere else specified by the attacker.
REP cmd

• Repeats a specific string-related command using CX as a counter
• Decrements CX each time until CX is 0

; CX = 0x03 (decimal 3)
REP MOVSB

The MOVSB instruction will be executed 3 times. At the end, CX will be 0x0 (0 decimal).

The REP instruction repeats a string related instruction using CX as a counter. For example, if CX is 0x03 (decimal 3), then REP MOVSB will execute the MOVSB instruction 3 times. Each time the CX register is decremented by one, until it is eventually 0.

; CX = 0x03 (decimal 3)
REP MOVSB

The MOVSB instruction will be executed 3 times. At the end, CX will be 0x0 (0 decimal).
REPXX Instructions

- REPXX cmd
- Repeats a specific string-related command until a specific condition is met
  - REPZ
  - REPE
  - REPNZ
  - REPNE

The REPXX set of instructions are similar to the REP instruction in that they execute a given string related instruction the number of times specified by the CX register, except each specific REPXX instruction has an additional requirement for execution. Each time the string related instruction is executed, CX is decremented by 1.

**REpeat while Zero (REPZ) / REpeat while Equal (REPE)**
Repeats the string related instruction while CX is greater than 0 AND the Zero Flag (ZF) is set.

**REpeat while Not Zero (REPNZ) / REpeat while Not Equal (REPE)**
Repeats the string related instruction while CX is greater than 0 AND the Zero Flag (ZF) is not set.
SCAS Instructions

- SCASB/SCASW/SCASD
- Scans a string pointed to by EDI for a value that matches the value in AL/AX/EAX
- Usually used in conjunction with REPXX commands

The SCAS instruction scans a string pointed at by ES:EDI for a value that matches the value in AL, AX, or EAX. The SCAS instruction requires that an operand be specified. The SCASB instruction scans 1 byte (8 bits) at a time. SCASW compares the word at ES:DI with the value in AL. The SCASW instruction scans 1 word (2 bytes, 16 bits) at a time. SCASW compares the word at ES:DI with the value in AX. The SCASD instruction scans 1 double word (4 bytes, 32 bits) at a time. SCASW compares the byte at ES:DI with the value in EAX. If the Direction Flag (DF) is set to 0, then EDI is incremented. If the Direction Flag (DF) is set to 1, then EDI is decremented.

The SCAS instructions are normally used in conjunction with the REPXX instructions to compare strings in memory.

```
: At memory address 0x8048123 is stored
: the string "Malware", DF = 0

MOV EAX, 0x77
: EAX = 0x77 (decimal 119) (ASCII 'w')
MOV EDI, 0x8048123
: EDI = 0x8048123 (decimal 134512931) the
: start of the string "Malware" in memory

REPNE SCASB
```

After the REPNE instruction, EDI will have the value 0x8048126 (decimal 134512934). This points to the address of the 'w' in the string "Malware".
NOP

- Does nothing (No OPeration)
- Commonly used for alignment purposes

The NOP instruction simply does nothing. It stands for No OPeration. This is commonly used for alignment purposes. The NOP instruction is also frequently used in classic buffer overflow attacks. A series of NOP instructions are used as padding, giving the attacker a margin of error when redirecting the program control. This is called a NOP sled.
LEA dest, src

- Loads the address of source and places it in destination
- Source can be a calculation

```
MOV EBX, 0x8043A00 ; EBX = 0x8043A00
LEA EAX, EBX+4
```

After the LEA, EAX will have the value 0x8043A04, EBX will still have the value 0x8043A00.

The LEA instruction loads the address of the source operand into the destination operand. The source operand can be a calculation. The usefulness of this command is that the calculation is done at runtime. That is, the value that gets stored in the destination won’t be known until the instruction is executed. This is different from doing a traditional MOV instruction because the MOV must either have a numeric value (e.g. MOV EBX, 0x8043A00), or the MOV instruction cannot have a calculation (e.g. MOV EBX, EAX is legal while MOV EBX, EAX+4 is not).

```
MOV EBX, 0x8043A00 ; EBX = 0x8043A00
LEA EAX, EBX+4
```

After the LEA operation, EAX will have the value 0x8043A04, EBX will still have the value 0x8043A00.
End of Appendix

- Brief summary of 80x86 instructions
- Further reference material

This ends the appendix on commonly used assembly instructions. Again, this is not meant to be an exhaustive reference, nor is it meant to teach you how to program in x86 assembly.

If you are interested, you can find further material on x86 assembly at:

The Art of Assembly Language Programming and HLA
http://webster.cs.ucr.edu/

Gavin's Guide to 80x86 Assembly
http://burks.brighton.ac.uk/burks/language/asm/asmtut/asm1.htm

Intel® Pentium® Processor Manuals
http://www.intel.com/design/pentium/MANUALS/index.htm