Game Hacking

Developing Autonomous Bots for Online Games

EARLY ACCESS

Nick Cano

no starch press

NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

Welcome to the Early Access edition of the as yet unpublished *Game Hacking* by Nick Cano! As a prepublication title, this book may be incomplete and some chapters may not have been proofread.

Our goal is always to make the best books possible, and we look forward to hearing your thoughts. If you have any comments or questions, email us at **earlyaccess@nostarch.com**. If you have specific feedback for us, please include the page number, book title, and edition date in your note, and we'll be sure to review it. We appreciate your help and support!

We'll email you as new chapters become available. In the meantime, enjoy!

GAME HACKING

NICK CANO

Early Access edition, 12/16/15

ISBN-10: 1593276699
ISBN-13: 978-1593276690

Publisher: William Pollock
Production Editor: Laurel Chun
Cover Illustration: Ryan Milner
Developmental Editor: Jennifer Griffith-Delgado
Technical Reviewer: Stephen Lawler
Copyeditor: Rachel Monaghan

BRIEF CONTENTS

Introduction

PART I: TOOLS OF THE TRADE
Chapter 1: Scanning Memory Using Cheat Engine 3
Chapter 2: Debugging Games with OllyDbg 23
Chapter 3: Reconnaissance with Process Monitor and Process Explorer 49

PART II: GAME DISSECTION
Chapter 4: From Code to Memory: A General Primer 65
Chapter 5: Advanced Memory Foresics 97
Chapter 6: Programmatically Reading Memory

PART III: FULL CONTROL
Chapter 7: Code Injection
Chapter 8: Control Flow Manipulation

PART IV: COMMON HACKS AND AUTOMATION
Chapter 9: Memory Hacks
Chapter 10: Event-Driven Hacks
Chapter 11: Automated Hacks

PART V: STEALTH, MORALITY, AND LEGALITY
Chapter 12: Staying Hidden
Chapter 13: Moral and Legal Implications

CONTENTS IN DETAIL

PART I: TOOLS OF THE TRADE 1

1
SCANNING MEMORY USING CHEAT ENGINE 3

Why Memory Scanners Are Important 4
Basic Memory Scanning 4
Cheat Engine's Memory Scanner 5
- Scan Types 6
- Running Your First Scan 6
- Next Scans 7
- When You Can't Get a Single Result 7
- Cheat Tables 7

Memory Modification in Games 8
- Manual Modification with Cheat Engine 8
- Trainer Generator 9

Pointer Scanning 11
- Pointer Chains 11
- Pointer Scanning Basics 12
- Pointer Scanning with Cheat Engine 14
- Pointer Rescanning 17

Lua Scripting Environment 18
- Searching for Assembly Patterns 19
- Searching for Strings 21

Closing Thoughts 22

2
DEBUGGING GAMES WITH OLLYDBG 23

A Brief Look at OllyDbg's User Interface 24
OllyDbg's CPU Window 26
- Viewing and Navigating a Game's Assembly Code 27
- Viewing and Editing Register Contents 29
- Viewing and Searching a Game's Memory 29
- Viewing a Game's Call Stack 30

Creating Code Patches 30
Tracing Through Assembly Code 32
OllyDbg's Expression Engine 33
- Using Expressions in Breakpoints 33
- Using Operators in the Expression Engine 34
- Working with Basic Expression Elements 35
- Accessing Memory Contents with Expressions 35

OllyDbg Expressions in Action 36
Pausing Execution When a Specific Player's Name Is Printed 37
Pausing Execution When Your Character's Health Drops 38
OllyDbg Plug-ins for Game Hackers 42
Copying Assembly Code with Asm2Clipboard 42
Adding Cheat Engine to OllyDbg with Cheat Utility 42
Controlling OllyDbg Through the Command Line 43
Visualizing Control Flow with OllyFlow 45
Closing Thoughts 47

3
RECONNAISSANCE WITH PROCESS MONITOR AND PROCESS EXPLORER 49

Process Monitor 50
Logging In-Game Events 50
Inspecting Events in the Process Monitor Log 52
Debugging a Game to Collect More Data 54
Process Explorer 55
Process Explorer's User Interface and Controls 56
Examining Process Properties 57
Handle Manipulation Options 59
Closing Thoughts 61

PART II: GAME DISSECTION 63

4
FROM CODE TO MEMORY: A GENERAL PRIMER 65

How Variables and Other Data Manifest in Memory 66
Numeric Data 67
String Data 69
Data Structures 71
Unions 73
Classes and VF Tables 74
x86 Assembly Crash Course 78
Command Syntax 79
Processor Registers 81
The Call Stack 86
Important x86 Instructions for Game Hacking 89
Closing Thoughts 96

5
ADVANCED MEMORY FORENSICS 97

Advanced Memory Scanning 98
Deducing Purpose 98
Finding the Player's Health with OllyDbg 99
Determining New Addresses After Game Updates 102
Identifying Complex Structures in Game Data 105
The std::string Class. 105
The std::vector<T> Class 108
The std::list<T> Class 110
The std::map<T, T> Class 114
Closing Thoughts 118

PART I

TOOLS OF THE TRADE

1

SCANNING MEMORY USING CHEAT ENGINE

The best game hackers in the world spend years personalizing expansive arsenals with custom-built tools. Such potent toolkits enable these hackers to seamlessly analyze games, effortlessly prototype hacks, and effectively develop bots. At the core, however, each unique kit is built from the same four-piece powerhouse: a memory scanner, an assembler-level debugger, a process monitor, and a hex editor.

Memory scanning is the gateway to game hacking, and this chapter will teach you about Cheat Engine, a powerful memory scanner that searches a game's operating memory (which lives in RAM) for values like the player's level, health, or in-game money. First, I'll focus on basic memory scanning, memory modification, and pointer scanning. Following that, we'll dive into Cheat Engine's powerful embedded Lua scripting engine.

NOTE *You can grab Cheat Engine from* http://www.cheatengine.org/. *Pay attention when running the installer because it will try to install some toolbars and other bloatware. You can disable those options if you wish.*

Why Memory Scanners Are Important

Knowing a game's state is paramount to interacting with the game intelligently, but unlike humans, software can't determine the state of a game simply by looking at what's on the screen. Fortunately, underneath all of the stimuli produced by a game, a computer's memory contains a purely numeric representation of that game's state—and programs can understand numbers easily. Hackers use memory scanners to find those values in memory, and then in their programs, they read the memory in these locations to understand the game's state.

For example, a program that heals players when they fall below 500 health needs to know how to do two things: track a player's current health and cast a healing spell. The former requires access to the game's state, while the latter might only require a button to be pressed. Given the location where a player's health is stored and the way to read a game's memory, the program would look something like this pseudocode:

```
// do this in some loop
health = readMemory(game, HEALTH_LOCATION)
if (health < 500)
    pressButton(HEAL_BUTTON)
```

A memory scanner allows you to find `HEALTH_LOCATION` so that your software can query it for you later.

Basic Memory Scanning

The memory scanner is the most basic, yet most important, tool for the aspiring game hacker. As in any program, all data in the memory of a game resides at an absolute location called a *memory address*. If you think of the memory as a very large byte array, a memory address is an index pointing to a value in that array. When a memory scanner is told to find some value *x* (called a *scan value*, because it's the value you're scanning for) in a game's memory, the scanner loops through the byte array looking for any value equal to *x*. Every time it finds a matching value, it adds the index of the match to a result list.

Due to the sheer size of a game's memory, however, the value of *x* can appear in hundreds of locations. Imagine that *x* is the player's health, which is currently 500. Our *x* uniquely holds `500`, but `500` is not uniquely held by *x*, so a scan for *x* returns all variables with a value of `500`. Any addresses not related to *x* are ultimately clutter; they share a value of `500` with *x* only by

chance. To filter out these unwanted values, the memory scanner allows you to rescan the result list, removing addresses that no longer hold the same value as *x*, whether *x* is still 500 or has changed.

For these rescans to be effective, the overall state of the game must have significant *entropy*—a measure of disorder. You increase entropy by changing the in-game environment, often by moving around, killing creatures, or switching characters. As entropy increases, unrelated addresses are less likely to continue to arbitrarily hold the same value, and given enough entropy, a few rescans should filter out all false positives and leave you with the true address of *x*.

Cheat Engine's Memory Scanner

This section gives you a tour of Cheat Engine's memory-scanning options, which will help you track down the addresses of game state values in memory. I'll give you a chance to try the scanner out in Lab 1-1 on page 11; for now, open Cheat Engine and have a look around. The memory scanner is tightly encapsulated in its main window, as shown in Figure 1-1.

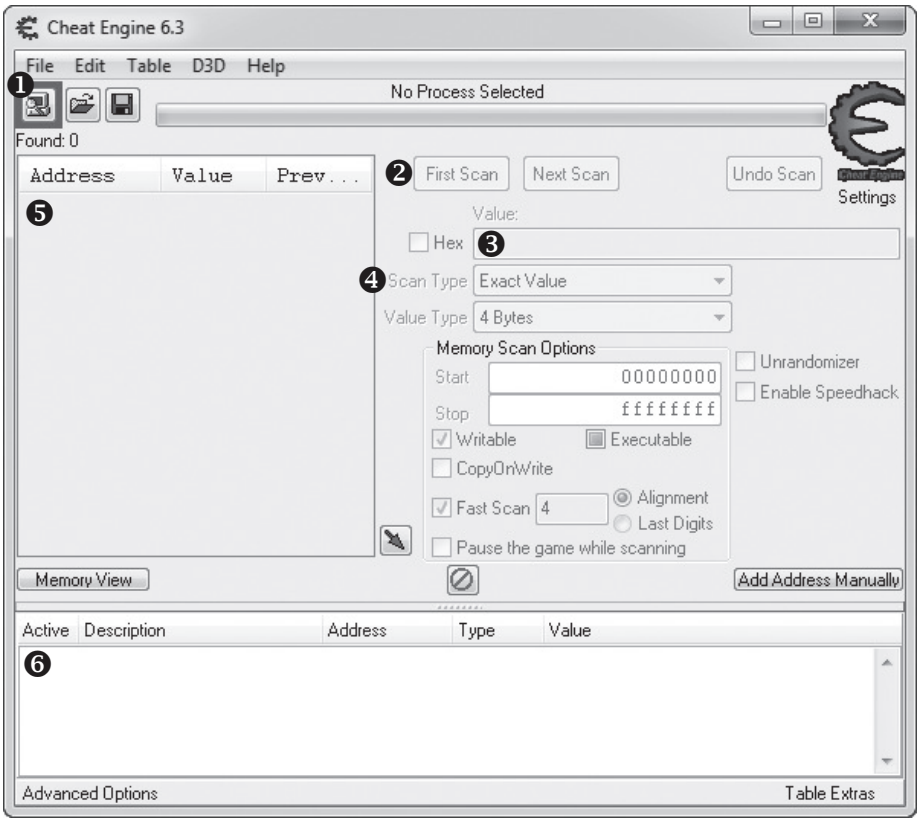


Figure 1-1: Cheat Engine main screen

To begin scanning a game's memory, click the Attach icon ❶ to attach to a process and then enter the scan value (referred to as *x* in our conceptual scanner) you want to locate ❸. By attaching to a process, we're telling

Cheat Engine to prepare to operate on it; in this case, that operation is a scan. It helps to also tell Cheat Engine what kind of scan to run, as I'll discuss next.

Scan Types

Cheat Engine allows you to select two different scan directives, called Scan Type and Value Type ❹. Scan Type tells the scanner how to compare your scan value with the memory being scanned using one of the following scan types:

Exact Value Returns addresses pointing to values equal to the scan value. Choose this option if the value you are looking for won't change during the scan; health, mana, and level typically fall into this category.

Bigger Than Returns addresses pointing to values greater than the scan value. This option is useful when the value you're searching for is steadily increasing, which often happens with timers.

Smaller Than Returns addresses pointing to values smaller than the scan value. Like Bigger Than, this option is useful for finding timers (in this case, ones that count down rather than up).

Value Between Returns addresses pointing to values within a scan value range. This option combines Bigger Than and Smaller Than, displaying a secondary scan value box that allows you to input a much smaller range of values.

Unknown Initial Value Returns all addresses in a program's memory, allowing rescans to examine the entire address range relative to their initial values. This option is useful for finding item or creature types, since you won't always know the internal values the game developers used to represent these objects.

The Value Type directive tells the Cheat Engine scanner what type of variable it's searching for.

Running Your First Scan

Once the two scan directives are set, click **First Scan** ❷ to run an initial scan for values, and the scanner will populate the results list ❺. Any green addresses in this list are *static*, meaning that they should remain persistent across program restarts. Addresses listed in black reside in *dynamically allocated memory*, memory that is allocated at runtime.

When the results list is first populated, it shows the address and real-time value of each result. Each rescan will also show the value of each result during the previous scan. (Any real-time values displayed are updated at an interval that you can set in Edit ▸ Settings ▸ General Settings ▸ Update Interval.)

Next Scans

Once the results list is populated, the scanner enables the Next Scan ❷ button, which offers six new scan types. These additional scan types allow you to compare the addresses in the results list to their values in the previous scan, which will help you narrow down which address holds the game state value you're scanning for. They are as follows:

Increased Value Returns addresses pointing to values that have increased. This complements the Bigger Than scan type by keeping the same minimum value and removing any address whose value has decreased.

Increased Value By Returns addresses pointing to values that have increased by a defined amount. This scan type usually returns far fewer false positives, but you can use it only when you know exactly how much a value has increased.

Decreased Value This option is the opposite of Increased Value.

Decreased Value By This option is the opposite of Increased Value By.

Changed Value Returns addresses pointing to values that have changed. This type is useful when you know a value will mutate, but you're unsure how.

Unchanged Value Returns addresses pointing to values that haven't changed. This can help you eliminate false positives, since you can easily create a large amount of entropy while ensuring the desired value stays the same.

You'll usually need to use multiple scan types in order to narrow down a large result list and find the correct address. Eliminating false positives is often a matter of properly creating entropy (as described in "Basic Memory Scanning" on page 4), tactically changing your scan directives, bravely pressing Next Scan, and then repeating the process until you have a single remaining address.

When You Can't Get a Single Result

Sometimes it is impossible to pinpoint a single result in Cheat Engine, in which case you must determine the correct address through experimentation. For example, if you're looking for your character's health and can't narrow it down to fewer than five addresses, you could try modifying the value of each address (as discussed in "Manual Modification with Cheat Engine" on page 8) until you see the health display change or the other values automatically change to the one you set.

Cheat Tables

Once you've found the correct address, you can double-click it to add it to the *cheat table pane* ❻; addresses in the cheat table pane can be modified, watched, and saved to cheat table files for future use.

For each address in the cheat table pane, you can add a description by double-clicking the Description column, and you can add a color by right-clicking and selecting Change Color. You can also display the values of each address in hexadecimal or decimal format by right-clicking and selecting Show as Hexadecimal or Show as Decimal, respectively. Lastly, you can change the data type of each value by double-clicking the Type column, or you can change the value itself by double-clicking the Value column.

Since the main purpose of the cheat table pane is to allow a game hacker to neatly track addresses, it can be dynamically saved and loaded. Go to **File ▸ Save** or **File ▸ Save As** to save the current cheat table pane to a *.ct* document file containing each address with its value type, description, display color, and display format. To load the saved *.ct* documents, go to **File ▸ Load**. (You'll find many ready-made cheat tables for popular games at *http://cheatengine.org/tables.php*.)

Now that I've described how to scan for a game state value, I'll discuss how you can change that value when you know where it lives in memory.

Memory Modification in Games

Bots cheat a game system by modifying memory values in the game's state in order to give you lots of in-game money, modify your character's health, change your character's position, and so on. In most online games, a character's vitals (such as health, mana, skills, and position) are held in memory but are controlled by the game server and relayed to your local game client over the Internet, so modifying such values during online play is merely cosmetic and doesn't affect the actual values. (Any useful memory modification to an online game requires a much more advanced hack that's beyond Cheat Engine's capabilities.) In local games with no remote server, however, you can manipulate all of these values at will.

Manual Modification with Cheat Engine

We'll use Cheat Engine to understand how the memory modification magic works.

To modify memory manually, do the following:

1. Attach Cheat Engine to a game.
2. Either scan for the address you wish to modify or load a cheat table that contains it.
3. Double-click on the Value column for the address to open an input prompt where you can enter a new value.
4. If you want to make sure the new value can't be overwritten, select the box under the Active column to *freeze* the address, which will make Cheat Engine keep writing the same value back to it every time it changes.

This method works wonders for quick-and-dirty hacks, but constantly changing values by hand is cumbersome; an automated solution would be much more appealing.

Trainer Generator

Cheat Engine's trainer generator allows you to automate the whole memory modification process without writing any code.

To create a *trainer* (a simple bot that binds memory modification actions to keyboard hotkeys), go to **File ▸ Create Generic Trainer Lua Script From Table**. This opens a Trainer Generator dialog similar to the one shown in Figure 1-2.

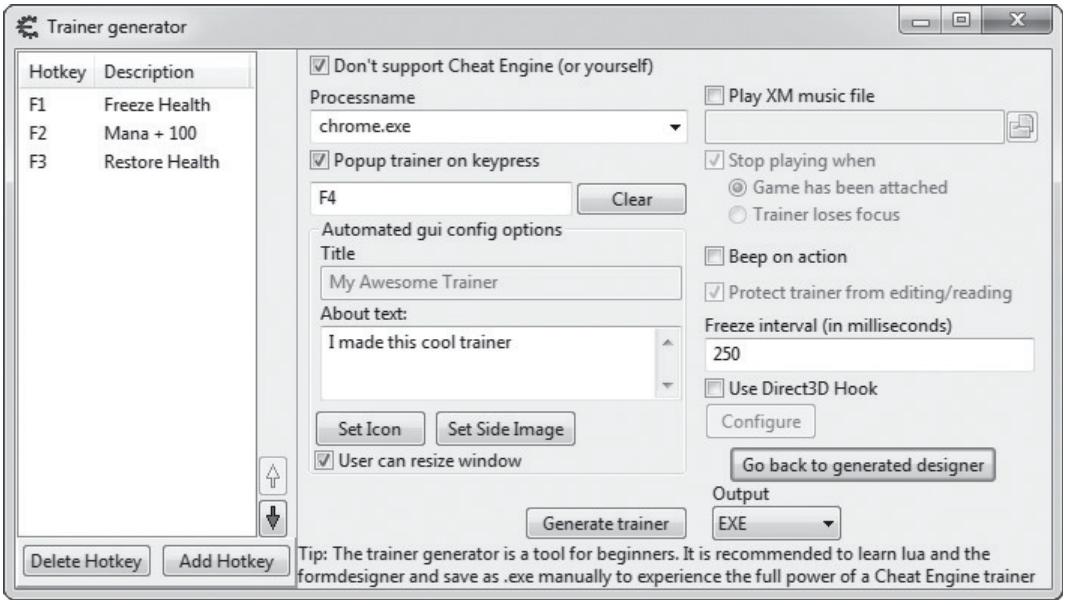


Figure 1-2: Cheat Engine Trainer Generator dialog

There are a number of fields to modify here:

Processname The name of the executable the trainer should attach to. This is the name shown in the process list when you attach with Cheat Engine, and it should be autofilled with the name of the process Cheat Engine is attached to.

Popup Trainer on Keypress Optionally enables a hotkey—which you set by entering a key combination in the box below the checkbox—to display the trainer's main window.

Title The name of your trainer, which will be displayed on its interface. This is optional.

About Text The description of your trainer, to be displayed on the interface; this is also optional.

Freeze Interval (in Milliseconds) The interval during which a freeze operation overwrites the value. You should generally leave this at 250, as lower intervals can sap resources and higher values may be too slow.

Once these values are configured, click **Add Hotkey** to set up a key sequence to activate your trainer. You will be prompted to select a value from your cheat table. Enter a value, and you will be taken to a Set/Change Hotkey screen similar to Figure 1-3.

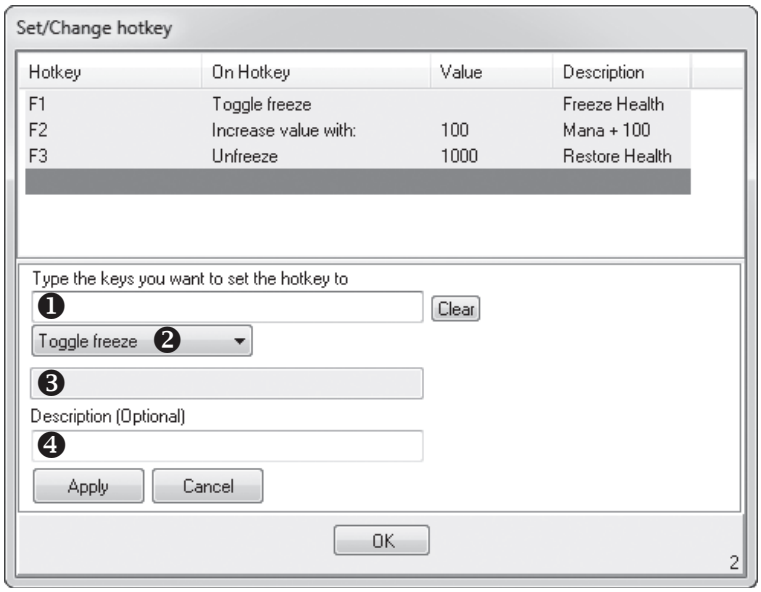


Figure 1-3: Cheat Engine Set/Change Hotkey screen

On this screen, place your cursor in the box labeled Type the Keys You Want to Set the Hotkey To ❶ and enter the desired key combination. Next, choose the desired action from the drop-down menu ❷; your options should appear in the following order:

Toggle Freeze Toggles the freeze state of the address.

Toggle Freeze and Allow Increase Toggles the freeze state of the address but allows the value to increase. Any time the value decreases, the trainer overwrites it with its previous value. Increased values will not be overwritten.

Toggle Freeze and Allow Decrease Does the opposite of Toggle Freeze and Allow Increase.

Freeze Sets the address to frozen if it's not frozen already.

Unfreeze Unfreezes the address if it's frozen.

Set Value To Sets the value to whatever you specify in the value box ❸.

Decrease Value With Decreases the value by the amount you specify in the value box ❸.

Increase Value With Does the opposite of Decrease Value With.

Finally, you can set a description for the action ❹. Click **Apply**, then **OK**, and your action will appear in the list on the Trainer Generator screen. At this point, Cheat Engine runs the trainer in the background, and you can simply press the hotkeys you configured to execute the memory actions.

To save your trainer to a portable executable, click **Generate Trainer**. Running this executable after the game is launched will attach your trainer to the game so you can use it without starting Cheat Engine.

Now that you know your way around Cheat Engine's memory scanner and trainer generator, try modifying some memory yourself in Lab 1-1.

LAB 1-1: BASIC MEMORY EDITING

Download the labs for this book from *http://www.nostarch.com/gamehacking/*. Open the *labs* folder and run the file *Lab01_01-BasicMemory.exe*. Next, start up Cheat Engine and attach to the lab binary. Then, using only Cheat Engine, find the addresses for the x- and y-coordinates of the gray ball.

HINT Use the *4 Byte* value type.

Once you've found the values, modify them to place the ball on top of the black square. The game will let you know once you've succeeded by displaying the text "Good job! You've completed the first lab!"

HINT Each time the ball is moved, its position (stored as a 4-byte integer) in that plane is changed by 1. Also, try to look only for static (green) results.

Pointer Scanning

As I've mentioned, online games often store values in dynamically allocated memory. While addresses that reference dynamic memory are useless to us in and of themselves, some static address will always point to another address, which in turn points to another, and so on, until the tail of the chain points to the dynamic memory we're interested in. Cheat Engine can locate these chains using a method called *pointer scanning*.

In this section, I'll introduce you to pointer chains and then describe how pointer scanning works in Cheat Engine. When you have a good grasp of the user interface, you can get some hands-on experience in Lab 1-2 on page 18.

Pointer Chains

The chain of offsets I've just described is called a *pointer chain* and looks like this:

```
list<int> chain = {start, offset1, offset2[, ...]}
```

The first value in this pointer chain (`start`) is called a *memory pointer*. It's an address that starts the chain. The remaining values (`offset1`, `offset2`, and so on) make up the route to the desired value, called a *pointer path*.

This pseudocode shows how a pointer chain might be read:

```
int readPointerChain(chain) {
❶     ret = read(chain[0])
    for i = 1, chain.len - 1, 1 {
        offset = chain[i]
        ret = read(ret + offset)
    }
    return ret
}
```

This code creates the function `readPointerPath()`, which takes a pointer chain called `chain` as a parameter. The function `readPointerPath()` treats the pointer path in `chain` as a list of memory offsets from the address `ret`, which is initially set to the memory pointer at ❶. It then loops through these offsets, updating the value of `ret` to the result of `read(ret + offset)` on each iteration and returning `ret` once it's finished. This pseudocode shows what `readPointerPath()` looks like when the loop is unrolled:

```
list<int> chain = {0xDEADBEEF, 0xAB, 0x10, 0xCC}
value = readPointerPath(chain)
// the function call unrolls to this
ret = read(0xDEADBEEF) //chain[0]
ret = read(ret + 0xAB)
ret = read(ret + 0x10)
ret = read(ret + 0xCC)
int value = ret
```

The function ultimately calls `read` four times, on four different addresses—one for each element in `chain`.

NOTE *Many game hackers prefer to code their chain reads in place, instead of encapsulating them in functions like* `readPointerPath()`.

Pointer Scanning Basics

Pointer chains exist because every chunk of dynamically allocated memory must have a corresponding static address that the game's code can use to reference it. Game hackers can access these chunks by locating the pointer chains that reference them. Because of their multitier structure, however, pointer chains cannot be located through the linear approach that memory scanners use, so game hackers have devised new ways to find them.

From a reverse-engineering perspective, you could locate and analyze the assembly code in order to deduce what pointer path it used to access the value, but doing so is very time-consuming and requires advanced tools. *Pointer scanners* solve this problem by using brute-force to recursively iterate over every possible pointer chain until they find one that resolves to the target memory address.

The Listing 1-1 pseudocode should give you a general idea of how a pointer scanner works.

```
list<int> pointerScan(target, maxAdd, maxDepth) {
❶  for address = BASE, 0x7FFFFF, 4 {
       ret = rScan(address, target, maxAdd, maxDepth, 1)
       if (ret.len > 0) {
           ret.pushFront(address)
           return ret
       }
   }
   return {}
}
list<int> rScan(address, target, maxAdd, maxDepth, curDepth) {
❷  for offset = 0, maxAdd, 4 {
       value = read(address + offset)
❸    if (value == target)
           return list<int>(offset)
   }
❹  if (curDepth < maxDepth) {
       curDepth++
❺    for offset = 0, maxAdd, 4 {
           ret = rScan(address + offset, target, maxAdd, maxDepth, curDepth)
❻        if (ret.len > 0) {
             ret.pushFront(offset)
❼            return ret
           }
       }
   }
   return {}
}
```

Listing 1-1: Pseudocode for a pointer scanner

This code creates the functions `pointerScan()` and `rScan()`.

pointerScan()

The `pointerScan()` function is the entry point to the scan. It takes the parameters `target` (the dynamic memory address to find), `maxAdd` (the maximum value of any offset), and `maxDepth` (the maximum length of the pointer path). It then loops through every 4-byte aligned address ❶ in the game, calling `rScan()` with the parameters `address` (the address in the current iteration), `target`, `maxAdd`, `maxDepth`, and `curDepth` (the depth of the path, which is always 1 in this case).

rScan()

The `rScan()` function reads memory from every 4-byte aligned offset between `0` and `maxAdd` ❷, and returns if a result is equal to `target` ❸. If `rScan()` doesn't return in the first loop and the recursion is not too deep ❹, it increments `curDepth` and again loops over each offset ❺, calling itself for each iteration.

If a self call returns a partial pointer path ❻, `rScan()` will prepend the current offset to the path and return up the recursion chain ❼ until it

reaches `pointerScan()`. When a call to `rScan()` from `pointerScan()` returns a pointer path, `pointerScan()` pushes the current address to the front of the path and returns it as a complete chain.

Pointer Scanning with Cheat Engine

The previous example showed the basic process of pointer scanning, but the implementation I've shown is primitive. Aside from being insanely slow to execute, it would generate countless false positives. Cheat Engine's pointer scanner uses a number of advanced interpolations to speed up the scan and make it more accurate, and in this section, I'll introduce you to the smorgasbord of available scanning options.

To initiate a pointer scan in Cheat Engine, right-click on a dynamic memory address in your cheat table and click **Pointer Scan For This Address**. When you initiate a pointer scan, Cheat Engine will ask you where to store the scan results as a *.ptr* file. Once you enter a location, a Pointerscanner Scanoptions dialog similar to the one shown in Figure 1-4 will appear.

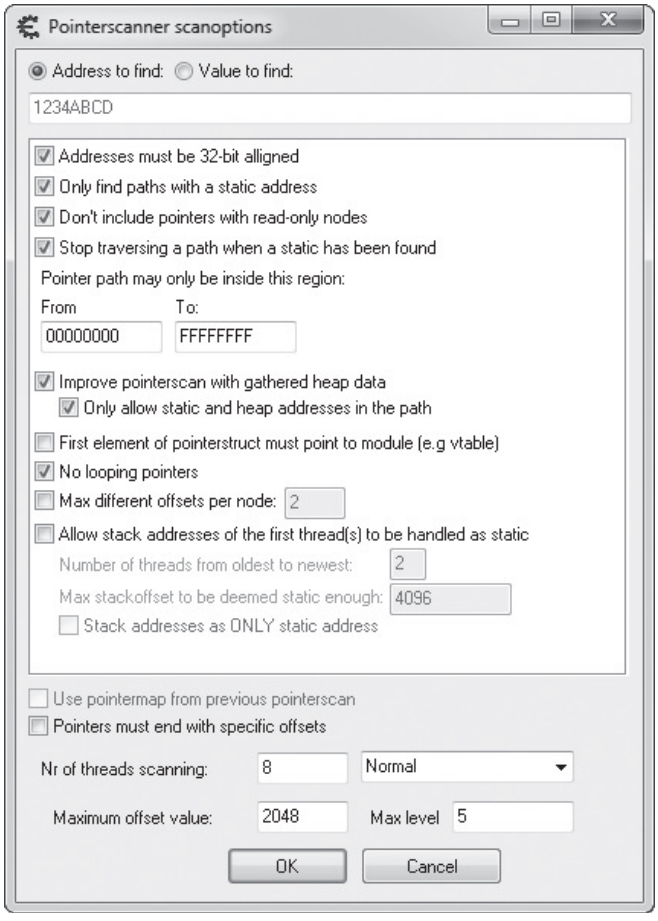


Figure 1-4: Cheat Engine Pointerscanner Scanoptions dialog

The Address to Find input field at the top displays your dynamic memory address. Now carefully select from among Cheat Engine's many scan options.

Key Options

Several of Cheat Engine's scan options typically retain their default values. Those options are as follows:

Addresses Must Be 32-Bit Aligned Tells Cheat Engine to scan only addresses that are multiples of 4, which greatly increases the scan speed. As you'll learn in Chapter 4, compilers align data so that most addresses will be multiples of 4 anyway by default. You'll rarely need to disable this option.

Only Find Paths with a Static Address Speeds up the scan by preventing Cheat Engine from searching paths with a dynamic start pointer. This option should *always* be enabled because scanning for a path starting at another dynamic address can be counterproductive.

Don't Include Pointers with Read-Only Nodes Should also always be enabled. Dynamically allocated memory that stores volatile data should never be read-only.

Stop Traversing a Path When a Static Has Been Found Terminates the scan when it finds a pointer path with a static start address. This should be enabled to reduce false positives and speed up the scan.

Pointer Path May Only Be Inside This Region Can typically be left as is. The other options available to you compensate for this large range by intelligently narrowing the scope of the scan.

First Element of Pointerstruct Must Point to Module Tells Cheat Engine not to search heap chunks in which virtual function tables are not found, under the assumption that the game was coded using object orientation. While this setting can immensely speed up scans, it's highly unreliable and you should almost always leave it disabled.

No Looping Pointers Invalidates any paths that point to themselves, weeding out inefficient paths but slightly slowing down the scan. This should usually be enabled.

Nr of Threads Scanning Determines how many threads the scanner will use. A number equal to the number of cores in your processor works best. The accompanying drop-down specifies the priority each thread will have; Normal should be fine.

Max Level Determines the maximum length of the pointer path. (Remember the `maxDepth` variable in the example code in Listing 1-1?) This should be kept around 6 or 7.

Of course, there will be times when you'll need to change these options from the settings described. For example, failing to obtain reliable results with the No Looping Pointers or Max Level settings typically means that the value you're looking for exists in a dynamic data structure, like a linked list, binary tree, or vector. Another example is the Stop Traversing a Path When a Static Has Been Found option, which in rare cases can prevent you from getting reliable results.

Situational Options

Unlike the previous options, your settings for the remaining ones will depend on your situation. Here's how to determine the best configuration for each:

Improve Pointerscan with Gathered Heap Data Allows Cheat Engine to use the heap allocation record to determine offset limits, effectively speeding up the scan by weeding out many false positives. If you run into a game using a custom memory allocator (which is becoming increasingly common), this option can actually do the exact opposite of what it's meant to do. You can leave this setting enabled in initial scans, but it should be the first to go when you're unable to find reliable paths.

Only Allow Static and Heap Addresses in the Path Invalidates all paths that can't be optimized with heap data, making this approach even more aggressive.

Max Different Offsets per Node Limits the number of same-value pointers the scanner checks. That is, if *n* different addresses point to `0x0BADF00D`, this option tells Cheat Engine to consider only the first *m* addresses. This can be extremely helpful when you're unable to narrow down your result set. In other cases, you may want to disable it, as it will miss many valid paths.

Allow Stack Addresses of the First Thread(s) to Be Handled as Static Scans the call stacks of oldest *m* threads in the game, considering the first *n* bytes in each one. This allows Cheat Engine to scan the parameters and local variables of functions in the game's call chain (the goal being to find variables used by the game's main loop). The paths found with this option can be both highly volatile and extremely useful; I use it only when I fail to find heap addresses.

Stack Addresses as Only Static Address Takes the previous option even further by allowing only stack addresses in pointer paths.

Pointers Must End with Specific Offsets Can be useful if you know the offset(s) at the end of a valid path. This option will allow you to specify those offsets (starting with the last offset at the top), greatly reducing the scope of the scan.

Nr of Threads Scanning Determines how many threads the scanner will use. A number equal to the number of cores in your processor often works best. A drop-down menu with options allows you to specify the priority for each thread. Idle is best if you want your scan to go very slowly, Normal is what you should use for most scans, and Time Critical is useful for lengthy scans but will render your computer useless for the scan duration.

Maximum Offset Value Determines the maximum value of each offset in the path. (Remember the `maxAdd` variable in Listing 1-1?) I typically start with a low value, increasing it only if my scan fails; `128` is a good starting value. Keep in mind that this value is mostly ignored if you're using the heap optimization options.

NOTE *What if both Only Allow Static and Heap Addresses in the Path and Stack Addresses as Only Static Address are enabled? Will the scan come up empty? Seems like a fun, albeit useless, experiment.*

Once you have defined your scan options, click **OK** to start a pointer scan. When the scan completes, a results window will appear with the list of pointer chains found. This list often has thousands of results, containing both real chains and false positives.

Pointer Rescanning

The pointer scanner has a rescan feature that can help you eliminate false positives. To begin, press CTRL-R from the results window to open the Rescan Pointerlist window, as shown in Figure 1-5.

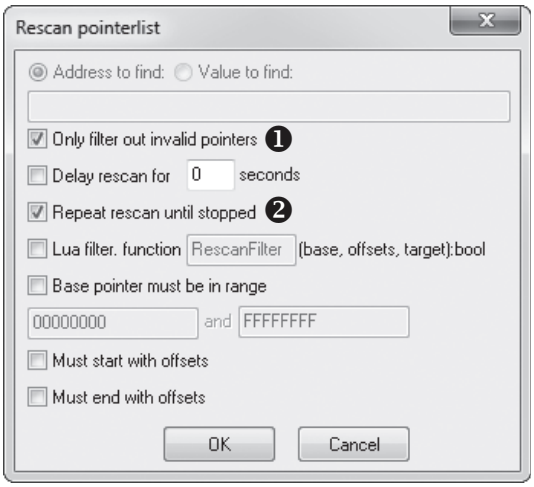


Figure 1-5: Cheat Engine Rescan Pointerlist window

There are two main options to consider when you tell Cheat Engine to rescan:

Only Filter Out Invalid Pointers If you check this box ❶, the rescan will discard only pointer chains that point to invalid memory, which helps if your initial result set is very large. Disable this to filter out paths that don't resolve to a specific address or value (as shown in the figure).

Repeat Rescan Until Stopped If you check this box ❷, the rescan will execute in a continuous loop. Ideally, you should enable this setting and let rescan run while you create a large amount of memory entropy.

For the initial rescan, enable both **Only Filter Out Invalid Pointers** and **Repeat Rescan Until Stopped**, and then press **OK** to initiate the rescan. The rescan window will go away, and a Stop Rescan Loop button will appear in the results window. The result list will be constantly rescanned until you click Stop Rescan Loop, but spend a few minutes creating memory entropy before doing so.

In rare cases, rescanning using a rescan loop may still leave you with a large list of possible paths. When this happens, you may need to restart the game, find the address that holds your value (it may have changed!), and use the rescan feature on this address to further narrow results. In this scan, leave **Only Filter Out Invalid Pointers** unchecked and enter the *new* address in the **Address to Find** field.

NOTE *If you had to close the results window, you can reopen it and load the result list by going to the main Cheat Engine window and pressing the Memory View button below the results pane. This should bring up a memory dump window. When the window appears, press* CTRL-*P to open the pointer scan results list,. Then press* CTRL-*O to open the* .ptr *file where you saved the pointer scan.*

If your results still aren't narrow enough, try running the same scan across system restarts or even on different systems. If this still yields a large result set, each result can safely be considered static because more than one pointer chain can resolve to the same address.

Once you've narrowed down your result set, double-click on a usable pointer chain to add it to your cheat table. If you have a handful of seemingly usable chains, grab the one with the fewest offsets. If you find multiple chains with identical offsets that start with the same pointer but diverge after a certain point, your data may be stored in a dynamic data structure.

That's all there is to pointer scanning in Cheat Engine. Try it yourself in Lab 1-2!

LAB 1-2: POINTER SCANNING

From the same directory you used for Lab 1-1, open *Lab01_01-MemoryPointers.exe*. Unlike the last task, which required you to win only once, this lab requires that you win 50 times in 10 seconds. Upon each win, the memory addresses for the x- and y-coordinates will change, meaning you will be able to freeze the value only if you have found a proper pointer path. Start this lab the same way as the previous one, but once you've found the addresses, use the Pointer Scan feature to locate pointer paths to them. Then, place the ball on top of the black square, freeze the value in place, and press TAB to begin the test. Just as before, the game will let you know once you've won.

HINT Try setting the maximum level to 5 and the maximum offset value to 512. Also, play with the options to allow stack addresses, terminate the scan when a static is found, and improve the pointer scan with heap data. See which combination of options gives the best results.

Lua Scripting Environment

Historically, bot developers rarely used Cheat Engine to update their addresses when a game released a patch because it was much easier to do

so in OllyDbg. This made Cheat Engine useless to game hackers other than for initial research and development—that is, until a powerful Lua-based embedded scripting engine was implemented around Cheat Engine's robust scanning environment. While this engine was created to enable the development of simple bots within Cheat Engine, professional game hackers found they could also use it to easily write complex scripts to automatically locate addresses across different versions of a game's binary—a task that might otherwise take hours.

NOTE *You'll find more detail about the Cheat Engine Lua scripting engine on the wiki at* http://wiki.cheatengine.org/.

To start using the Lua engine, press CTRL-ALT-L from the main Cheat Engine window. Once the window opens, write your script in the text area and click **Execute Script** to run it. Save a script with CTRL-S and open a saved script with CTRL-O.

The scripting engine has hundreds of functions and infinite use cases, so I'll give you just a glimpse of its abilities by breaking down two scripts that I actively use in development. These scripts are tailored to my personal toolkit and, in their current form, are useful only for demonstration purposes. This is true of nearly all scripts for Cheat Engine, since every game is different and every game hacker writes scripts to accomplish unique goals.

Searching for Assembly Patterns

This first script locates functions that compose outgoing packets and sends them to the game server. It works by searching a game's assembly code for functions that contain a certain code sequence.

```
❶ BASEADDRESS = getAddress("Game.exe")
❷ function LocatePacketCreation(packetType)
❸  for address = BASEADDRESS, (BASEADDRESS + 0x2ffffff) do
         local push = readBytes(address, 1, false)
         local type = readInteger(address + 1)
         local call = readInteger(address + 5)
❹       if (push == 0x68 and type == packetType and call == 0xE8) then
             return address
         end
     end
     return 0
 end
 FUNCTIONHEADER = { 0xCC, 0x55, 0x8B, 0xEC, 0x6A }
❺ function LocateFunctionHead(checkAddress)
     if (checkAddress == 0) then return 0 end
❻  for address = checkAddress, (checkAddress - 0x1fff), -1 do
         local match = true
         local checkheader = readBytes(address, #FUNCTIONHEADER, true)
❼       for i, v in ipairs(FUNCTIONHEADER) do
             if (v ~= checkheader[i]) then
                 match = false
                 break
```

```
            end
         end
❽      if (match) then return address + 1 end
      end
      return 0
   end

❾ local funcAddress = LocateFunctionHead(LocatePacketCreation(0x64))
   if (funcAddress ~= 0) then
      print(string.format("0x%x",funcAddress))
   else
      print("Not found!")
   end
```

The code begins by getting the base address of the module that Cheat Engine is attached to ❶. Once it has the base address, I define the function `LocatePacketCreation()` ❷, which loops through the first `0x2FFFFFF` bytes of memory in the game ❸, searching for a sequence that represents this x86 assembler code:

```
PUSH type   ; Data is: 0x68 [4byte type]
CALL offset ; Data is: 0xE8 [4byte offset]
```

The function checks that the type is equal to `packetType`, but it doesn't care what the function offset is ❹. Once this sequence is found, the function returns.

Next, I define a function `LocateFunctionHead()` ❺, which backtracks up to 0x1FFF bytes from a given address ❻. At each address, it checks for a stub of assembler code ❼ that looks something like this:

```
INT3          ; 0xCC
PUSH EBP      ; 0x55
MOV EBP, ESP  ; 0x8B 0xEC
PUSH [-1]     ; 0x6A 0xFF
```

This stub will be present at the beginning of every function, because it's part of the function prologue that sets up the function's stack frame. Once it finds the code, the function will return the address of the stub plus 1 ❽ (the first byte, `0xCC`, is padding).

To tie these steps together, I call the `LocatePacketCreation()` function with the `packetType` that I'm looking for (arbitrarily `0x64`) and pass the resulting address into the `LocateFunctionHead()` function ❾. This effectively locates the first function that pushes `packetType` into a function call and stores its address in `funcAddress`. This stub shows the result:

```
INT3          ; LocateFunctionHead back-tracked to here
PUSH EBP      ;   and returned this address
MOV EBP, ESP
PUSH [-1]
{...}         ; unrelated function code
```

```
PUSH [0x64]   ; LocatePacketCreation returned this address
CALL [something]
```

In development, I use this 35-line script to automatically locate 15 different functions in under a minute. It would take me about 15 minutes to do that with OllyDbg.

Searching for Strings

This next Lua script scans a game's memory for text strings. It works much as the Cheat Engine's memory scanner does when you use the string value type.

```
BASEADDRESS = getAddress("Game.exe")
❶ function findString(str)
    local len = string.len(str)
❷  local chunkSize = 4096
❸  local chunkStep = chunkSize - len
    print("Found '" .. str .. "' at:")
❹  for address = BASEADDRESS, (BASEADDRESS + 0x2ffffff), chunkStep  do
        local chunk = readBytes(address, chunkSize, true)
        if (not chunk) then break end
❺      for c = 0, chunkSize-len do
❻          checkForString(address , chunk, c, str, len)
        end
    end
end
function checkForString(address, chunk, start, str, len)
    for i = 1, len do
        if (chunk[start+i] ~= string.byte(str, i)) then
            return false
        end
    end
❼  print(string.format("\t0x%x", address + start))
end

❽ findString("hello")
❾ findString("world")
```

After getting the base address, I define the function `findString()` ❶, which takes a string, `str`, as a parameter. This function loops through the game's memory ❹ in 4,096-byte-long chunks ❷. The chunks are scanned sequentially, each one starting `len` (the length of `str`) bytes before the end of the previous one ❸ to prevent missing a string that begins on one chunk and ends on another.

As `findString()` reads each chunk, it iterates over every byte until the overlap point in the chunk ❺, passing each subchunk into the `checkForString()` function ❻. If `checkForString()` matches the subchunk to `str`, it prints the address of that subchunk to the console ❼.

Lastly, to find all addresses that reference the strings `"hello"` and `"world"`, I call `findString("hello")` ❽ and `findString("world")` ❾. By using this code to

search for embedded debug strings and pairing it with the previous code to locate function headers, I'm able to find a large number of internal functions within a game in mere seconds.

OPTIMIZING MEMORY CODE

Due to the high overhead of memory reading, optimization is extremely important when you're writing code that performs memory reads. In the previous code snippet, notice that the function `findString()` does not use the Lua engine's built-in `readString()` function. Instead, it reads big chunks of memory and searches them for the desired string. Let's break down the numbers.

A scan using `readString()` would try to read a string of `len` bytes at every possible memory address. This means it would read, at most, `(0x2FFFFFF * len + len)` bytes. However, `findString()` reads chunks of 4,096 bytes and scans them locally for matching strings. This means it would read, at most, `(0x2FFFFFF + 4096 + (0x2FFFFFF / (4096 - 10)) * len)` bytes. When searching for a string with a length of 10, the number of bytes that each method would read is 503,316,480 and 50,458,923, respectively.

Not only does `findString()` read an order of magnitude less data, it also invokes far fewer memory reads. Reading in chunks of 4,096 bytes would require a total of `(0x2FFFFFF / (4096 - len))` reads. Compare that to a scan using `readString()`, which would need `0x2FFFFFF` reads. The scan that uses `findString()` is a huge improvement because invoking a read is much more expensive than increasing the size of data being read. (Note that I chose 4,096 arbitrarily. I keep the chunk relatively small because reading memory can be time-consuming, and it might be wasteful to read four pages at a time just to find the string in the first.)

Closing Thoughts

By this point, you should have a basic understanding of Cheat Engine and how it works. Cheat Engine is a very important tool in your kit, and I encourage you to get some hands-on experience with it by following Labs 1-1 and 1-2 on pages 11 and 18 and playing around with it on your own.

2

DEBUGGING GAMES WITH OLLYDBG

You can scratch the surface of what happens as a game runs with Cheat Engine, but with a good debugger, you can dig deeper until you understand the game's structure and execution flow. That makes OllyDbg essential to your game-hacking arsenal. It's packed with a myriad of powerful tools like conditional breakpoints, referenced string search, assembly pattern search, and execution tracing, making it a robust assembler-level debugger for 32-bit Windows applications.

I'll cover low-level code structure in detail in Chapter 4, but for this chapter, I assume you're at least familiar with modern code-level debuggers, such as the one packaged with Microsoft Visual Studio. OllyDbg is functionally similar to those, with one major difference: it interfaces with the assembly code of an application, working even in the absence of source code and/

or debug symbols, making it ideal when you need to dig into the internals of a game. After all, game companies are rarely nice (or dumb) enough to ship their games with debug symbols!

In this chapter, I'll go over OllyDbg's user interface, show you how to use its most common debugging features, break down its expression engine, and provide some real-world examples of how you can tie it in to your game hacking endeavors. As a wrap-up, I'll teach you about some useful plug-ins and send you off with a lab designed to get you started in OllyDbg.

NOTE *This chapter focuses on OllyDbg 1.10 and may not be entirely accurate for later versions. I use this version because, at the time of writing, OllyDbg 2 is still under development and its plug-in interface is less robust.*

When you feel like you have a handle on OllyDbg's interface and features, you can try it on a game yourself with Lab 2-1 on page 46.

A Brief Look at OllyDbg's User Interface

Go to the OllyDbg website (*http://www.ollydbg.de/*), download and install OllyDbg, and open the program. You should see the toolbar shown in Figure 2-1 above a multiple window interface area.

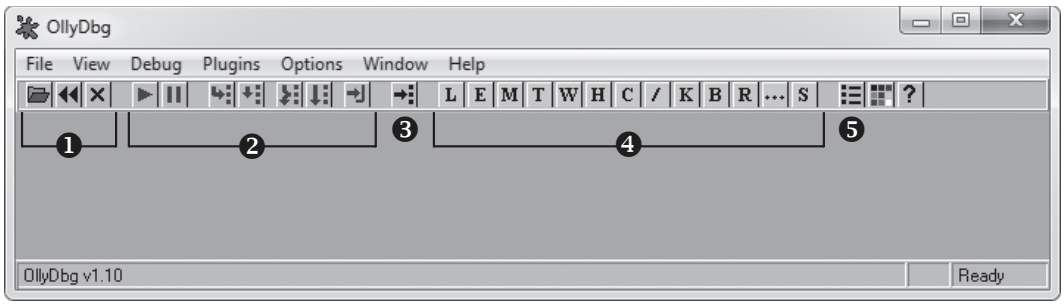


Figure 2-1: OllyDbg main window

This toolbar contains the program controls ❶, the debug buttons ❷, the Go To button ❸, the control window buttons ❹, and the Settings button ❺.

The three program controls allow you to open an executable and attach to the process it creates, restart the current process, or terminate execution of the current process, respectively. You can also complete these functions with the hotkeys F3, CTRL-F2, and ALT-F2, respectively. To attach to a process that is already running, click **File ▸ Attach**.

The debug buttons control the debugger actions. Table 2-1 describes what these buttons do in order, along with their hotkeys and functions. This table also lists three useful debugger actions that don't have buttons on the debug toolbar.

Table 2-1: Debug Buttons and Other Debugger Functions

Button	Hotkey	Function
Play	F9	Resumes normal execution of the process.
Pause	F12	Pauses execution of all threads within the process and brings up the CPU window at the instruction currently being executed.
Step Into	F7	Single-steps to the next operation to be executed (will dive down into function calls).
Step Over	F8	Steps to the next operation to be executed within current scope (will skip over function calls).
Trace Into	CTRL-F11	Runs a deep trace, tracing every operation that is executed.
Trace Over	CTRL-F12	Runs a passive trace that traces only operations within the current scope.
Execute Until Return	CTRL-F9	Executes until a return operation is hit within the current scope.
	CTRL-F7	Automatically single-steps on every operation, following execution in the disassembly window. This makes execution appear to be animated.
	CTRL-F8	Also animates execution, but steps over functions instead of stepping into them.
	ESC	Stops animation, pausing execution on the current operation.

The Go To button opens a dialog asking for a hexadecimal address. Once you enter the address, OllyDbg opens the CPU window and shows the disassembly at the specified address. When the CPU window is in focus, you can also show that information with the hotkey CTRL-G.

The control window buttons open different *control windows*, which display useful information about the process you're debugging and expose more debugging functions, like the ability to set breakpoints. OllyDbg has a total of 13 control windows, which can all be open simultaneously within the multiple window interface. Table 2-2 describes these windows, listed in the order in which they appear on the window buttons toolbar.

Table 2-2: OllyDbg's Control Windows

Window	Hotkey	Function
Log	ALT-L	Displays a list of log messages, including debug prints, thread events, debugger events, module loads, and much more.
Modules	ALT-E	Displays a list of all executable modules loaded into the process. Double-click a module to open it in the CPU window.

Table 2-2 (continued)

Memory Map	ALT-M	Displays a list of all blocks of memory allocated by the process. Double-click a block in the list to bring up a dump window of that memory block.
Threads		Displays a list of threads running in the process. For each thread in this list, the process has a structure called a *Thread Information Block (TIB)*. OllyDbg allows you to view each thread's TIB; simply right-click a thread and select **Dump Thread Data Block**.
Windows		Displays a list of window handles held by the process. Right-click a window in this list to jump to or set a breakpoint on its class procedure (the function that gets called when a message is sent to the window).
Handles		Displays a list of handles held by the process. (Note that Process Explorer has a much better handle list than OllyDbg, as I will discuss in Chapter 3.)
CPU	ALT-C	Displays the main disassembler interface and controls a majority of the debugger functionality.
Patches	CTRL-P	Displays a list of any assembly code modifications you have made to modules within the process.
Call Stack	ALT-K	Displays the call stack for the active thread. The window updates when the process halts.
Breakpoints	ALT-B	Displays a list of active debugger breakpoints and allows you to toggle them on and off.
References		Displays the reference list, which typically holds the search results for many different types of searches. It pops up on its own when you run a search.
Run Trace		Displays a list of operations logged by a debugger trace.
Source		Displays the source code of the disassembled module if a program debug database is present.

Finally, the Settings button opens the OllyDbg settings window. Keep the default settings for now.

Now that you've had a tour of the main OllyDbg window, let's explore the CPU, Patches, and Run Trace windows more closely. You'll use those windows extensively as a game hacker, and knowing your way around them is key.

OllyDbg's CPU Window

The CPU window in Figure 2-2 is where game hackers spend most of their time in OllyDbg because it is the main control window for the debugging features.

This window houses four distinct control panes: the disassembler pane ❶, the registers pane ❷, the dump pane ❸, and the stack pane ❹. These four panes encapsulate OllyDbg's main debugger functions, so it's important to know them inside and out.

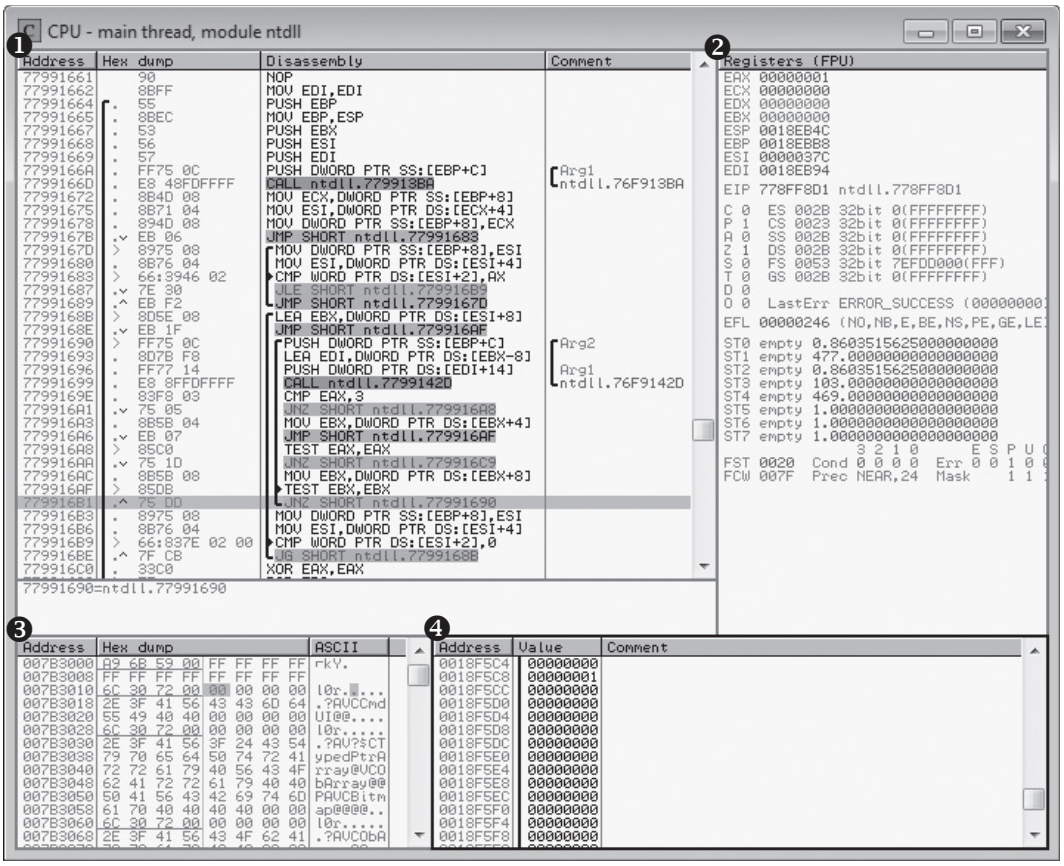


Figure 2-2: OllyDbg CPU window

Viewing and Navigating a Game's Assembly Code

You'll navigate game code and control most aspects of debugging from OllyDbg's disassembler pane. This pane displays the assembly code for the current module, and its data is neatly displayed in a table composed of four distinct columns: Address, Hex dump, Disassembly, and Comment.

The Address column displays the memory addresses of each operation in the game process you're attached to. You can double-click an address in this column to toggle whether it's the *display base*. When an address is set as the display base, the Address column displays all other addresses as offsets relative to it.

The Hex dump column displays the byte code for each operation, grouping operation codes and parameters accordingly. Black braces spanning multiple lines on the left side of this column mark known function boundaries. Operations that have jumps going to them are shown with a right-facing arrow on the inside of these braces. Operations that perform jumps are shown with either up-facing or down-facing arrows, depending on the direction in which they jump, on the inside of these braces. For

example, in Figure 2-2, the instruction at address 0x779916B1 (highlighted in gray) has an up-facing arrow, indicating it's an upward jump. You can think of a jump as a `goto` operator.

The Disassembly column displays the assembly code of each operation the game performs. So, for example, you can confirm that the instruction at 0x779916B1 in Figure 2-2 is a jump by looking at the assembly, which shows a JNZ (jump if nonzero) instruction. Black braces in this column mark the boundaries of loops. Right-facing arrows attached to these braces point to the conditional statements that control whether the loops continue or exit. The three right-facing arrows in this column in Figure 2-2 point to CMP (compare) and TEST instructions, which are used by assembly code to compare values.

The Comment column displays human-readable comments about each operation the game performs. If OllyDbg encounters known API function names, it will automatically insert a comment with the name of the function. Similarly, if it successfully detects arguments being passed to a function, it will label them (for example, `Arg1`, `Arg2`, . . . , `ArgN`). You can double-click in this column to add a customized comment. Black braces in this column mark the assumed boundaries of function call parameters.

NOTE *OllyDbg infers function boundaries, jump directions, loop structures, and function parameters during code analysis, so if these columns lack boundary lines or jump arrows, just press CTRL-A to run a code analysis on the binary.*

When the disassembler pane is in focus, there are a few hotkeys you can use to quickly navigate code and control the debugger. Use F2 for Toggle Breakpoint, SHIFT-F12 for Place Conditional Breakpoint, - (hyphen) for Go Back and + (plus) for Go Forward (these two work as you'd expect in a web browser), * (asterisk) for Go to EIP (which is the execution pointer in the x86 architecture), CTRL- - for Go to Previous Function, and CTRL-+ for Go to Next Function.

The disassembler can also populate the References window with different types of search results. When you want to change the References window's contents, right-click in the disassembler pane, mouse over the Search For menu to expand it, and select one of the following options:

All Intermodular Calls Searches for all calls to functions in remote modules. This can, for example, allow you to see everywhere that a game calls `Sleep()`, `PeekMessage()`, or any other Windows API function, enabling you to inspect or set breakpoints on the calls.

All Commands Searches for all occurrences of a given operation written in assembly, where the added operators `CONST` and `R32` will match a constant value or a register value, respectively. One use for this option might be searching for commands like `MOV [0xDEADBEEF], CONST`; `MOV [0xDEADBEEF], R32`; and `MOV [0xDEADBEEF], [R32+CONST]` to list all operations that modify memory at the address `0xDEADBEEF`, which could be anything, including the address of your player's health.

All Sequences Searches for all occurrences of a given sequence of operations. This is similar to the previous options, but it allows you to specify multiple commands.

All Constants Searches for all instances of a given hexadecimal constant. For instance, if you enter the address of your character's health, this will list all of the commands that directly access it.

All Switches Searches for all switch-case blocks.

All Referenced Text Strings Searches for all strings referenced in code. You can use this option to search through all referenced strings and see what code accesses them, which can be useful for correlating in-game text displays with the code that displays them. This option is also very useful for locating any debug assertion or logging strings, which can be a tremendous help in determining the purpose of code parts.

The disassembler can also populate the Names window with all labels in the current module (CTRL-N) or all known labels in all modules (Search For ▸ Name in All Modules). Known API functions will be automatically labeled with their names, and you can add a label to a command by highlighting it, pressing :, and entering the label when prompted. When a labeled command is referenced in code, the label will be shown in place of the address. One way to use this feature is to name functions that you've analyzed (just set a label on the first command in a function) so you can see their names when other functions call them.

Viewing and Editing Register Contents

The registers pane displays the contents of the eight processor registers, all eight flag bits, the six segment registers, the last Windows error code, and EIP. Underneath these values, this pane can display either *Floating-Point Unit (FPU)* registers or debug registers; click on the pane's header to change which registers are displayed. The values in this pane are populated only if you freeze your process. Values that are displayed in red have been changed since the previous pause. Double-click on values in this pane to edit them.

Viewing and Searching a Game's Memory

The dump pane displays a dump of the memory at a specific address. To jump to an address and display the memory contents, press CTRL-G and enter the address in the box that appears. You can also jump to the address of an entry in the other CPU window panes by right-clicking on the Address column and selecting Follow in Dump.

While there are always three columns in the dump pane, the only one you should always see is the Address column, which behaves much like its cousin within the disassembler pane. The data display type you choose determines the other two columns shown. Right-click the dump pane to change the display type; for the one shown in Figure 2-2, you'd right-click and select Hex ▸ Hex/ASCII (8 bytes).

You can set a memory breakpoint on an address shown in the dump pane by right-clicking that address and expanding the Breakpoint

submenu. Select **Memory ▸ On Access** from this menu to break on any code that uses the address at all, or select **Memory ▸ On Write** to break only on code that writes to that space in memory. To remove a memory breakpoint, select **Remove Memory Breakpoint** in the same menu; this option appears only when the address you right-click has a breakpoint.

With one or more values selected in the dump, you can press CTRL-R to search the current module's code for references to addresses of the selected values; results of this search appear in the References window. You can also search for values in this pane using CTRL-B for binary strings and CTRL-N for labels. After you initiate a search, press CTRL-L to jump to the next match. CTRL-E allows you to edit any values you have selected.

NOTE *The dump windows that you can open from the Memory window work in the same way as the dump pane.*

Viewing a Game's Call Stack

The final CPU pane is the stack pane, and as the name suggests, it shows the call stack. Like the dump and disassembler panes, the stack pane has an Address column. The stack pane also has a Value column, which shows the stack as an array of 32-bit integers, and a Comment column, which shows return addresses, known function names, and other informative labels. The stack pane supports all the same hotkeys as the dump pane, with the exception of CTRL-N.

Creating Code Patches

OllyDbg's *code patches* let you make assembly code modifications for a game you want to hack, removing the need to engineer a tool tailored to that specific game. This makes prototyping *control flow hacks*—which manipulate game behavior through a mix of game design flaws, x86 assembly protocols, and common binary constructs—much easier.

MULTICLIENT PATCHING

One type of hack, called a *multiclient patch*, overwrites the single-instance limitation code within a game's binary with no-operation code, allowing the user to run multiple game clients, even when doing so is normally forbidden. Because the code that performs instance limitation must be executed very early after a game client is launched, it can be nearly impossible for a bot to inject its patch on time. The easiest workaround for this is to make multiclient patches persist by applying them within OllyDbg and saving them directly to the game binary.

Game hackers typically include perfected patches as optional features in a bot's tool suite, but in some cases, making those features persistent is actually more convenient for your end user. Luckily, OllyDbg patches provide the complete functionality you need to design, test, and permanently save code modifications to an executable binary using only OllyDbg.

To place a patch, navigate to the line of assembly code you want to patch in the CPU window, double-click the instruction you wish to modify, place a new assembly instruction in the pop-up prompt, and click **Assemble**, as shown in Figure 2-3.

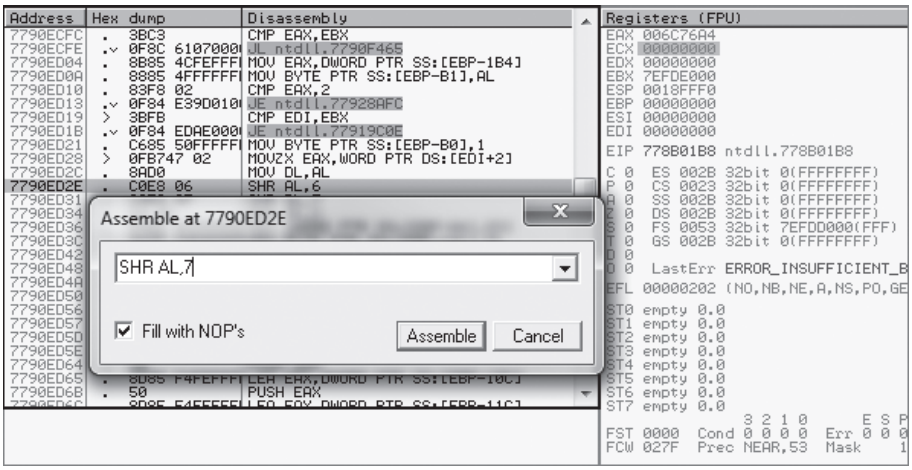


Figure 2-3: Placing a patch with OllyDbg

Always pay attention to the size of your patch—you can't just resize and move around assembled code however you'd like. Patches *larger* than the code you intend to replace will overflow into subsequent operations, potentially removing critical functionality. Patches *smaller* than the operations you intend to replace are safe, as long as Fill with NOPs is checked. This option fills any abandoned bytes with *no-operation (NOP)* commands, which are single-byte operations that do nothing when executed.

All patches you place are listed, along with the address, size, state, old code, new code, and comment, in the Patches window. Select a patch in this list to access a small but powerful set of hotkeys, shown in Table 2-3.

Table 2-3: Patches Window Hotkeys

Operator	Function
ENTER	Jumps to the patch in the disassembler.
SPACE	Toggles the patch on or off.
F2	Places a breakpoint on the patch.
SHIFT-F2	Places a conditional breakpoint on the patch.
SHIFT-F4	Places a conditional log breakpoint on the patch.
DEL	Removes the patch entry from the list only.

DETERMINING PATCH SIZE

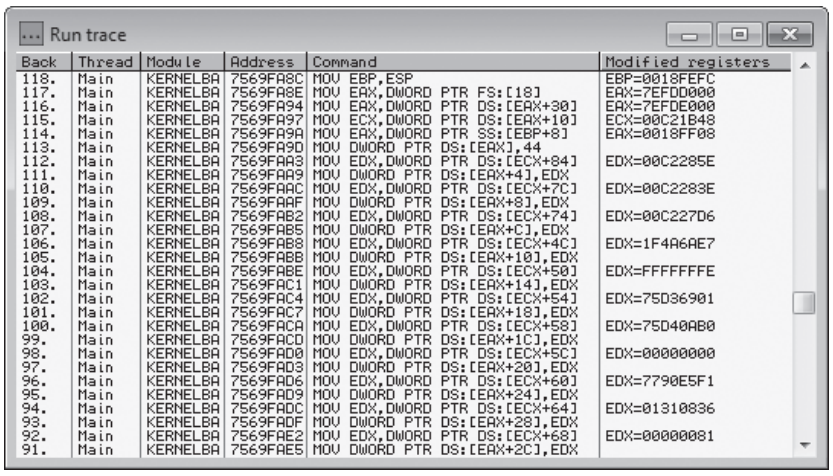
There are a few ways to determine whether your patch will be a different size than the original code. For example, in Figure 2-3, you can see the command at `0x7790ED2E` being changed from `SHR AL, 6` to `SHR AL, 7`. If you look at the bytes to the left of the command, you see 3 bytes that represent the memory of the command. This means our new command must either be 3 bytes or padded with NOPs if it's less than 3 bytes. Furthermore, these bytes are arranged in two columns. The first column contains `0xC0` and `0x08`, which represent the command `SHR` and the first operand, `AL`. The second column contains `0x06`, which represents the original operand. Because the second column shows a single byte, any replacement operand must also be 1 byte (between `0x00` and `0xFF`). If this column had shown `0x00000006` instead, a replacement operand could be up to 4 bytes in length.

Typical code patches will either use all NOPs to completely remove a command (by leaving the box empty and letting it fill the entire command with NOPs) or just replace a single operand, so this method of checking patch size is almost always effective.

In OllyDbg, you can also save your patches directly to the binary. First, right-click in the disassembler and click **Copy to Executable ▸ All Modifications**. If you want to copy only certain patches, highlight them in the disassembly pane and press **Copy to Executable ▸ Selection** instead.

Tracing Through Assembly Code

When you run a trace on any program, OllyDbg single-steps over every executed operation and stores data about each one. When the trace is complete, the logged data is displayed in the Run Trace window, shown in Figure 2-4.


Run trace

Back	Thread	Module	Address	Command	Modified registers
118.	Main	KERNELBA	7569FA8C	MOV EBP,ESP	EBP=0018FEFC
117.	Main	KERNELBA	7569FA8E	MOV EAX,DWORD PTR FS:[18]	EAX=7EFDD000
116.	Main	KERNELBA	7569FA94	MOV EAX,DWORD PTR DS:[EAX+30]	EAX=7EFDE000
115.	Main	KERNELBA	7569FA97	MOV ECX,DWORD PTR DS:[EAX+10]	ECX=00C21B48
114.	Main	KERNELBA	7569FA9A	MOV EAX,DWORD PTR SS:[EBP+8]	EAX=0018FF08
113.	Main	KERNELBA	7569FA9D	MOV DWORD PTR DS:[EAX],44	
112.	Main	KERNELBA	7569FAA3	MOV EDX,DWORD PTR DS:[ECX+84]	EDX=00C2285E
111.	Main	KERNELBA	7569FAA9	MOV DWORD PTR DS:[EAX+4],EDX	
110.	Main	KERNELBA	7569FAAC	MOV EDX,DWORD PTR DS:[ECX+7C]	EDX=00C2283E
109.	Main	KERNELBA	7569FAAF	MOV DWORD PTR DS:[EAX+8],EDX	
108.	Main	KERNELBA	7569FAB2	MOV EDX,DWORD PTR DS:[ECX+74]	EDX=00C227D6
107.	Main	KERNELBA	7569FAB5	MOV DWORD PTR DS:[EAX+C],EDX	
106.	Main	KERNELBA	7569FAB8	MOV EDX,DWORD PTR DS:[ECX+4C]	EDX=1F4A6AE7
105.	Main	KERNELBA	7569FABB	MOV DWORD PTR DS:[EAX+10],EDX	
104.	Main	KERNELBA	7569FABE	MOV EDX,DWORD PTR DS:[ECX+50]	EDX=FFFFFFFE
103.	Main	KERNELBA	7569FAC1	MOV DWORD PTR DS:[EAX+14],EDX	
102.	Main	KERNELBA	7569FAC4	MOV EDX,DWORD PTR DS:[ECX+54]	EDX=75D36901
101.	Main	KERNELBA	7569FAC7	MOV DWORD PTR DS:[EAX+18],EDX	
100.	Main	KERNELBA	7569FACA	MOV EDX,DWORD PTR DS:[ECX+58]	EDX=75D40AB0
99.	Main	KERNELBA	7569FACD	MOV DWORD PTR DS:[EAX+1C],EDX	
98.	Main	KERNELBA	7569FAD0	MOV EDX,DWORD PTR DS:[ECX+5C]	EDX=00000000
97.	Main	KERNELBA	7569FAD3	MOV DWORD PTR DS:[EAX+20],EDX	
96.	Main	KERNELBA	7569FAD6	MOV EDX,DWORD PTR DS:[ECX+60]	EDX=7790E5F1
95.	Main	KERNELBA	7569FAD9	MOV DWORD PTR DS:[EAX+24],EDX	
94.	Main	KERNELBA	7569FADC	MOV EDX,DWORD PTR DS:[ECX+64]	EDX=01310836
93.	Main	KERNELBA	7569FADF	MOV DWORD PTR DS:[EAX+28],EDX	
92.	Main	KERNELBA	7569FAE2	MOV EDX,DWORD PTR DS:[ECX+68]	EDX=00000081
91.	Main	KERNELBA	7569FAE5	MOV DWORD PTR DS:[EAX+2C],EDX	

Figure 2-4: The Run Trace window

The Run Trace window is organized into the following six columns:

Back The number of operations logged between an operation and the current execution state

Thread The thread that executed the operation

Module The module where the operation resides

Address The address of the operation

Command The operation that was executed

Modified Registers The registers changed by the operation and their new values

When hacking games, I find OllyDbg's trace feature very effective at helping me find pointer paths to dynamic memory when Cheat Engine scans prove inconclusive. This works because you can follow the log in the Run Trace window backward from the point when the memory is used to the point where it is resolved from a static address.

This potent feature's usefulness is limited only by the creativity of the hacker using it. Though I typically use it only to find pointer paths, I've come across a few other situations where it has proven invaluable. The example anecdotes in "OllyDbg Expressions in Action" on page 36 will help to illuminate the functionality and power of tracing.

OllyDbg's Expression Engine

OllyDbg is home to a custom expression engine that can compile and evaluate advanced expressions with a simple syntax. The expression engine is surprisingly powerful and, when utilized properly, can be the difference between an average OllyDbg user and an OllyDbg wizard. You can use this engine to specify expressions for many features, such as conditional breakpoints, conditional traces, and the Command Line plug-in. This section introduces the expression engine and the options it provides, and in "OllyDbg Expressions in Action" on page 36, I'll show you some ways I've used this handy tool in my own game-hacking experiences.

NOTE *Parts of this section are based on the official expressions documentation (*http://www.ollydbg.de/Help/i_Expressions.htm*). I have found, however, that a few of the components defined in the documentation don't seem to work, at least not in OllyDbg v1.10. Two examples are the* `INT` *and* `ASCII` *data types, which must be substituted with the aliases* `LONG` *and* `STRING`*. For this reason, here I include only components that I've personally tested and fully understand.*

Using Expressions in Breakpoints

When a *conditional breakpoint* is toggled on, OllyDbg prompts you to enter an expression for the condition; this is where most expressions are used. When that breakpoint is executed, OllyDbg silently pauses execution and evaluates the expression. If the result of the evaluation is nonzero, execution

remains paused and you will see the breakpoint get triggered. But if the result of the evaluation is 0, OllyDbg silently resumes execution as if nothing happened.

With the huge number of executions that happen within a game every second, you'll often find that a piece of code is executed in far too many contexts for a breakpoint to be an effective way of getting the data you are looking for. A conditional breakpoint paired with a good understanding of the code surrounding it is a foolproof way to avoid these situations.

Using Operators in the Expression Engine

For numeric data types, OllyDbg expressions support general C-style operators, as seen in Table 2-4. While there is no clear documentation on the operator precedence, OllyDbg seems to follow C-style precedence and can use parenthesized scoping.

Table 2-4: OllyDbg Numeric Operators

Operator	Function
`a == b`	Returns 1 if a is equal to b, else returns 0
`a != b`	Returns 1 if a is not equal to b, else returns 0
`a > b`	Returns 1 if a is greater than b, else returns 0
`a < b`	Returns 1 if a is less than b, else returns 0
`a >= b`	Returns 1 if a is greater than or equal to b, else returns 0
`a <= b`	Returns 1 if a is less than or equal to b, else returns 0
`a && b`	Returns 1 if a and b are both nonzero, else returns 0
`a || b`	Returns 1 if either a or b are nonzero, else returns 0
`a ^ b`	Returns the result of `XOR(a, b)`
`a % b`	Returns the result of `MODULUS(a, b)`
`a & b`	Return the result of `AND(a, b)`
`a | b`	Return the result of `OR(a, b)`
`a << b`	Returns the result of a shifted b bits to the left
`a >> b`	Returns the result of a shifted b bits to the right
`a + b`	Returns the sum of a plus b
`a - b`	Returns the difference of a minus b
`a / b`	Returns the quotient of a divided by b
`a * b`	Returns the product of a times b
`+a`	Returns the signed representation of a
`-a`	Returns `a*-1`
`!a`	Returns 1 if a is 0, else returns 0

For strings, on the other hand, the only available operators are `==` and `!=`, which both adhere to the following set of rules:

- String comparisons are case insensitive.
- If only one of the operands is a string literal, the comparison will terminate after it reaches the length of the literal. As a result, the expression `[STRING EAX]=="ABC123"`, where `EAX` is a pointer to the string `ABC123XYZ`, will evaluate to `1` instead of `0`.
- If no type is specified for an operand in a string comparison and the other operand is a string literal (for example, `"MyString"!=EAX`), the comparison will first assume the nonliteral operand is an ASCII string, and, if that compare would return `0`, it will try a second compare assuming the operand is a Unicode string.

Of course, operators aren't much use without operands. Let's look at some of the data you can evaluate in expressions.

Working with Basic Expression Elements

Expressions are able to evaluate many different elements, including:

CPU registers `EAX`, `EBX`, `ECX`, `EDX`, `ESP`, `EBP`, `ESI`, and `EDI`. You can also use the 1-byte and 2-byte registers (for example, `AL` for the low byte and `AX` for the low word of `EAX`). `EIP` can also be used.

Segment registers `CS`, `DS`, `ES`, `SS`, `FS`, and `GS`.

FPU registers `ST0`, `ST1`, `ST2`, `ST3`, `ST4`, `ST5`, `ST6`, and `ST7`.

Simple labels Can be API function names, such as `GetModuleHandle`, or user-defined labels.

Windows constants Such as `ERROR_SUCCESS`.

Integers Are written in hexadecimal format or decimal format if followed by a trailing decimal point (for example, `FFFF` or `65535.`).

Floating-point numbers Allow exponents in decimal format (for example, `654.123e-5`).

String literals Are wrapped in quotation marks (for example, `"my string"`).

The expressions engine looks for these elements in the order they're listed here. For example, if you have a label that matches the name of a Windows constant, the engine uses the address of the label instead of the constant's value. But if you have a label named after a register, such as EAX, the engine uses the register value, not the label value.

Accessing Memory Contents with Expressions

OllyDbg expressions are also powerful enough to incorporate memory reading, which you can do by wrapping a memory address, or an expression that evaluates to one, in square brackets. For example,`[EAX+C]` and `[401000]`

represent the contents at the addresses EAX+C and 401000. To read the memory as a type other than `DWORD`, you can specify the desired type either before the brackets, as in `BYTE [EAX]`, or as the first token within them, as in `[STRING ESP+C]`. Supported types are listed in Table 2-5.

Table 2-5: OllyDbg Data Types

Data type	Interpretation
`BYTE`	8-bit integer (unsigned)
`CHAR`	8-bit integer (signed)
`WORD`	16-bit integer (unsigned)
`SHORT`	16-bit integer (signed)
`DWORD`	32-bit integer (unsigned)
`LONG`	32-bit integer (signed)
`FLOAT`	32-bit floating-point number
`DOUBLE`	64-bit floating-point number
`STRING`	Pointer to an ASCII string (null-terminated)
`UNICODE`	Pointer to a Unicode string (null-terminated)

Plugging memory contents directly into your OllyDbg expressions is incredibly useful in game hacking, in part because you can tell the debugger to check a character's health, name, gold, and so on in memory before breaking. You'll see an example of this in "Pausing Execution When a Specific Player's Name Is Printed" on page 37.

OllyDbg Expressions in Action

Expressions in OllyDbg use a syntax similar to that of most programming languages; you can even combine multiple expressions and nest one expression within another. Game hackers (really, all hackers) commonly use them to create conditional breakpoints, as I described in "Using Expressions in Breakpoints" on page 33, but you can use them in many different places in OllyDbg. For instance, OllyDbg's Command Line plug-in can evaluate expressions in place and display their results, allowing you to easily read arbitrary memory, inspect values that are being calculated by assembly code, or quickly get the results of mathematical equations. Furthermore, hackers can even create intelligent, position-agnostic breakpoints by coupling expressions with the trace feature.

In this section, I'll share some anecdotes where the expression engine has come in handy during my work. There's no lab for this part, but I will explain my thought process, walk through my entire debugging session, and break each expression down into its component parts so you can see some ways to use OllyDbg expressions in game hacking.

NOTE *These examples contain some assembly code, but if you don't have much experience with assembly, don't worry. Just ignore the fine details and know that values like* `ECX`*,* `EAX`*, and* `ESP` *are process registers like the ones discussed in "Viewing and Editing Register Contents" on page 29. From there, I'll explain everything else.*

If you get confused about an operator, element, or data type in an expression as I walk through these anecdotes, just refer to "OllyDbg's Expression Engine" on page 33.

Pausing Execution When a Specific Player's Name Is Printed

During one particular debugging session, I needed to figure out exactly what was happening when a game was drawing the names of players on screen. Specifically, I needed to invoke a breakpoint before the game drew the name "Player 1," ignoring all other names that were drawn.

Figuring Out Where to Pause

As a starting point, I used Cheat Engine to find the address of Player 1's name in memory. Once I had the address, I used OllyDbg to set a memory breakpoint on the first byte of the string. Every time this breakpoint got hit, I quickly inspected the assembly code to determine how it was using Player 1's name. Eventually, I found the name being accessed directly above a call to a function that I had previously given the name `printText()`. I had found the code that was drawing the name.

I removed my memory breakpoint and replaced it with a code breakpoint on the call to `printText()`. There was a problem, however: because the call to `printText()` was inside a loop that iterated over every player in the game, my new breakpoint was getting hit every time a name was drawn—and that was much too often. I needed to fix it to hit only on a specific player.

Inspecting the assembly code at my previous memory breakpoint told me that each player's name was accessed using the following assembly code:

```
PUSH DWORD PTR DS:[EAX+ECX*90+50]
```

The `EAX` register contained the address of an array of player data; I'll call it `playerStruct`. The size of `playerStruct` was 0x90 bytes, the `ECX` register contained the iteration index (the famous variable `i`), and each player's name was stored 0x50 bytes after the start of its respective `playerStruct`. This meant that this `PUSH` instruction essentially put `EAX[ECX].name` (the name of the player at index `i`) on the stack to be passed as an argument to the `printText()` function call. The loop, then, broke down to something like the following psuedocode:

```
playerStruct EAX[MAX_PLAYERS]; // this is filled elsewhere
for (int ❶ECX = 0; ECX < MAX_PLAYERS; ECX++) {
    char* name = ❷EAX[ECX].name;
    breakpoint(); // my code breakpoint was basically right here
    printText(name);
}
```

Purely through analysis, I determined that the `playerStruct()` function contained data for all players, and the loop iterated over the total number of players (counting up with `ECX` ❶), fetched the character name ❷ for each index, and printed the name.

Crafting the Conditional Breakpoint

Knowing that, to pause execution only when printing "Player 1" all I had to do was check the current player name before executing my breakpoint. In pseudocode, the new breakpoint would look like this:

```
if (EAX[ECX].name == "Player 1") breakpoint();
```

Once I figured out the form of my new breakpoint, I needed to access `EAX[ECX].name` from within the loop. That's where OllyDbg's expression engine came in: I could achieve my goal by making slight modifications to the expression that the assembly code used, leaving me with this expression:

```
[STRING EAX + ECX*0x90 + 0x50] == "Player 1"
```

I removed the code breakpoint on `printText()` and replaced it with a conditional breakpoint that used this expression, which told OllyDbg to break only if the string value stored at `EAX + ECX*0x90 + 0x50` matched Player 1's name. This breakpoint hit only when `"Player 1"` was being drawn, allowing me to continue my analysis.

The amount of work it took to engineer this breakpoint might seem extensive, but with practice, the entire process becomes as intuitive as writing code. Experienced hackers can do this in a matter of seconds.

In practice, this breakpoint enabled me to inspect certain values in the `playerStruct()` function for `"Player 1"` as soon as he appeared on screen. Doing it this way was important, as the states of these values were relevant to my analysis only in the first few frames after the player entered the screen. Creatively using breakpoints like this can enable you to analyze all sorts of complex game behavior.

Pausing Execution When Your Character's Health Drops

During another debugging session, I needed to find the first function called after my character's health dropped below the maximum. I knew two ways to approach this problem:

- Find every piece of code that accesses the health value and place a conditional breakpoint that checks the health on each one. Then, once one of these breakpoints is hit, single-step through the code until the next function call.
- Use OllyDbg's trace function to create a dynamic breakpoint that can stop exactly where I need.

The first method required more setup and was not easily repeatable, mostly due to the sheer number of breakpoints needed and the fact that I'd have to single-step by hand. In contrast, the latter method had a quick setup, and since it did everything automatically, it was easily repeatable. Though using the trace function would slow the game down considerably (every single operation was captured by the trace), I chose the latter method.

Writing an Expression to Check Health

Once again, I started by using Cheat Engine to find the address that stored my health. Using the method described in "Cheat Engine's Memory Scanner" on page 5, I determined the address to be 0x40A000.

Next, I needed an expression that told OllyDbg to return `1` when my health was below maximum and return `0` otherwise. Knowing that my health was stored at 0x40A000 and that the maximum value was `500`, I initially devised this expression:

```
[0x40A000] < 500.
```

This expression would invoke a break when my health was below 500 (remember, decimal numbers must be suffixed with a period in the expression engine), but instead of waiting for a function to be called, the break would happen immediately. To ensure that it waited until a function was called, I appended another expression with the `&&` operator:

```
[0x40A000] < 500. && [❶BYTE EIP] == 0xE8
```

On x86 processors, the EIP register stores the address of the operation being executed, so I decided to check the first byte at `EIP` ❶ to see if it was equal to `0xE8`. This value tells the processor to execute a *near function call*, which is the type of call I was looking for.

Before starting my trace, I had to do one last thing. Because the trace feature repeatedly single-steps (Trace Into uses step into and Trace Over uses step over, as described in "A Brief Look at OllyDbg's User Interface" on page 24), I needed to start the trace at a location scoped at or above the level of any code that could possibly update the health value.

Figuring Out Where to Start the Trace

To find a good location, I opened the game's main module in OllyDbg's CPU window, right-clicked in the disassembler pane, and selected Search For ▸All Intermodular Calls. The References window popped up and displayed a list of external API functions that were called by the game. Nearly all gaming software polls for new messages using the Windows `USER32.PeekMessage()` function, so I sorted the list using the Destination column and typed `PEEK` (you can search the list by simply typing a name with the window in focus) to locate the first call to `USER32.PeekMessage()`.

Thanks to the Destination sorting, every call to this function was listed in a contiguous chunk following the first, as shown in Figure 2-5. I set a breakpoint on each by selecting it and pressing F2.

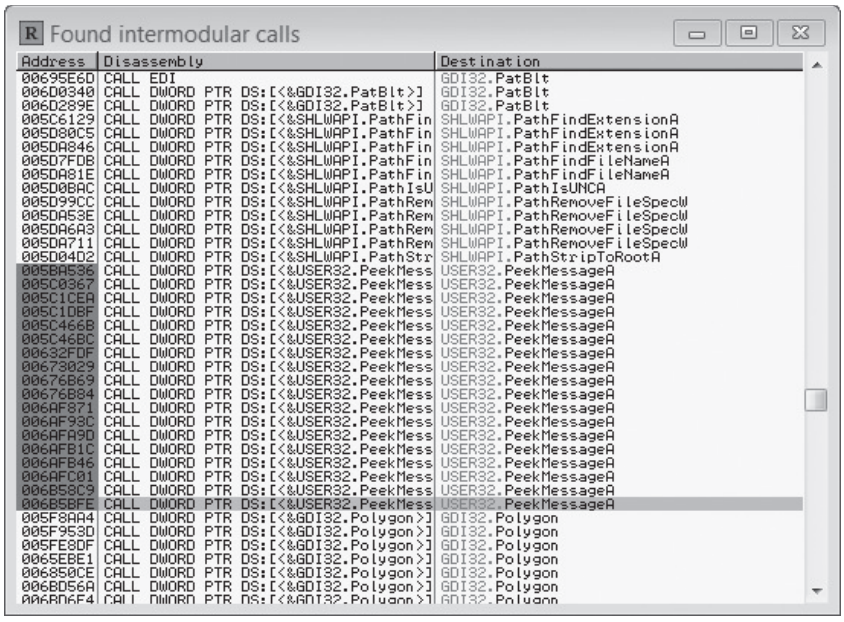


Figure 2-5: OllyDbg's Found Intermodular Calls window

Though there were around a dozen calls to `USER32.PeekMessage()`, only two of them were setting off my breakpoints. Even better, the active calls were beside one another in an unconditional loop. At the bottom of this loop were a number of internal function calls. This looked exactly like a main game loop.

Activating the Trace

To finally set my trace, I removed all of my previous breakpoints and placed one at the top of the suspected main loop. I removed the breakpoint as soon as it was hit. I then pressed CTRL-T from the CPU window, which brought up a window called Condition to Pause Run Trace, shown in Figure 2-6. Within this new window, I enabled the Condition Is TRUE option, placed my expression in the box beside it, and pressed OK. Then, I went back to the CPU window and pressed CTRL-F11 to begin a Trace Into session.

Once the trace began, the game ran so slowly it was nearly unplayable. To decrease my test character's health, I opened a second instance of the game, logged into a different character, and attacked my test character. When the execution of the trace caught up to real time, OllyDbg saw my health change and triggered the breakpoint on the following function call—just as expected.

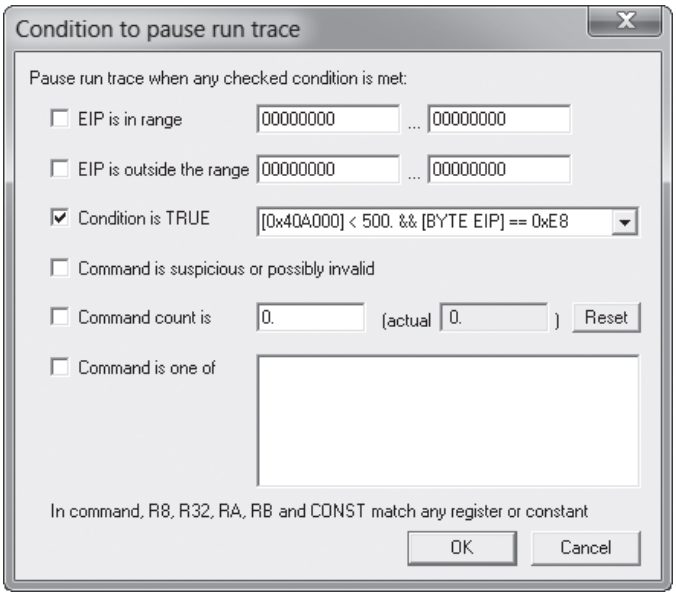


Figure 2-6: Condition to Pause Run Trace

In this game, the main pieces of code that would modify the health value were directly invoked from the network code. Using this trace, I was able to find the function that the network module called directly after a network packet told the game to change the player's health. Here's the psuedocode of what the game was doing:

```
void network::check() {
    while (this->hasPacket()) {
        packet = this->getPacket();
        if (packet.type == UPDATE_HEALTH) {
            oldHealth =  player->health;
            player->health = packet.getInteger();
❶           observe(HEALTH_CHANGE, oldHealth,  player->health);
        }
    }
}
```

I knew the game had code that needed to execute only when the player's health was changed, and I needed to add code that could also respond to such changes. Without knowing the overall code structure, I guessed that the health-dependent code would be executed from some function call directly after `health` was updated. My trace conditional breakpoint confirmed this hunch, as it broke directly on the `observe()` function ❶. From there, I was able to place a *hook* on the function (*hooking*, way to intercept function calls, will be described in Chapter 8) and execute my own code when the player's health changed.

OllyDbg Plug-ins for Game Hackers

OllyDbg's highly versatile plug-in system is perhaps one of its most powerful features. Experienced game hackers often configure their OllyDbg environments with dozens of useful plug-ins, both publicly available and custom-made.

You can download popular plug-ins from the OpenRCE (*http://www.openrce.org/downloads/browse/OllyDbg_Plugins*) and tuts4you (*http://www.tuts4you.com/download.php?list.9/*) plug-in repositories. Installing them is easy: just unzip the plug-in files and place them inside OllyDbg's installation folder.

Once installed, some plug-ins can be accessed from the OllyDbg's Plugin menu item. Other plug-ins, however, might be found only in specific places throughout the OllyDbg interface.

You can find hundreds of potent plug-ins using these online repositories, but you should be careful when constructing your arsenal. Working in an environment bloated by unused plug-ins can actually impede productivity. In this section, I've carefully selected four plug-ins that I believe are not only integral to a game hacker's toolkit but also noninvasive to the environment.

Copying Assembly Code with Asm2Clipboard

Asm2Clipboard is a minimalistic plug-in from the OpenRCE repository that allows you to copy chunks of assembly code from the disassembler pane to the clipboard. This can be useful for updating address offsets and devising code caves, two game-hacking essentials I cover deeply in Chapters 5 and 7.

With Asm2Clipboard installed, you can highlight a block of assembly code in the disassembler, right-click the highlighted code, expand the Asm2Clipboard submenu, and select either Copy Fixed Asm Code to Clipboard or Copy Asm Code to Clipboard. The latter prepends the code address of each instruction as a comment, while the former copies only the pure code.

Adding Cheat Engine to OllyDbg with Cheat Utility

The Cheat Utility plug-in from tuts4you provides a highly slimmed-down version of Cheat Engine within OllyDbg. While Cheat Utility only allows you to do exact-value scans with a very limited number of data types, it can make simple scans much easier when you don't need the full functionality of Cheat Engine to find what you're looking for. After installing Cheat Utility, to open its window (shown in Figure 2-7), select **Plugins ▸ Cheat Utility ▸ Start**.

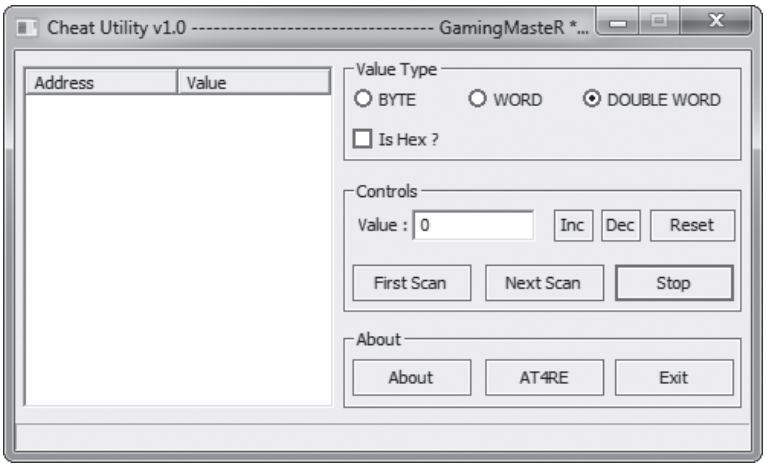


Figure 2-7: Cheat Utility interface

Cheat Utility's user interface and operation mimic Cheat Engine closely, so review Chapter 1 if you need a refresher.

NOTE *Games Invader, an updated version of Cheat Utility also from tuts4you, was created to provide more functionality. I've found it buggy, however, and I prefer Cheat Utility since I can always use Cheat Engine for advanced scans.*

Controlling OllyDbg Through the Command Line

The Command Line plug-in enables you to control OllyDbg through a small command line interface. To access the plug-in, either press ALT-F1 or select Plugins ▸ Command Line ▸ Command Line. You should then see a dialog, shown in Figure 2-8, which acts as the command line interface.

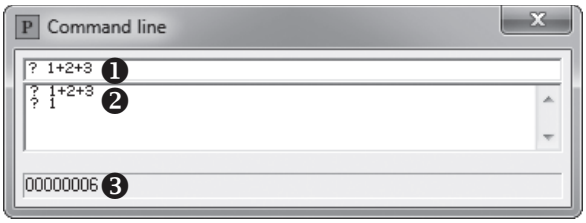


Figure 2-8: Command Line interface

To execute a command, type it into the input box ❶ and press ENTER. You will see a session-level command history in the center list ❷, and the bottom label displays the command's return value ❸ (if any).

Though there are many commands available, I find a majority of them useless. I primarily use this tool as a way to test that expressions are parsing as expected and as a handy calculator, but there are a few additional use cases that are also worth mentioning. I've described these in Table 2-6.

Table 2-6: Command Line Plug-in Commands

Command	Function
`BP identifier [,condition]`	Places a debugger breakpoint on `identifier`, which can be a code address or API function name. When `identifier` is an API function name, the breakpoint will be placed on the function entry point. The `condition` parameter is an optional expression that, if present, will be set as the breakpoint condition.
`BC identifier`	Removes any breakpoints present on `identifier`, which can be a code address or API function name.
`BPX label`	Places a debugger breakpoint on every instance of `label` within the module currently being disassembled. This `label` will typically be an API function name.
`MR address1, address2`	Places a memory on-access breakpoint starting at `address1` and spanning until `address2`. Will replace any existing memory breakpoint.
`MR address1, address2`	Places a memory on-write breakpoint starting at `address1` and spanning until `address2`. Will replace any existing memory breakpoint.
`MD`	Removes any existing memory breakpoint, if present.
`HR address`	Places a hardware on-access breakpoint on `address`. Only four hardware breakpoints can exist at a time.
`HW address`	Places a hardware on-write breakpoint on `address`.
`HE address`	Places a hardware on-execute breakpoint on `address`.
`HD address`	Removes any hardware breakpoints present on `address`.
`CALC expression` `? expression`	Evaluates `expression` and displays the result.
`WATCH expression` `W expression`	Opens the Watches window and adds `expression` to the watch list. Expressions in this list will be reevaluated every time the process receives a message and the evaluation results will be displayed beside them.

The Command Line plug-in was made by the OllyDbg developer and should come preinstalled with OllyDbg.

Visualizing Control Flow with OllyFlow

OllyFlow, which can be found in the OpenRCE plug-in directory, is a purely visual plug-in that can generate code graphs like the one in Figure 2-9 and display them using Wingraph32.[1]

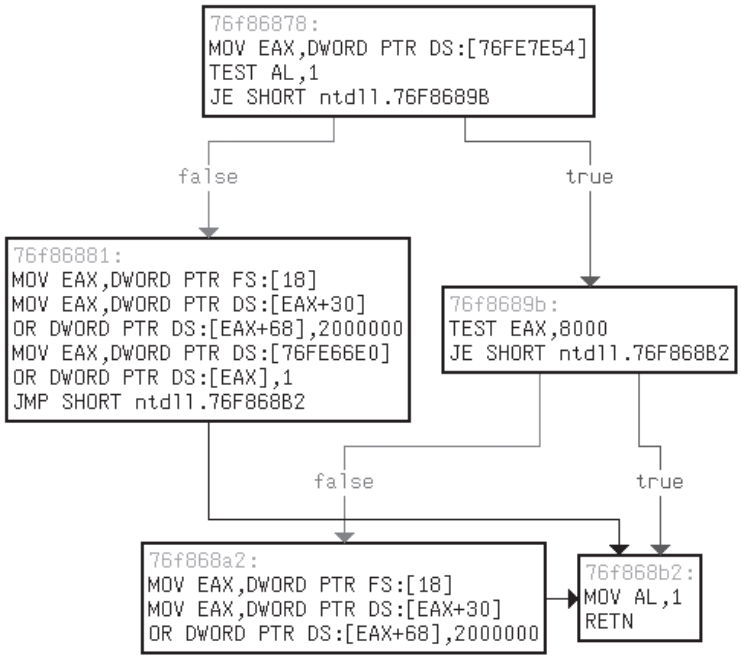


Figure 2-9: An OllyFlow function flowchart

Though not interactive, these graphs allow you to easily identify constructs such as loops and nested `if()` statements in game code, which can be paramount in control flow analysis. With OllyFlow installed, you can generate a graph by going to Plugins ▸ OllyFlow (alternatively, right-click in the disassembler pane and expand the OllyFlow Graph submenu) and selecting one of the following options:

Generate Function Flowchart Generates a graph of the function currently in scope, breaking apart different code blocks and showing jump paths. Figure 2-9 shows a function flowchart. Without a doubt, this is OllyFlow's most useful feature.

Generate XRefs from Graph Generates a graph of all functions called by the function that is currently in scope.

Generate XRefs to Graph Generates a graph of all functions that call the function currently in scope.

1. Wingraph32 is not provided with OllyFlow, but it is available with the free version of IDA here: *https://www.hex-rays.com/products/ida/*. Download it and drop the *.exe* in your OllyDbg installation folder.

Generate Call Stack Graph Generates a graph of the assumed call path from the process entry point to the function currently in scope.

Generate Module Graph Theoretically generates a complete graph of all function calls in the entire module, but rarely actually works.

To get an idea of the usefulness of OllyFlow, take a look at the graph in Figure 2-9 and compare it to the relatively simple assembly function that generated it:

```
76f86878:
❶    MOV EAX,DWORD PTR DS:[76FE7E54]
     TEST AL,1
     JE ntdll.76F8689B
76f86881:
❷    MOV EAX,DWORD PTR FS:[18]
     MOV EAX,DWORD PTR DS:[EAX+30]
     OR DWORD PTR DS:[EAX+68],2000000
     MOV EAX,DWORD PTR DS:[76FE66E0]
     OR DWORD PTR DS:[EAX],1
     JMP ntdll.76F868B2
76f8689b:
❸    TEST EAX,8000
     JE ntdll.76F868B2
76f868a2:
❹    MOV EAX,DWORD PTR FS:[18]
     MOV EAX,DWORD PTR DS:[EAX+30]
     OR DWORD PTR DS:[EAX+68],2000000
76f868b2:
❺    MOV AL,1
     RETN
```

There are five boxes in Figure 2-9, and they map to the five pieces of this function. The function starts with ❶, and it falls through to ❷ if the branch fails or jumps to ❸ if it succeeds. After ❷ executes, it jumps directly to piece ❺, which then returns out of the function. After ❸ executes, it either falls through to ❹ or branches to ❺ to return directly. After ❹ executes, it unconditionally falls through to ❺. What this function does is irrelevant to understanding OllyFlow; for now, just focus on seeing how the code maps to the graph.

LAB 2-1: PATCHING AN IF() STATEMENT

If you think you're ready to get your hands dirty with OllyDbg, keep reading. Go to *http://www.nostarch.com/gamehacking/*, download the book's resource files, grab *Lab02_01-BasicDebugging.exe*, and execute it. At first glance, you'll see that it looks like the classic game Pong. In this version of Pong, the ball is invisible to you when it is on your opponent's screen. Your task is to disable this feature so that you can always see the ball. To make it easier for you, I've made the game autonomous. You don't have to play, only hack.

To start, attach OllyDbg to the game. Then focus the CPU window on the main module (find the *.exe* in the module list and double-click it) and use the *Referenced Text Strings* feature to locate the string that is displayed when the ball is hidden. Next, double-click on the string to bring it up in the code and analyze the surrounding code until you find the `if()` statement that determines whether to hide the ball. Lastly, using the code-patching feature, patch the `if()` statement so the ball is always drawn. As an added bonus, you might try using OllyFlow to graph this function so you can get a better understanding of what exactly it is doing.

HINT The `if()` statement checks whether the ball's x-coordinate is less than 0x140. If so, it jumps to code that draws the ball. If not, it draws the scene without the ball. If you can change 0x140 to, say, 0xFFFF, the ball will never get hidden.

Closing Thoughts

OllyDbg is a much more complex beast than Cheat Engine, but you'll learn best by using it, so dive in and get your hands dirty! You can start by pairing the controls taught in this chapter with your debugging skills and going to work on some real games. If you are not yet ready to tamper with your virtual fate, however, try tackling Lab 2-1 for a practice environment. When you're done with the lab, read on to Chapter 3, where I'll introduce you to Process Monitor and Process Explorer, two tools you'll find invaluable in game-hacking reconnaissance.

3

RECONNAISSANCE WITH PROCESS MONITOR AND PROCESS EXPLORER

Cheat Engine and OllyDbg can help you tear apart a game's memory and code, but you also need to understand how the game interacts with files, registry values, network connections, and other processes. To learn how those interactions work, you must use two tools that excel at monitoring the external actions of processes: Process Monitor and Process Explorer. With these tools, you can track down the complete game map, locate save files, identify registry keys used to store settings, and enumerate the Internet Protocol (IP) addresses of remote game servers.

In this chapter, I'll teach you how to use both Process Monitor and Process Explorer to log system events and inspect them to see how a game was involved. Useful mainly for initial reconnaissance, these tools are

amazing at giving a clear, verbose picture of exactly how a game interacts with your system. You can download both programs from the Windows Sysinternals website (*https://technet.microsoft.com/en-us/sysinternals/*).

Process Monitor

You can learn a lot about a game simply by exploring how it interacts with the registry, filesystem, and network. Process Monitor is a powerful system-monitoring tool that logs such events in real time and lets you seamlessly integrate the data into a debugging session. This tool provides extensive amounts of useful data regarding a game's interaction with the external environment. With calculated review (and sometimes, spontaneous intuition) on your part, this data can reveal details about data files, network connections, and registry events that are helpful to your ability to see and manipulate how the game functions.

In this section, I'll show you how to use Process Monitor to log data, navigate it, and make educated guesses about the files a game interacts with. After this interface tour, you'll have a chance to try out Process Monitor for yourself in Lab 3-1 on page 55.

Logging In-Game Events

Process Monitor's logs can hold all sorts of potentially useful information, but their most practical use is to help you figure out where data files, such as in-game item definitions, might be stored. When you start Process Monitor, the first dialog you see is the Process Monitor Filter, shown in Figure 3-1.

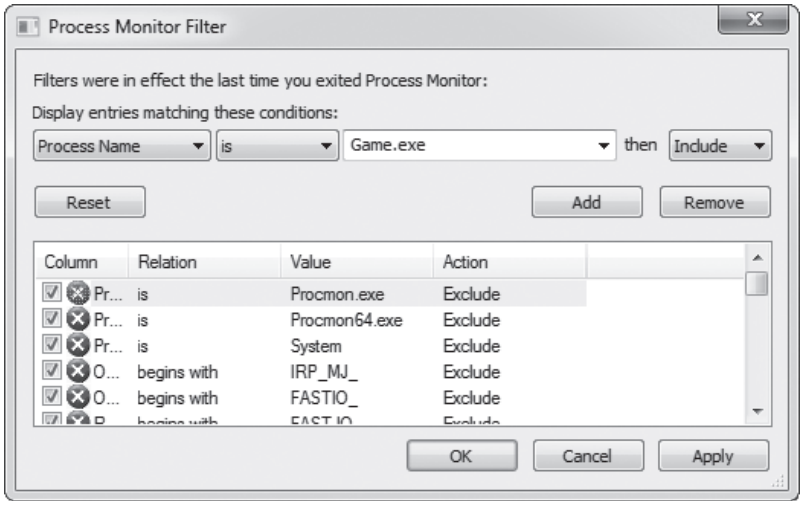


Figure 3-1: Process Monitor Filter dialog

This dialog allows you to show or suppress events based on a number of dynamic properties they possess. To start monitoring processes, select **Process Name** ▸ **Is** ▸ ***YourGameFilename.exe*** ▸ **Include** and then press **Add**,

Apply, and **OK**. This tells Process Monitor to show events invoked by *YourGameFilename.exe*. With the proper filters set, you will be taken to the main window shown in Figure 3-2.

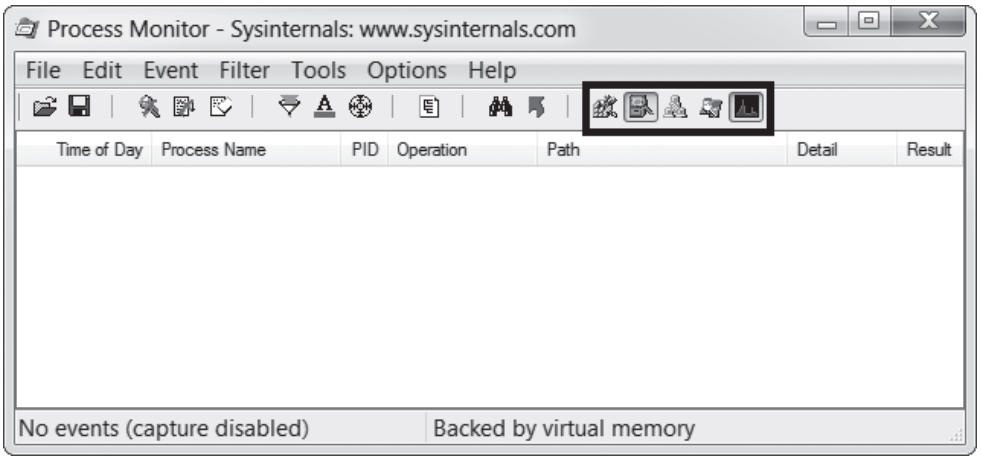


Figure 3-2: Process Monitor main window

To configure the columns displayed in Process Monitor's log area, right-click on the header and choose **Select Columns**. There's an impressive number of options, but I recommend seven.

Time of Day Lets you see when actions are happening.

Process Name Is useful if you're monitoring multiple processes, but with the single-process filter that is typically used for games; disabling this option can save precious space.

Process ID Is like Process Name, but it shows the ID rather than the name.

Operation Shows what action was performed; thus, this option is compulsory.

Path Shows the path of the action's target; also compulsory.

Detail Is useful only in some cases, but enabling it won't hurt.

Result Shows when actions, such as loading files, fail.

As you show more columns, the log can get very crowded, but sticking with these options should help keep the output succinct.

Once the monitor is running and you've defined the columns you wish to see, there are five event class filters, outlined in black in Figure 3-2, that you can toggle to clean up your logs even further. Event class filters let you choose which events to show in the log, based on type. From left to right, these filters are as follows:

Registry Shows all registry activity. There will be a lot of white noise in the registry upon process creation, as games rarely use the registry and Windows libraries always use it. Leaving this filter disabled can save a lot of space in the log.

File System Shows all filesystem activity. This is the most important event class filter, since knowing where data files are stored and how they are accessed is integral to writing an effective bot.

Network Shows all network activity. The call stack on network events can be useful in finding network-related code within a game.

Process and Thread Activity Shows all process and thread actions. The call stack on these events can give you insight into how a game's code handles threads.

Process Profiling Periodically shows information about the memory and CPU usage of each running process; a game hacker will rarely use it.

If class-level event filtering is still not precise enough to filter out unwanted pollution in your logs, right-click on specific events for event-level filtering options. Once you have your event filtering configured to log only what you need, you can begin navigating the log. Table 3-1 lists some useful hotkeys for controlling the log's behavior.

Table 3-1: Process Monitor Hotkeys

Hotkey	Action
CTRL-E	Toggles logging.
CTRL-A	Toggles automatic scrolling of the log.
CTRL-X	Clears the log.
CTRL-L	Displays the Filter dialog.
CTRL-H	Displays the Highlight dialog. This dialog looks very similar to the Filter dialog, but it is used to indicate which events should be highlighted.
CTRL-F	Displays the Search dialog.
CTRL-P	Displays the Event Properties dialog for the selected event.

As you navigate the log, you can examine the operations recorded to see the fine-grained details of an event.

Inspecting Events in the Process Monitor Log

Process Monitor logs every data point it possibly can about an event, enabling you to learn more about these events than just the files they act upon. Carefully inspecting data-rich columns, such as Result and Detail, can yield some very interesting information.

For example, I've found that games sometimes read data structures, element by element, directly from files. This behavior is apparent when a log contains a large number of reads to the same file, where each read has sequential offsets but differing lengths. Consider the hypothetical event log shown in Table 3-2.

Table 3-2: Example Event Log

Operation	Path	Detail
Create File	*C:\file.dat*	Desired Access: Read
Read File	*C:\file.dat*	Offset: 0 Size: 4
Read File	*C:\file.dat*	Offset: 4 Size: 2
Read File	*C:\file.dat*	Offset: 6 Size: 2
Read File	*C:\file.dat*	Offset: 8 Size: 4
Read File	*C:\file.dat*	Offset: 12 Size: 4
...	...	...Continues to read chunks of 4 bytes for a while

This log reveals that the game is reading a structure from the file piece by piece, disclosing some hints about what the structure looks like. For example, let's say that these reads reflect the following data file:

```
struct myDataFile
{
    int header;        // 4 bytes (offset 0)
    short effectCount; // 2 bytes (offset 4)
    short itemCount;   // 2 bytes (offset 6)
    int* effects;
    int* items;
};
```

Compare the log in Table 3-2 with this structure. First, the game reads the 4 `header` bytes. Then, it reads two 2-byte values: `effectCount` and `itemCount`. It then creates two integer arrays, `effects` and `items`, of respective lengths `effectCount` and `itemCount`. The game then fills these arrays with data from the file, reading 4 bytes for each `effectCount` and each `itemCount`.

NOTE *Developers definitely shouldn't use a process like this to read data from a file, but you'd be amazed at how often it happens. Fortunately for you, naïveté like this just makes your analysis easier.*

In this case, the event log can identify small pieces of information within a file. But keep in mind that, while correlating the reads with the known structure is easy, it's much harder to reverse-engineer an unknown structure from nothing but an event log. Typically, game hackers will use a debugger to get more context about each interesting event, and the data from Process Monitor can be seamlessly integrated into a debugging session, effectively tying together the two powerful reverse-engineering paradigms.

Debugging a Game to Collect More Data

Let's step away from this hypothetical file read and look at how Process Monitor lets you transition from event logging to debugging. Process Monitor stores a complete stack trace for each event, showing the full execution chain that led to the event being triggered. You can view these stack traces in the Stack tab of the Event Properties dialog (double-click the event or press CTRL-P), as shown in Figure 3-3.

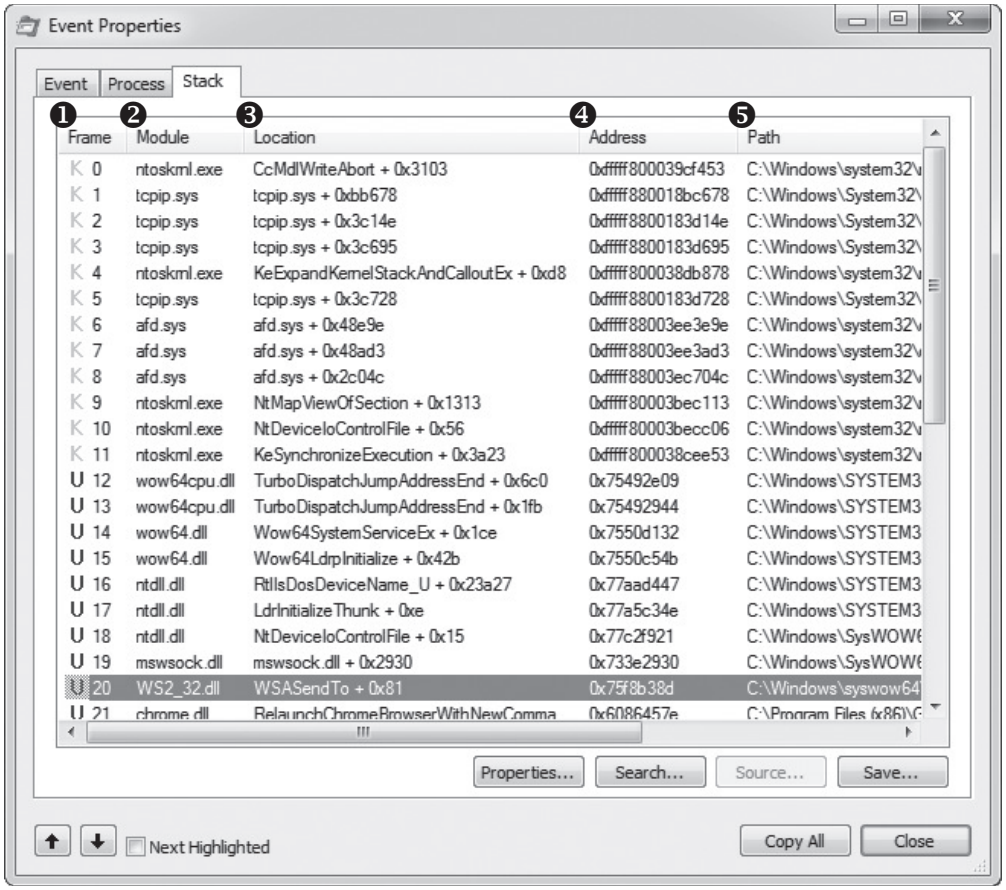


Figure 3-3: Process Monitor event call stack

The stack trace is displayed in a table starting with a Frame column ❶, which shows the execution mode and stack frame index. A pink *K* in this column means the call happened in kernel mode, while a blue *U* means it happened in user mode. Since game hackers typically work in user mode, kernel mode operations are usually meaningless.

The Module column ❷ shows the executable module where the calling code was located. Each module is just the name of the binary that made the call; this makes it easy to identify which calls were actually made from within a game binary.

The Location column ❸ shows the name of the function that made each call, as well as the call offset. These function names are deduced from the export table of the module and will generally not be present for the functions within a game binary. When no function names are present, the

Location column instead shows the module name and the call's *offset* (how many bytes past the origin address the call is in memory) from the module's base address.

NOTE *In the context of code, the offset is how many bytes of assembly code are between an item and its origin.*

The Address column ❹ shows the code address of the call, which is very useful because you can jump to the address in the OllyDbg disassembler. Finally, the Path column ❺ shows the path to the module that made the call.

In my opinion, the stack trace is, by far, the most powerful feature in Process Monitor. It reveals the entire context that led to an event, which can be immensely useful when you are debugging a game. You can use it to find the exact code that triggered an event, crawl up the call chain to see how it got there, and even determine exactly what libraries were used to complete each action.

LAB 3-1: FINDING A HIGH SCORE FILE

If you're ready to test your Process Monitor skills, you've come to the right place. Open the resource files for this book, and from the *GameHackingLabs/Chapter03* labs directory, execute *Lab03_01-FindingFiles.exe*. You'll see that it is a game of Pong, like the one in Lab 2-1. Unlike in Chapter 2, though, now the game is actually playable. It also displays your current score and your all-time-high score.

Now restart the game, firing up Process Monitor before executing it for the second time. Filtering for filesystem activity and creating any other filters you see fit, try to locate where the game stores the high-score file. For bonus points, try to modify this file to make the game show the highest possible score.

Process Monitor's sister application, Process Explorer, doesn't have many capabilities beyond those in Process Monitor or OllyDbg. But it does expose some of those capabilities much more effectively, making it an ideal pick in certain situations.

Process Explorer

Process Explorer is an advanced task manager (it even has a button you can press to make it your default task manager), and it's very handy when you're starting to understand how a game operates. It provides complex data about running processes, such as parent and child processes, CPU usage, memory usage, loaded modules, open handles, and command line arguments, and it can manipulate those processes. It exceeds at showing you high-level information, such as process trees, memory consumption, file access, and process IDs, all of which can be very useful.

Of course, none of this data is specifically useful in isolation. But with a keen eye, you can make correlations and draw some useful conclusions about what global objects—including files, mutexes, and shared memory segments—a game has access to. Additionally, the data shown in Process Explorer can be even more valuable when cross-referenced with data gathered in a debugging session.

This section introduces the Process Explorer interface, discusses the properties it shows, and describes how you can use this tool to manipulate *handles* (references to system resources). After this introduction, use Lab 3-2 on page 60 to hone your skills.

Process Explorer's User Interface and Controls

When you open Process Explorer, you see a window that is split into three distinct sections, as in Figure 3-4.

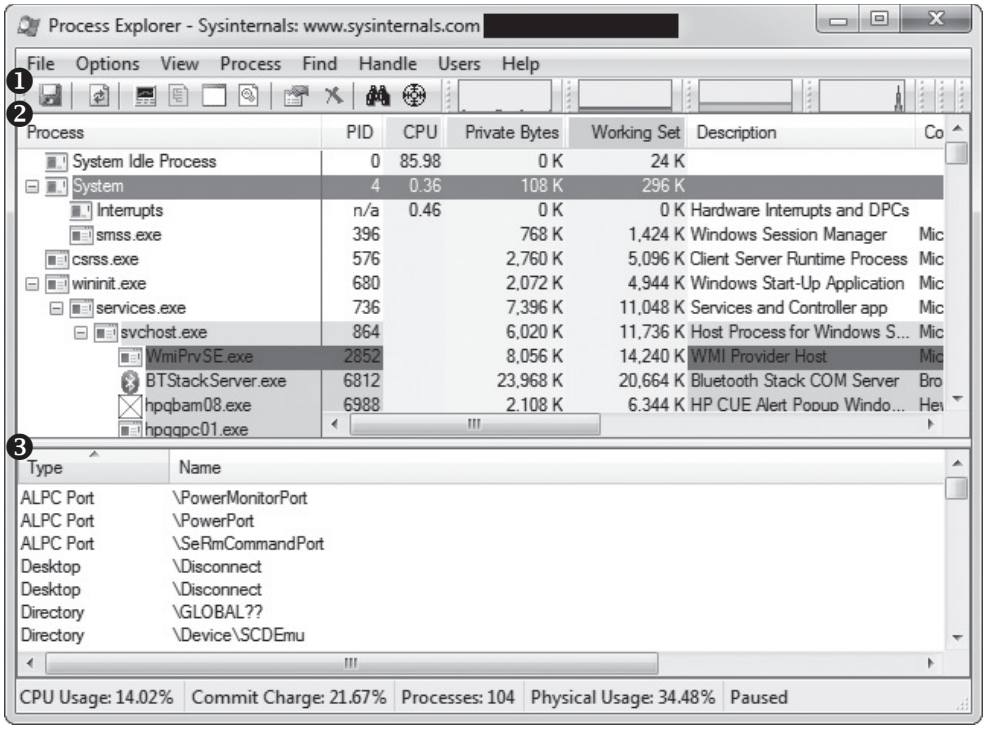


Figure 3-4: Process Explorer main window

Those three sections are the toolbar ❶, an upper pane ❷, and a lower pane ❸. The upper pane shows a list of processes, utilizing a tree structure to display their parent/child relationships. Different processes are highlighted with different colors; if you don't like the current colors, click **Options ▸ Configure Colors** to display a dialog that allows you to view and change them.

Just as in Process Monitor, the display for this table is highly versatile, and you can customize it by right-clicking on the table header and choosing Select Columns. There are probably more than 100 customization options, but I find that the defaults with the addition of the ASLR Enabled column work just fine.

NOTE Address Space Layout Randomization (ASLR) *is a Windows security feature that allocates executable images at unpredictable locations, and knowing whether it's on is invaluable when you're trying to alter game state values in memory.*

The lower pane has three possible states: Hidden, DLLs, and Handles. The Hidden option hides the pane from view, DLLs displays a list of Dynamic Link Libraries loaded within the current process, and Handles shows a list of handles held by the process (visible in Figure 3-4). You can hide or unhide the entire lower pane by toggling View ▸ Show Lower Pane. When it is visible, you can change the information display by selecting either View ▸ Lower Pane View ▸ DLLs or View ▸ Lower Pane View ▸ Handles.

You can also use hotkeys to quickly change between lower pane modes without affecting processes in the upper pane. These hotkeys are listed in Table 3-3.

Table 3-3: Process Explorer Hotkeys

Hotkey	Action
CTRL-F	Search through lower pane data sets for a value.
CTRL-L	Toggle the lower pane between hidden and visible.
CTRL-D	Toggle the lower pane to display DLLs.
CTRL-H	Toggle the lower pane to display handles.
SPACE	Toggle process list autorefresh.
ENTER	Display the Properties dialog for the selected process.
DEL	Kill the selected process.
SHIFT-DEL	Kill the selected process and all child processes.

Use the GUI or hotkeys to practice changing modes. When you're acquainted with the main window, we'll look at another important Process Explorer dialog, called Properties.

Examining Process Properties

Much like Process Monitor, Process Explorer has a very kinetic approach to data gathering; the end result is a broad and verbose spectrum of information. In fact, if you open the Properties dialog (shown in Figure 3-5) for a process, you'll see a massive tab bar containing 10 tabs.

The Image tab, selected by default and shown in Figure 3-5, displays the executable name, version, build date, and complete path. It also displays the current working directory and the Address Space Layout Randomization status of the executable. ASLR status is the most important piece of information here, because it has a direct effect on how a bot can read the memory from a game. I'll talk about this more in Chapter 6.

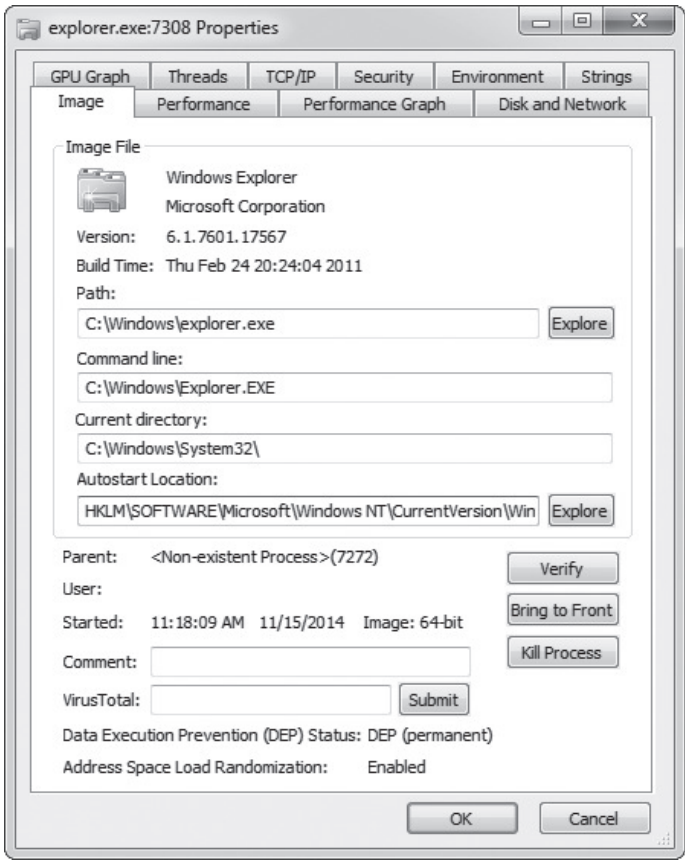


Figure 3-5: Process Explorer Properties dialog

The Performance, Performance Graph, Disk and Network, and GPU Graph tabs display a myriad of metrics about the CPU, memory, disk, network, and GPU usage of the process. If you create a bot that injects into a game, this information can be very useful to determine how much of a performance impact your bot has on the game.

The TCP/IP tab displays a list of active TCP connections, which you can use to find any game server IP addresses that a game connects to. If you're trying to test connection speed, terminate connections, or research a game's network protocol, this information is critical.

The Strings tab displays a list of strings found in either the binary or the memory of the process. Unlike the string list in OllyDbg, which shows only strings referenced by assembly code, the list includes any occurrences of three or more consecutive readable characters, followed by a null terminator. When a game binary is updated, you can use a diffing tool on this list from each game version to determine whether there are any new strings that you want to investigate.

The Threads tab shows you a list of threads running within the process and allows you to pause, resume, or kill each thread; the Security tab displays the security privileges of the process; and the Environment tab displays any environment variables known to or set by the process.

NOTE *If you open the Properties dialog for a .NET process, you'll notice two additional tabs: .NET Assemblies and .NET Performance. The data in these tabs is pretty self-explanatory. Please keep in mind that a majority of the techniques in this book won't work with games written in .NET.*

Handle Manipulation Options

As you've seen, Process Explorer can provide you with a wealth of information about a process. That's not all it's good for, though: it can also manipulate certain parts of a process. For example, you can view and manipulate open handles from the comfort of Process Explorer's lower pane (see Figure 3-4). This alone makes a strong argument for adding Process Explorer to your toolbox. Closing a handle is as simple as right-clicking on it and selecting Close Handle. This can come in handy when you want, for instance, to close mutexes, which is essential to certain types of hacks.

NOTE *You can right-click on the lower pane header and click Select Columns to customize the display. One column you might find particularly useful is Handle Value, which can help when you see a handle being passed around in OllyDbg and want to know what it does.*

Closing Mutexes

Games often allow only one client to run at a time; this is called *single-instance limitation*. You can implement single-instance limitation in a number of ways, but using a system mutex is common because mutexes are sessionwide and can be accessed by a simple name. It's trivial to limit instances with mutexes, and thanks to Process Explorer, it's just as trivial to remove that limit, allowing you to run multiple instances of a game at the same time.

First, here's how a game might tackle single-instance limitation with a mutex:

```
int main(int argc, char *argv[]) {
    // create the mutex
    HANDLE mutex = CreateMutex(NULL, FALSE, "onlyoneplease");
    if (GetLastError() == ERROR_ALREADY_EXISTS) {
        // the mutex already exists, so exit
        ErrorBox("An instance is already running.");
        return 0;
    }
    // the mutex didn't exist; it was just created, so
    // let the game run
    RunGame();
    // the game is over; close the mutex to free it up
    // for future instances
    if (mutex)
        CloseHandle(mutex);
    return 0;
}
```

This example code creates a mutex named `onlyoneplease`. Next, the function checks `GetLastError()` to see whether the mutex was already created, and if so, it closes the game. If the mutex doesn't already exist, the game creates the first instance, thereby blocking any future game clients from running. In this example, the game runs normally, and once it finishes, `CloseHandle()` is called to close the mutex and allow future game instances to run.

You can use Process Explorer to close instance-limiting mutexes and run many game instances simultaneously. To do so, choose the Handles view of the lower pane, look for all handles with a type of `Mutant`, determine which one is limiting instances of the game, and close that mutex.

WARNING *Mutexes are also used to synchronize data across threads and processes. Close one only if you're sure that its sole purpose is the one you're trying to subvert!*

Multiclient hacks are generally in high demand, so being able to quickly develop them for emerging games is crucial to your overall success as a bot developer within that market. Since mutexes are one of the most common ways to achieve single-instance limitation, Process Explorer is an integral tool for prototyping these kinds of hacks.

Inspecting File Accesses

Unlike Process Monitor, Process Explorer can't show a list of filesystem calls. On the other hand, the Handles view of Process Explorer's lower pane can show all file handles that a game currently has open, revealing exactly what files are in continuous use without the need to set up advanced filtering criteria in Process Monitor. Just look for handles with a type of `File` to see all files the game is currently using.

This functionality can come in handy if you're trying to locate logfiles or save files. Moreover, you can locate named pipes that are used for interprocess communication (IPC); these are files prefixed with *\Device\NamedPipe*. Seeing one of these pipes is often a hint that the game is talking to another process.

LAB 3-2: FINDING AND CLOSING A MUTEX

To put your Process Explorer skills to use, go to the *GameHackingLabs/Chapter03* labs directory and execute *Lab03_02-CloseMutex.exe*. This game plays exactly like the one in Lab 3-1, but it prevents you from simultaneously running multiple instances. As you might have guessed, it does this using a single-instance-limitation mutex. Using Process Explorer's Handles view in the lower pane, find the mutex responsible for this limitation and close it. If you succeed, you'll be able to open a second instance of the game.

Closing Thoughts

To be effective when using Process Monitor and Process Explorer, you need, above all else, a deep familiarity with the data that these applications display as well as the interfaces they use to display it. While this chapter's overview is a good baseline, the intricacies of these applications can be learned only through experience, so I encourage you to play around with them on your system.

In my day-to-day work I rarely use these tools, but I'll sometimes find myself struggling to figure something out, only to have the day saved when I recall some obscure piece of information that caught my eye during a previous Process Explorer or Process Monitor session. Because of this, I consider both to be useful reconnaissance tools.

PART II

GAME DISSECTION

4

FROM CODE TO MEMORY: A GENERAL PRIMER

At the lowest level, a game's code, data, input, and output are complex abstractions of erratically changing bytes. Many of these bytes represent variables or machine code generated by a compiler that was fed the game's source code. Some represent images, models, and sounds. Others exist only for an instant, posted by the computer's hardware as input and destroyed when the game finishes processing them. The bytes that remain inform the player of the game's internal state. But humans can't think in bytes, so the computer must translate them in a way we can understand.

There's a huge disconnect in the opposite direction as well. A computer doesn't actually understand high-level code and visceral game content, so these must be translated from the abstract into bytes. Some content—such as images, sounds, and text—is stored losslessly, ready to be presented to the player at a microsecond's notice. A game's code, logic, and variables, on the other hand, are stripped of all human readability and compiled down to machine data.

By manipulating a game's data, game hackers obtain humanly improbable advantages within the game. To do this, however, they must understand how a developer's code manifests once it has been compiled and executed. Essentially, they must think like computers.

To get you thinking like a computer, this chapter will begin by teaching you how numbers, text, simple structures, and unions are represented in memory at the byte level. Then you'll dive deeper to explore how class instances are stored in memory and how abstract instances know which virtual functions to call at runtime. In the last half of the chapter, you'll take an x86 assembly language crash course that covers syntax, registers, operands, the call stack, arithmetic operations, branching operations, function calls, and calling conventions.

This chapter focuses very heavily on general technical details. There isn't a lot of juicy information that immediately relates to hacking games, but the knowledge you gain here will be central in the coming chapters, when we talk about topics like programmatically reading and writing memory, injecting code, and manipulating control flow.

Since C++ is the de facto standard for both game and bot development, this chapter explains the relationships between C++ code and the memory that represents it. Most native languages have very similar (sometimes identical) low-level structure and behavior, however, so you should be able to apply what you learn here to just about any piece of software.

NOTE *You'll find all of the example code in this chapter in the* Examples/Chapter4_CodeToMemory directory *of this book's source files. The included projects can be compiled with Visual Studio 2010, but should also work with any other C++ compiler. Download them at* http://www.nostarch.com/gamehacking/ *and compile them if you want to follow along.*

How Variables and Other Data Manifest in Memory

Properly manipulating a game's state can be very hard, and finding the data that controls it is not always as easy as clicking Next Scan and hoping Cheat Engine won't fail you. In fact, many hacks must manipulate dozens of related values at once. Finding these values and their relationships often requires you to analytically identify structures and patterns. Moreover, developing game hacks typically means re-creating the original structures within your bot's code.

To do these things, you need an in-depth understanding of exactly how variables and data are laid out in the game's memory. Through some

example code, OllyDbg memory dumps, and some tables to tie everything together, this section will teach you everything there is to know about how different types of data manifest in memory.

Numeric Data

Most of the values game hackers need (like the player's health, mana, location, and level) are represented by numeric data types. Because numeric data types are also a building block for all other data types, understanding them is extremely important. Luckily, they have relatively straightforward representations in memory: they are predictably aligned and have a fixed bit width. Table 4-1 shows the five main numeric data types you'll find in Windows games, along with their sizes and ranges.

Table 5-1: Numeric Data Types

Type name(s)	Size	Signed range	Unsigned range
`char`, `BYTE`	8 bits	-128 to 127	0 to 255
`short`, `WORD`, `wchar_t`	16 bits	-32,768 to -32,767	0 to 65535
`int`, `long`, `DWORD`	32 bits	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
`long long`	64 bits	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615
`float`	32 bits	+/-1.17549*10^-38 to +/-3.40282*10^38	N/A

The sizes of numeric data types can differ between architectures and even compilers. Since this book focuses on hacking x86 games on Windows, I'm using type names and sizes made standard by Microsoft. With the exception of `float`, the data types in Table 4-1 are stored with *little-endian ordering*, meaning the least significant bytes of an integer are stored in the lowest addresses occupied by that integer. For example, Figure 4-1 shows that `DWORD` 0x0A0B0C0D is represented by the bytes 0x0D 0x0C 0x0B 0x0A.

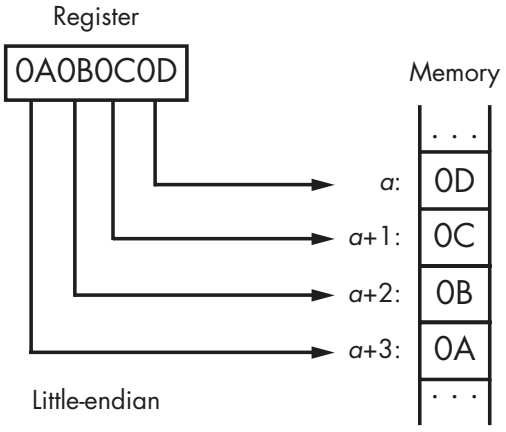


Figure 5-1: Little-endian ordering diagram

The `float` data type can hold mixed numbers, so its representation in memory isn't as simple as other data types. For example, if you see 0x0D 0x0C 0x0B 0x0A in memory and that value is a `float`, you can't simply convert it to 0x0A0B0C0D. Instead, `float` values have three components: the *sign* (bit 0), *exponent* (bits 1–8), and *mantissa* (bits 9–31).

The sign determines if the number is negative or positive, the exponent determines how many places to move the decimal point (starting before the mantissa), and the mantissa holds an approximation of the value. You can retrieve the stored value by evaluating the expression $mantissa \times 10^N$ (where N is the exponent) and multiplying the result by –1 if the sign is set.

Now let's look at some numeric data types in memory. Listing 4-1 initializes nine variables.

```
unsigned char ubyteValue = 0xFF;
char byteValue = 0xFE;
unsigned short uwordValue = 0x4142;
short wordValue = 0x4344;
unsigned int udwordValue = 0xDEADBEEF;
int dwordValue = 0xDEADBEEF;
unsigned long long ulongLongValue = 0xEFCDAB8967452301;
long long longLongValue = 0xEFCDAB8967452301;
float floatValue = 1337.7331;
```

Listing 5-1: Creating variables of numeric data types in C++

Starting from the top, this example includes variables of types `char`, `short`, `int`, `long long`, and `float`. Four of these are unsigned, and five are signed. (In C++, a `float` can't be unsigned.) Taking into account what you've learned so far, carefully study the relationship between the code in Listing 4-1 and the memory dump in Figure 4-2. Assume that the variables are declared in global scope.

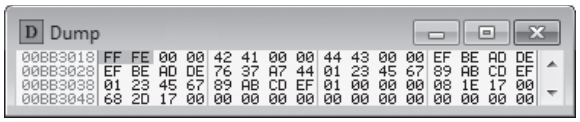


Figure 5-2: OllyDbg memory dump of our numeric data

You might notice that some values seem arbitrarily spaced out. Since it's much faster for processors to access values residing at addresses that are multiples of the address size (which is 32 bits in x86), compilers *pad* values with zeros in order to align them on such addresses—hence, padding is also called *alignment*. Single-byte values are not padded, since operations that access them perform the same regardless of alignment.

Keeping this in mind, take a look at Table 4-2, which provides a sort of memory-to-code crosswalk between the memory dump in Figure 4-2 and the variables declared in Listing 4-1.

Table 5-2: Memory-to-Code Crosswalk for Listing 4-1

Address	Size	Data	Object
0x00BB3018	1 byte	0xFF	`ubyteValue`
0x00BB3019	1 byte	0xFE	`byteValue`
0x00BB301A	2 bytes	0x00 0x00	Padding before `uwordValue`
0x00BB301C	2 bytes	0x42 0x41	`uwordValue`
0x00BB301E	2 bytes	0x00 0x00	Padding before `wordValue`
0x00BB3020	2 bytes	0x44 0x43	`wordValue`
0x00BB3022	2 bytes	0x00 0x00	Padding before `udwordValue`
0x00BB3024	4 bytes	0xEF 0xBE 0xAD 0xDE	`udwordValue`
0x00BB3028	4 bytes	0xEF 0xBE 0xAD 0xDE	`dwordValue`
0x00BB302C	4 bytes	0x76 0x37 0xA7 0x44	`floatValue`
0x00BB3030	8 bytes	0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF	`ulongLongValue`
0x00BB3038	8 bytes	0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF	`LongLongValue`

The Address column lists locations in memory, and the Data column tells you exactly what's stored there. The Object column tells you which variable from Listing 4-1 each piece of data relates to. Notice that `floatValue` is placed before `ulongLongValue` in memory, even though it's the last variable declared in Listing 4-1. Because these variables are declared in global scope, the compiler can place them wherever it wants. This particular move is likely a result of either alignment or optimization.

String Data

Most developers use the term "string" as if it's synonymous with "text," but text is only the most common use for strings. At a low level, strings are just arrays of arbitrary numeric objects that appear linear and unaligned in memory. Listing 4-2 shows four text string declarations.

```
// char will be 1 byte per character
char* thinStringP = "my_thin_terminated_value_pointer";
char thinStringA[40] = "my_thin_terminated_value_array";

// wchar_t will be 2 bytes per character
wchar_t* wideStringP = L"my_wide_terminated_value_pointer";
wchar_t wideStringA[40] = L"my_wide_terminated_value_array";
```

Listing 5-2: Declaring several strings in C++

In the context of text, strings hold character objects (`char` for 8-bit encoding or `wchar_t` for 16-bit encoding), and the end of each string is specified by a *null terminator*, a character equal to 0x0. Let's look at the memory where these variables are stored, as shown in the two memory dumps in Figure 4-3.

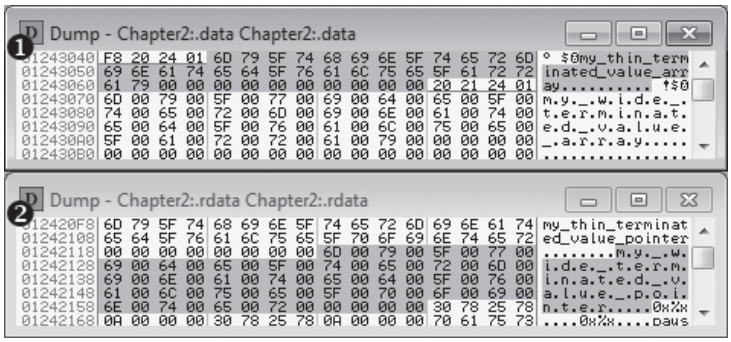


Figure 5-3: In this OllyDbg memory dump of string data, the human-readable text in the ASCII column is the text we stored in Listing 4-2.

If you're not used to reading memory, the OllyDbg dump might be a bit difficult to follow at this point. Table 4-3 shows a deeper look at the correlation between the code in Listing 4-2 and the memory in Figure 4-3.

Table 5-3: Memory-to-Code Crosswalk for Listing 4-2 and Figure 4-3

Address	Size	Data	Object
		Pane 2	
0x012420F8	32 bytes	0x6D 0x79 0x5F {...} 0x74 0x65 0x72	`thinStringP` characters
0x01242118	4 bytes	0x00 0x00 0x00 0x00	`thinStringP` terminator and padding
0x0124211C	4 bytes	0x00 0x00 0x00 0x00	Unrelated data
0x01242120	64 bytes	0x6D 0x00 0x79 {...} 0x00 0x72 0x00	`wideStringP` characters
0x01242160	4 bytes	0x00 0x00 0x00 0x00	`wideStringP` terminator and padding
{...}			Unrelated data
		Pane 1	
0x01243040	4 bytes	0xF8 0x20 0x24 0x01	Pointer to `thinStringP` at `0x012420F8`
0x01243044	30 bytes	0x6D 0x79 0x5F {...} 0x72 0x61 0x79	`thinStringA` characters
0x01243062	10 bytes	0x00 repeated 10 times	`thinStringA` terminator and array fill
0x0124306C	4 bytes	0x20 0x21 0x24 0x01	Pointer to `wideStringP` at `0x01242120`
0x01243070	60 bytes	0x6D 0x00 0x79 {...} 0x00 0x79 0x00	`wideStringA` characters
0x012430AC	20 bytes	0x00 repeated 10 times	`wideStringA` terminator and array fill

In Figure 4-3, pane ❶ shows that the values stored where `thinStringP` (address 0x01243040) and `wideStringP` (address 0x0124306C) belong in memory are only 4 bytes long and contain no string data. That's because these variables are actually pointers to the first characters of their respective arrays. For example, `thinStringP` contains 0x012420F8, and in pane ❷ in Figure 4-3, you can see `"my_thin_terminated_value_pointer"` located at address 0x012420F8.

Look at the data between these pointers in pane ❶, and you can see the text being stored by `thinStringA` and `wideStringA`. Furthermore, notice that `thinStringA` and `wideStringA` are padded beyond their null terminators; this is because these variables were declared as arrays with length `40`, so they are filled up to 40 characters.

Data Structures

Unlike the data types we have previously discussed, *structures* are containers that hold multiple pieces of simple, related data. Game hackers who know how to identify structures in memory can mimic those structures in their own code. This can greatly reduce the number of addresses they must find, as they need to find only the address to the start of the structure, not the address of every individual item.

NOTE *This section talks about structures as simple containers that lack member functions and contain only simple data. Objects that exceed these limitations will be discussed in "Classes" on page XX.*

Structure Element Order and Alignment

Since structures simply represent an assortment of objects, they don't visibly manifest in memory dumps. Instead, a memory dump of a structure shows the objects that are contained within that structure. The dump would look much like the others I've shown in this chapter, but with important differences in both order and alignment.

To see these differences, start by taking a look at Listing 4-3.

```
struct MyStruct {
    unsigned char ubyteValue;
    char byteValue;
    unsigned short uwordValue;
    short wordValue;
    unsigned int udwordValue;
    int dwordValue;
    unsigned long long ulongLongValue;
    long long longLongValue;
    float floatValue;
};
MyStruct& m = 0;
```

```
printf("Offsets: %d,%d,%d,%d,%d,%d,%d,%d,%d\n",
        &m->ubyteValue, &m->byteValue,
        &m->uwordValue, &m->wordValue,
        &m->udwordValue, &m->dwordValue,
        &m->ulongLongValue, &m->longLongValue,
        &m->floatValue);
```

Listing 5-3: Listing 4-3: A C++ structure and some code that uses it

This code declares a structure named `MyStruct` and creates a variable named `m` that supposedly points to an instance of the structure at address 0. There's not actually an instance of the structure at address 0, but this trick lets me use the ampersand operator (`&`) in the `printf()` call to get the address of each member of the structure. Since the structure is located at address 0, the address printed for each member is equivalent to its offset from the start of the structure.

The ultimate purpose of this example is to see exactly how each member is laid out in memory, relative to the start of the structure. If you were to run the code, you'd see the following output:

```
Offsets: 0,1,2,4,8,12,16,24,32
```

As you can see, the variables in `MyStruct` are ordered exactly as they were defined in code. This sequential member layout is a mandatory property of structures. Compare this to the example from Listing 4-1, when we declared an identical set of variables; in the memory dump from Figure 4-2, the compiler clearly placed some values out of order in memory.

Furthermore, you may have noticed that the members are not aligned like the globally scoped variables in Listing 4-1; if they were, for example, there would be two padding bytes before `uwordValue`.

This is because structure members are aligned on addresses divisible by either the *struct member alignment* (a compiler option that accepts 1, 2, 4, 8, or 16 bytes; in this example, it's set to 4) or the size of the member—whichever is smaller. I arranged the members of `MyStruct` so that the compiler didn't need to pad the values.

If, however, we put a `char` immediately after `ulongLongValue`, the `printf()` call would give the following output:

```
Offsets: 0,1,2,4,8,12,16,28,36
```

Now, take a look at the original and the modified outputs together:

```
Original: Offsets: 0,1,2,4,8,12,16,24,32
Modified: Offsets: 0,1,2,4,8,12,16,28,36
```

In the modified version, the last two values, which are the offsets for `longLongValue` and `floatValue` from the start of the structure, have changed. Thanks to the struct member alignment, the variable `longLongValue` moves by 4 bytes (1 for the `char` value and 3 following it) to ensure it gets placed on an address divisible by 4.

How Structures Work

Understanding structures, how they are aligned and how to mimic them, can be very useful. For instance, if you replicate a game's structures in your own code, you can read or write those entire structures from memory in a single operation. Consider a game that declares the player's current and max health like so:

```
struct {
    int current;
    int max;
} vital;
vital health;
```

If an inexperienced game hacker wants to read this information from memory, he might write something like this to fetch the health values:

```
int currentHealth = readIntegerFromMemory(currentHealthAddress);
int maxHealth =  readIntegerFromMemory(maxHealthAddress);
```

This game hacker doesn't realize that seeing these values right next to each other in memory could be more than a lucky happenstance, so he's used two separate variables. But if you came along with your knowledge of structures, you might come to the conclusion that, since these values are closely related and are adjacent in memory, he could have used a structure instead:

```
struct {
    int current;
    int max;
} _vital;
❶ _vital health = readTypeFromMemory<_vital>(healthStructureAddress);
```

Since this code assumes a structure is being used and correctly mimics it, it can fetch both health and max health in just one line ❶. We'll dive deeper in to how to write your own code to read memory in Chapter 6.

Unions

Unlike structures, which encapsulate multiple pieces of related data, *unions* contain a single piece of data that is exposed through multiple variables. Unions follow three rules:

- The size of a union in memory is equal to that of its largest member.
- Members of a union all reference the same memory.
- A union inherits the alignment of its largest member.

The `printf()` call in the following code helps illustrate the first two rules:

```
union {
    BYTE byteValue;
    struct {
```

```
        WORD first;
        WORD second;
    } words;
    DWORD value;
} dwValue;
dwValue.value = 0xDEADBEEF;
printf("Size %d\nAddresses 0x%x,0x%x\nValues 0x%x,0x%x\n",
    sizeof(dwValue), &dwValue.value, &dwValue.words,
    dwValue.words.first, dwValue.words.second);
```

This call to `printf()` outputs the following:

```
Size 4
Addresses 0x2efda8,0x2efda8
Values 0xbeef,0xdead
```

The first rule is illustrated by the `Size` value, which is printed first. Even though `dwValue` has three members that occupy a total of 9 bytes, it has a size of only 4 bytes. The size result also validates the second rule, because `dwValue.value` and `dwValue.words` both point to address `0x2EFDA8`, as shown by the values printed after the word `Addresses`. This is also validated by the fact that `dwValue.words.first` and `dwValue.words.second` contain `0xBEEF` and `0xDEAD`, printed after `Values`, which makes sense considering that `dwValue.value` is `0xDEADBEEF`. The third rule isn't demonstrated in this example because we don't have enough memory context, but if you were to put this union inside a structure and surround it with whatever types you like, it would in fact always align like a `DWORD`.

Classes and VF Tables

Much like structures, *classes* are containers that hold and isolate multiple pieces of data, but classes can also contain function definitions.

A Simple Class

Classes with normal functions, such as `bar` in Listing 4-4, conform to the same memory layouts as structures.

```
class bar {
public:
    bar() : bar1(0x898989), bar2(0x10203040) {}
    void myfunction() { bar1++; }
    int bar1, bar2;
};

bar _bar = bar();
printf("Size %d; Address 0x%x : _bar\n", sizeof(_bar), &_bar);
```

Listing 5-4: A C++ class

The `printf()` call in Listing 4-4 would output the following:

```
Size 8; Address 0x2efd80 : _bar
```

Even though `bar` has two member functions, this output shows that it spans only the 8 bytes needed to hold `bar1` and `bar2`. This is because the `bar` class doesn't include abstractions of those member functions, so the program can call them directly.

NOTE *Access levels such as `public`, `private`, and `protected` do not manifest in memory. Regardless of these modifiers, members of classes are still ordered as they are defined.*

A Class with Virtual Functions

In classes that do include abstract functions (often called *virtual* functions), the program must know which function to call. Consider the class definitions in Listing 4-5:

```
class foo {
public:
foo() : myValue1(0xDEADBEEF), myValue2(0xBABABABA) {}
    int myValue1;
    static int myStaticValue;
    virtual void bar() { printf("call foo::bar()\n"); }
    virtual void baz() { printf("call foo::baz()\n"); }
    virtual void barbaz() {}
    int myValue2;
};

int foo::myStaticValue = 0x12121212;

class fooa : public foo {
public:
    fooa() : foo() {}
    virtual void bar() { printf("call fooa::bar()\n"); }
    virtual void baz() { printf("call fooa::baz()\n"); }
};

class foob : public foo {
public:
    foob() : foo() {}
    virtual void bar() { printf("call foob::bar()\n"); }
    virtual void baz() { printf("call foob::baz()\n"); }
};
```

Listing 5-5: The `foo`, `fooa`, and `foob` classes

The class `foo` has three virtual functions: `bar`, `baz`, and `barbaz`. Classes `fooa` and `foob` inherit from class `foo` and overload both `bar` and `baz`. Since `fooa`

and `foob` have a public base class of `foo`, a `foo` pointer can point to them, but the program must still call the correct versions of `bar` and `baz`. You can see this by executing the following code:

```
foo* _testfoo = (foo*)new fooa();
_testfoo->bar(); // calls fooa::bar()
```

which prints this output:

```
call fooa::bar()
```

The output shows that `_testfoo->bar()` invoked `fooa::bar()` even though `_testfoo` is a `foo` pointer. The program knew which version of the function to call, because the compiler included a *VF (virtual function)* table in the memory of `_testfoo`. VF tables are arrays of function addresses that abstract class instances use to tell a program where their overloaded functions are located.

Class Instances and Virtual Function Tables

To understand the relationship between class instances and VF tables, let's inspect a memory dump of the three objects declared in this listing:

```
foo _foo = foo();
fooa _fooa = fooa();
foob _foob = foob();
```

These objects are of the types defined in Listing 4-5. You can see them in memory in Figure 4-4.

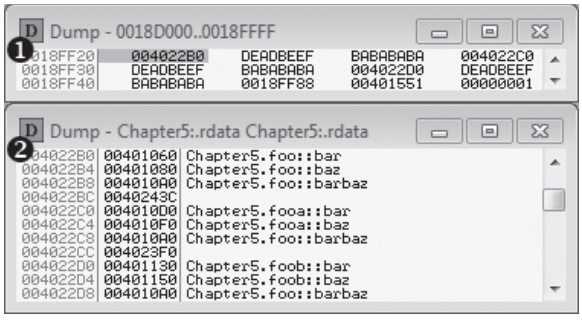


Figure 5-4: OllyDbg memory dump of class data

Pane ❶ shows that each class instance stores its members just like a structure, but precedes them with a `DWORD` value that points to the class instance's VF table. Pane ❷ shows the VF tables for each of our three class instances. The memory-to-code crosswalk in Table 4-4 shows how these panes and the code tie together.

Table 5-4: Memory-to-Code Crosswalk for Listing 4-5 and Figure 4-4

Address	Size	Data	Object
		Pane 1	
0x0018FF20	4 bytes	0x004022B0	Start of `_foo` and pointer to `foo` VF table
0x0018FF24	8 bytes	0xDEADBEEF 0xBABABABA	`_foo.myValue1` and `_foo.myValue2`
0x0018FF2C	4 bytes	0x004022C0	Start of `_fooa` and pointer to `fooa` VF table
0x0018FF30	8 bytes	0xDEADBEEF 0xBABABABA	`_fooa.myValue1` and `_fooa.myValue2`
0x0018FF38	4 bytes	0x004022D0	Start of `_foob` and pointer to `foob` VF table
0x0018FF3C	8 bytes	0xDEADBEEF 0xBABABABA	`_foob.myValue1` and `_foob.myValue2`
{...}			Unrelated data
		Pane 2	
0x004022B0	4 bytes	0x00401060	Start of `foo` VF table; address of `foo::bar`
0x004022B4	4 bytes	0x00401080	Address of `foo::baz`
0x004022B8	4 bytes	0x004010A0	Address of `foo::barbaz`
0x004022BC	4 bytes	0x0040243C	Unrelated data
0x004022C0	4 bytes	0x004010D0	Start of `fooa` VF table; address of `fooa::bar`
0x004022C4	4 bytes	0x004010F0	Address of `fooa::baz`
0x004022C8	4 bytes	0x004010A0	Address of `foo::barbaz`
0x004022CC	4 bytes	0x004023F0	Unrelated data
0x004022D0	4 bytes	0x00401130	Start of `foob` VF table; address of `foob::bar`
0x004022D4	4 bytes	0x00401150	Address of `foob::baz`
0x004022D8	4 bytes	0x004010A0	Address of `foo::barbaz`

This crosswalk shows how the VF tables for the code in Listing 4-5 are laid out in memory. Each VF table is generated by the compiler when the binary is made, and the tables remain constant. To save space, instances of the same class all point to the same VF table, which is why the VF tables aren't placed inline with the class.

Since we have three VF tables, you might wonder how a class instance knows which VF table to use. The compiler places code similar to the following bit of assembly in each virtual class constructor:

```
MOV DWORD PTR DS:[EAX], VFADDR
```

This example takes the static address of a VF table (`VFADDR`) and places it in memory as the first member of the class.

Now look at addresses 0x004022B0, 0x004022C0, and 0x004022D0 in Table 4-4. These addresses contain the beginning of the `foo`, `fooa`, and `foob` VF tables. Notice that `foo::barbaz` exists in all three VF tables; this is because the function is not overloaded by either subclass, meaning instances of each subclass will call the original implementation directly.

Notice, too, that `foo::myStaticValue` does not appear in this crosswalk. Since the value is static, it doesn't actually need to exist as a part of the `foo` class; it's placed inside this class only for better code organization. In reality, it gets treated like a global variable and is placed elsewhere.

> **VF TABLES AND CHEAT ENGINE**
>
> Remember Cheat Engine's First Element of Pointerstruct Must Point to a Module option for pointer scans from Chapter 1? Now that you've read a bit about VF tables, that knowledge should help you understand how this option works: it makes Cheat Engine ignore all heap chunks where the first member is not a pointer to a valid VF table. It speeds up scans, but it works only if every step in a pointer path is part of an abstract class instance.

The memory tour ends here, but if you have trouble identifying a chunk of data in the future, come back to this section for reference. Next, we'll look at how a computer can understand a game's high-level source code in the first place.

x86 Assembly Crash Course

When a program's source code is compiled into a binary, it is stripped of all unnecessary artifacts and translated into *machine code*. This machine code, made up of only bytes (command bytes are called *opcodes*, but there are also bytes representing operands), gets fed directly to the processor and tells it exactly how to behave. Those 1s and 0s flip transistors to control computation, and they can be extremely difficult to understand. To make computers a little easier to talk to, engineers working with such code use *assembly language*, a shorthand that represents raw machine opcodes with abbreviated names (called mnemonics) and a simplistic syntax.

Assembly language is important for game hackers to know because many powerful hacks can be achieved only through direct manipulation of a game's assembly code, via methods such as NOPing or hooking. In this section, you'll learn the basics of *x86 assembly language*, a specific flavor of

assembly made for speaking to 32-bit processors. Assembly language is very extensive, so for the sake of brevity this section talks only about the small subset of assembly concepts that are most useful to game hackers.[1]

NOTE *Throughout this section, many small snippets of assembly code include comments set off by a semicolon (;) to describe each instruction in greater detail.*

Command Syntax

Assembly language is used to describe machine code, so its syntax is pretty simplistic. While this syntax makes it very easy for someone to understand individual commands (also called *operations*), it also makes understanding complex blocks of code very hard. Even algorithms that are easily readable in high-level code seem obfuscated when written in assembly. For example, the following snippet of pseudocode:

```
if (EBX > EAX)
    ECX = EDX
else
    ECX = 0
```

would look like Listing 4-6 in x86 assembly:

```
    CMP EBX, EAX
    JG label1
    MOV ECX, 0
    JMP label2
label1:
    MOV ECX, EDX
label2:
```

Listing 5-6: Some x86 assembly commands

Therefore, it takes extensive practice to understand even the most trivial functions in assembly. Understanding individual commands, however, is very simple, and by the end of this section, you'll know how to parse the commands I just showed you.[2]

Instructions

The first part of an assembly command is called an *instruction*. If you equate an assembly command to a terminal command, the instruction is the program to run. At the machine code level, instructions are typically the first byte of a command; there are also some 2-byte instructions, where the first byte is 0x0F. Regardless, an instruction tells the processor exactly what to do. In Listing 4-6, `CMP`, `JG`, `MOV`, and `JMP` are all instructions.

1. Randall Hyde's *The Art of Assembly Language* (No Starch Press, 2003) is a wonderful book that can teach you everything there is to know about assembly.

2. Each command must fit within 16 bytes. Most commands are 6 or fewer.

Operand Syntax

While some instructions are complete commands, the vast majority are incomplete unless followed by *operands*, or parameters. Every command in Listing 4-6 has at least one operand, like `EBX`, `EAX`, and `label1`.

Assembly operands come in three forms:

Immediate value An integer value that is declared inline (hexadecimal values have a trailing `h`).

Register A name that refers to a processor register.

Memory offset An expression, placed in brackets, that represents the memory location of a value. The expression can be an immediate value or a register. Alternatively, it can be either the sum or difference of a register and immediate value (something like `[REG+Ah]` or `[REG-10h]`).

Each instruction in x86 assembly can have between zero and three operands, and commas are used to separate multiple operands. In most cases, instructions that require two operands have a *source operand* and a *destination operand*. The ordering of these operands is dependent on the assembly syntax. For example, Listing 4-7 shows a group of pseudocommands written in the Intel syntax, which is used by Windows (and, thus, by Windows game hackers):

```
MOV R1, 1          ; set R1 (register) to 1 (immediate)
❶ MOV R1, [BADF00Dh] ; set R1 to value at [BADF00Dh] (memory offset)
MOV R1, [R2+10h]   ; set R1 to value at [R2+10h] (memory offset)
MOV R1, [R2-20h]   ; set R1 to value at [R2+20h] (memory offset)
```

Listing 5-7: Demonstrating Intel syntax

In the Intel syntax, the destination operand comes first, followed by the source, so at ❶, `R1` is the destination and `[BADF00Dh]` is the source. On the other hand, compilers like GCC (which can be used to write bots on Windows) use a syntax known as AT&T, or UNIX, syntax. This syntax does things a little differently, as you can see in the following example:

```
MOV $1, %R1          ; set R1 (register) to 1 (immediate)
MOV 0xBADF00D, %R1   ; set R1 to value at 0xBADF00D (memory offset)
MOV 0x10(%R2), %R1   ; set R1 to value at 0x10(%R2) (memory offset)
MOV -0x20(%R2), %R1  ; set R1 to value at -0x20(%R2) (memory offset)
```

This code is the AT&T version of Listing 4-7. AT&T syntax not only reverses the operand order, but also requires operand prefixing and has a different format for memory offset operands.

Assembly Commands

Once you understand assembly instructions and how to format their operands, you can start writing commands. The following code shows an assembly function, consisting of some very basic commands, that essentially does nothing.

```
PUSH EBP      ; put EBP (register) on the stack
MOV EBP, ESP  ; set EBP to value of ESP (register, top of stack)
PUSH -1       ; put -1 (immediate) on the stack
ADD ESP, 4    ; negate the 'PUSH -1' to put ESP back where it was
              ; (a PUSH subtracts 4 from ESP, since it grows the stack)
MOV ESP, EBP  ; set ESP to the value of EBP (they will be the same
              ; anyway, since we have kept ESP in the same place)
POP EBP       ; set EBP to the value on top of the stack (it will be
              ; EBP started with, put on the stack by PUSH EBP)
XOR EAX, EAX  ; exclusive-or EAX (register) with itself (same
              ; effect as 'MOV EAX, 0' but much faster)
RETN          ; return from the function with a value of 0 (EAX
              ; typically holds the return value)
```

The first two lines, a `PUSH` command and a `MOV` command, set up a stack frame. The next line pushes –1 to the stack, which is undone when the stack is set back to its original position by the `ADD ESP, 4` command. Following that, the stack frame is removed, the return value (stored in `EAX`) is set to `0` with an `XOR` instruction, and the function returns.

You'll learn more about stack frames and functions in "The Call Stack" on page 86 and "Function Calls" on page 94. For now, turn your attention to the constants in the code—namely `EBP`, `ESP`, and `EAX`, which are used frequently in the code as operands. These values, among others, are called *processor registers*, and understanding them is essential to understanding the stack, function calls, and other low-level aspects of assembly code.

Processor Registers

Unlike high-level programming languages, assembly language does not have user-defined variable names. Instead, it accesses data by referencing its memory address. During intensive computation, however, it can be extremely costly for the processor to constantly deal with the overhead of reading and writing data to RAM. To mitigate this high cost, x86 processors provide a small set of temporary variables, called processor registers, which are small storage spaces within the processor itself. Since accessing these registers requires far less overhead than accessing RAM, assembly uses them to describe its internal state, pass volatile data around, and store context-sensitive variables.

General Registers

When assembly code needs to store or operate on arbitrary data, it uses a subset of process registers called *general registers*. These registers are used exclusively to store process-specific data, such as a function's local variables. Each general register is 32 bits and thus can be thought of as a `DWORD` variable. General registers are also optimized for specific purposes:

EAX, the accumulator This register is optimized for mathematical computations. Some operations, such as multiplication and division, can only occur in EAX.

EBX, the base register This register is used arbitrarily for extra storage. Since its 16-bit predecessor, BX, was the only register that operations could use to reference memory addresses, EBX was used as a reference to RAM. In x86 assembly, however, all registers can be address references, leaving EBX without a true purpose.

ECX, the counter This register is optimized to act as the counter variable (often called `i` in high-level code) in a loop.

EDX, the data register This register is optimized to act as a helper to EAX. In 64-bit computations, for instance, EAX acts as bits 0–31 and EDX acts as bits 32–63.

These registers also have a set of 8- and 16-bit subregisters that you can use to access partial data. Think of every general register as a union, where a register name describes the 32-bit member and the subregisters are alternate members that allow access to smaller pieces of the register. The following code shows what this union might look like for EAX:

```
union {
    DWORD EAX;
    WORD AX;
    struct {
        BYTE L;
        BYTE H;
    } A;
} EAX;
```

In this example, `AX` allows access to the lower `WORD` of `EAX`, while `AL` allows access to the lower `BYTE` of `AX` and `AH` to its higher `BYTE`. Every general register has this structure, and I outline the other registers' subregisters in Figure 4-5.

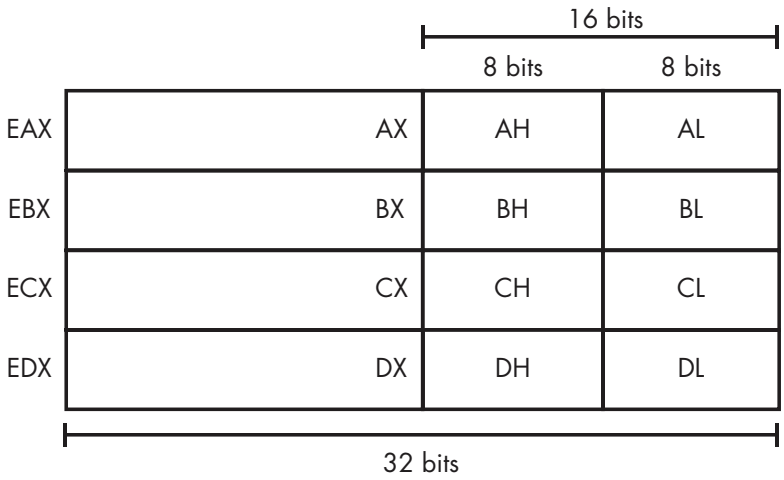


Figure 5-5: x86 registers and subregisters

EAX, EBC, ECX, and EDX have higher words, too, but the compiler will almost never access them on its own, as it can just use the lower word when it needs word-only storage.

Index Registers

x86 assembly also has four *index registers*, which are used to access data streams, reference the call stack, and keep track of local information. Like the general registers, index registers are 32 bits, but index registers have more strictly defined purposes:

EDI, the destination index This register is used to index memory targeted by write operations. If there are no write operations in a piece of code, the compiler can use EDI for arbitrary storage if needed.

ESI, the source index This register is used to index memory targeted by read operations. It can also be used arbitrarily.

ESP, the stack pointer This register is used to reference the top of the call stack. All stack operations directly access this register. You must use ESP only when working with the stack, and it must always point to the top of the stack.

EBP, the stack base pointer This register marks the bottom of the stack frame. Functions use it as a reference to their parameters and local variables. Some code may be compiled with an option to omit this behavior, in which case EBP can be used arbitrarily.

Like the general registers, each index register has a 16-bit counterpart: DI, SI, SP, and BP. However, the index registers have no 8-bit subregisters.

WHY DO SOME X86 REGISTERS HAVE SUBREGISTERS?

There is a historical reason why both general and index registers have 16-bit counterparts. The x86 architecture was based on a 16-bit architecture, from which it *extended* the registers AX, BX, CX, DX, DI, SI, SP, and BP. Appropriately, the extensions retain the same names, but are prefixed with an *E*, for "extended." The 16-bit versions remain for backward compatibility. This also explains why index registers have no 8-bit abstractions: they are intended to be used as memory-address offsets, and there is no practical need to know partial bytes of such values.

The Execution Index Register

The Execution Index register, referred to as EIP, has a very concrete purpose: it points to the address of the code currently being executed by the processor. Because it controls the flow of execution, it is directly incremented by the processor and is off-limits to assembly code. To modify EIP, assembly code must indirectly access it using operations such as `CALL`, `JMP`, and `RETN`.

The EFLAGS Register

Unlike high-level code, assembly language doesn't have binary comparison operators like ==, >, and <. Instead, it uses the `CMP` command to compare two values, storing the resulting information in the EFLAGS register. Then, the code changes its control flow using special operations that depend on the value stored in ELFAGS.

While comparison commands are the only user-mode operations that can access EFLAGS, they use only this register's *status* bits: 0, 2, 4, 6, 7, and 11. Bits 8–10 act as control flags, bits 12–14 and 16–21 act as system flags, and the remaining bits are reserved for the processor. Table 4-5 shows the type, name, and description of each EFLAGS bit.

Table 5-5: EFLAGS bits

Bit(s)	Type	Name	Description
0	Status	Carry	Set if a carry or borrow was generated from the most significant bit during the previous instruction.
2	Status	Parity	Set if the least significant `BYTE` resulting from the previous instruction has an even number of bits set.
4	Status	Adjust	Same as the carry flag, but considers the 4 least significant bits.
6	Status	Zero	Set if the resulting value from the previous instruction is equal to 0.
7	Status	Sign	Set if the resulting value from the previous instruction has its sign bit (most significant bit) set.
8	Control	Trap	When set, the processor sends an interrupt to the operating system kernel after executing the next operation.
9	Control	Interrupt	When not set, the system ignores maskable interrupts.
10	Control	Direction	When set, ESI and EDI are decremented by operations that automatically modify them. When not set, they are incremented.
11	Status	Overflow	Set when a value is overflowed by the previous instruction, such as when ADD is performed on a positive value and the result is a negative value.

The EFLAGS register also contains a system bit and a reserved bit, but those are irrelevant in user-mode assembly and game hacking, so I've omitted them from this table. Keep EFLAGS in mind when you're debugging game code to figure out how it works. For example, if you set a breakpoint on a JE (jump if equal) instruction, you can look at the EFLAGS 0 bit to see if the jump will be taken or not.

Segment Registers

Finally, assembly language has a set of 16-bit registers called *segment registers*. Unlike other registers, segment registers are not used to store data; they are used to locate it. In theory, they point to isolated segments of memory,

allowing different types of data to be stored in completely separate memory segments. The implementation of such segmentation is left up to the operating system. These are the x86 segment registers and their intended purposes:

CS, the code segment This register points to the memory that holds an application's code.

DS, the data segment This register points to the memory that holds an application's data.

ES, FS, and GS, the extra segments These registers point to any proprietary memory segments used by the operating system.

SS, the stack segment This register points to memory that acts as a dedicated call stack.

In assembly code, segment registers are used as prefixes to memory offset operands. When a segment register isn't specified, DS is used by default. This means that the command `PUSH [EBP]` is effectively the same as `PUSH DS:[EBP]`. But the command `PUSH FS:[EBP]` is different: it reads memory from the FS segment, not the DS segment.

If you look closely at the Windows x86 implementation of memory segmentation, you might notice that these segment registers were not exactly used as intended. To see this in action, you can run the following commands with the OllyDbg Command Line plug-in, while OllyDbg is attached to a paused process:

```
? CALC (DS==SS && SS==GS && GS==ES)
? 1
? CALC DS-CS
? 8
? CALC FS-DS ; returns nonzero (and changes between threads)
```

This output tells us three distinct things. First, it shows that there are only three segments being used by Windows: FS, CS, and everything else; this is demonstrated by DS, SS, GS, and ES being equal. For the same reason, this output shows that DS, SS, GS, and ES can all be used interchangeably, as they all point to the same memory segments. Lastly, since FS changes depending on the thread, this output shows that it is thread-dependent. FS is an interesting segment register, and it points to certain thread-specific data. In Chapter 6, we'll explore how the data in FS can be used to bypass ASLR—something most bots will need to do.

In fact, in assembly code generated for Windows by a compiler, you'd only ever see three segments used: DS, FS, and SS. Interestingly enough, even though CS seems to show a constant offset from DS, it has no real purpose in user-mode code. Knowing all of these things, you can further conclude that there are only two segments being used by Windows: FS and everything else.

These two segments actually point to different locations in the same memory (there's no simple way to verify this, but it is true), which shows that Windows actually doesn't use memory segments at all. Instead, it uses a

flat memory model in which segment registers are nearly irrelevant. While all segment registers point to the same memory, only FS and CS point to different locations, and CS is not used.

In conclusion, there are only three things you need to know about segment registers when working with x86 assembly in Windows. First, DS, SS, GS, and ES are interchangeable, but for clarity DS should be used to access data and SS should be used to access the call stack. Second, CS can be safely forgotten. Third, FS is the only segment register with a special purpose; it should be left alone for now.

The Call Stack

Registers are powerful, but unfortunately they come in very limited supply. In order for assembly code to effectively store all of its local data, it must also use the *call stack*. The stack is used to store many different values, including function parameters, return addresses, and some local variables.

Understanding the ins and outs of the call stack will come in handy when you're reverse-engineering a game. Moreover, you'll rely on this knowledge heavily when we jump into control flow manipulation in Chapter 8.

Structure

You can think of the call stack as a *FILO (first-in-last-out)* list of `DWORD` values that can be directly accessed and manipulated by assembly code. The name "stack" is used because the structure resembles a stack of paper: objects are both added to and removed from the top. Data is added to the stack through the `PUSH` *operand* command, and it is removed (and placed in a register) through the `POP` *register* command. Figure 4-6 shows how this process might look.

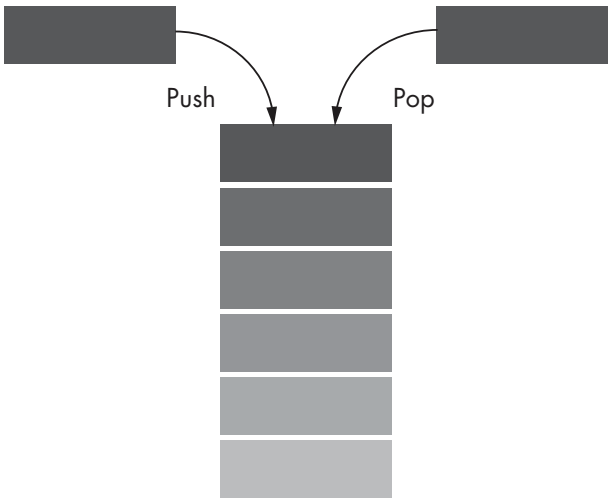


Figure 5-6: The structure of a stack

In Windows, the stack grows from higher memory addresses to lower ones. It occupies a finite block of memory, piling up to address 0x00000000 (the absolute top) from address *n* (the absolute bottom). This means that ESP (the pointer to the top of the stack) decreases as items are added and increases as items are removed.

The Stack Frame

When an assembly function uses the stack to store data, it references the data by creating a *stack frame*. It does so by storing ESP in EBP and then subtracting *n* bytes from ESP, effectively opening an *n*-byte gap that is *framed* between the registers EBP and ESP. To better understand this, first imagine that the stack in Figure 4-7 is passed to a function that requires 0x0C bytes of local storage space.

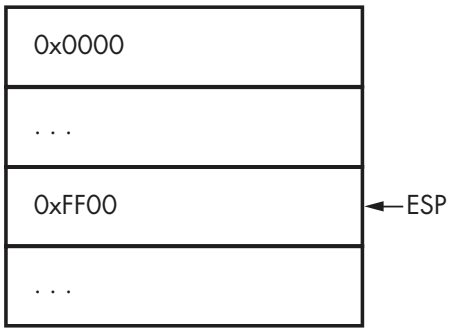


Figure 5-7: Initial example stack (read from bottom to top)

In this example, address 0x0000 is the absolute top of the stack. We have unused memory from addresses 0x0000 to 0xFF00 – 4, and at the time of the function call, 0xFF00 is the top of the stack. ESP points to this address. The stack memory after 0xFF00 is used by preceding functions in the call chain (from 0xFF04 to 0xFFFF). When the function is called, the first thing it does is execute the following assembly code, which creates a stack frame of 0x0C (12 in decimal) bytes:

```
PUSH EBP      ; saves the bottom of the lower stack frame
MOV EBP, ESP  ; stores the bottom of the current stack frame,
              ; in EBP (also 4 bytes above the lower stack frame)
SUB ESP, 0x0C ; subtracts 0x0C bytes from ESP, moving it up
              ; the stack to mark the top of the stack frame
```

After this code executes, the stack looks more like the one shown in Figure 4-8. After creating this stack, the function can work with the 0x0C bytes it allocated on the stack.

0x0000 is still the absolute top of the stack. We have unused stack memory from addresses 0x0000 to 0xFF00 – 20, and the memory at address 0xFF00 – 16 contains the final 4 bytes of local storage (referenced by `[EBP-Ch]`). This is also the top of the current stack frame, so ESP points here. 0xFF00 – 12 contains the middle 4 bytes of local storage (referenced by `[EBP-8h]`), and 0xFF00 – 8 contains the first 4 bytes of local storage (referenced by `[EBP-4h]`). EBP points to 0xFF00 – 4, which is the bottom of the current stack frame; this address holds the original value of EBP. 0xFF00 is

the top of the lower stack frame, and the original ESP in Figure 4-7 pointed here. Finally, you can still see the stack memory from preceding functions in the call chain from 0xFF04 to 0xFFFF.

Figure 5-8: Example stack with stack frame set up (read from bottom to top)

With the stack in this state, the function is free to use its local data as it pleases. If this function called another function, the new function would build its own stack frame using the same technique (the stack frames really stack up). Once a function finishes using a stack frame, however, it must restore the stack to its previous state. In our case, that means making the stack look like it did in Figure 4-7. When the second function finishes, our first function cleans the stack using the following two commands:

```
MOV ESP, EBP  ; demolishes the stack frame, bringing ESP to
              ; 4 bytes above its original value (0xFF00-4)
POP EBP       ; restores the bottom of the old stack frame
              ; that was saved by 'PUSH EBP'. Also adds 4 bytes
              ; to ESP, putting it back at its original value
```

But if you want to change the parameters passed to a function in a game, don't look for them in that function's stack frame. A function's parameters are stored in the stack frame of the function that called it, and they're referenced through `[EBP+8h]`, `[EBP+Ch]`, and so on. They start at `[EBP+8h]` because `[EBP+4h]` stores the function's return address. ("Function Calls" on page 94 explains more about this topic.)

NOTE *Code can be compiled with stack frames disabled. When this is the case, you'll notice that functions don't open with `PUSH EBP`, and instead reference everything relative to ESP. More often than not, though, stack frames are enabled in compiled game code.*

Now that you have a grasp on the fundamentals of assembly code, let's explore some specifics that will come in handy when hacking games.

Important x86 Instructions for Game Hacking

While assembly language has hundreds of instructions, many well-equipped game hackers understand only a small subset of them, which I cover in detail here. This subset typically encapsulates all instructions that are used to modify data, call functions, compare values, or jump around within code.

Data Modification

Data modification often happens over several assembly operations, but the end result has to be stored either in memory or in a register, typically with the `MOV` instruction. The `MOV` operation takes two operands: a destination and a source. Table 4-6 shows all possible sets of `MOV` operands and the results you can expect from those calls.

Table 5-6: Operands to the `MOV` Instruction

Instruction syntax	Result
`MOV R1, R2`	Copies `R2`'s value to `R1`.
`MOV R1, [R2]`	Copies the value from the memory referenced by `R2` to `R1`.
`MOV R1, [R2+Ah]`	Copies the value from the memory referenced by `R2+0xA` to `R1`.
`MOV R1, [DEADBEEFh]`	Copies the value from the memory at 0xDEADBEEF to `R1`.
`MOV R1, BADF00Dh`	Copies the value 0xBADF00D to `R1`.
`MOV [R1], R2`	Copies `R2`'s value to the memory referenced by `R1`.
`MOV [R1], BADF00Dh`	Copies the value 0xBADF00D to the memory referenced by `R1`.
`MOV [R1+4h], R2`	Copies `R2`'s value to the memory referenced by `R1+0x4`.
`MOV [R1+4h], BADF00Dh`	Copies the value 0xBADF00D to the memory referenced by `R1+0x4`.
`MOV [DEADBEEFh], R1`	Copies `R1`'s value to the memory at 0xDEADBEEF.
`MOV [DEADBEEFh], BADF00Dh`	Copies the value 0xBADF00D to the memory at 0xDEADBEEF.

The `MOV` instruction can take a lot of operand combinations, but some aren't allowed. First, the destination operand can't be an immediate value; it must be a register or memory address, because immediate values can't be modified. Second, values can't be directly copied from one memory address to another. Copying a value requires two separate operations, like so:

```
MOV EAX, [EBP+10h]   ; copy memory from EBP+0x10 to EAX
MOV [DEADBEEFh], EAX ; MOV the copied memory to memory at 0xDEADBEEF
```

Arithmetic

Like many high-level languages, assembly language has two types of arithmetic: unary and binary. Unary instructions take a single operand that acts as both a destination and a source. This operand can be a register or a memory address. Table 4-7 shows the common unary arithmetic instructions in x86.

Table 5-7: Unary Arithmetic Instructions

Instruction syntax	Result
`INC` *operand*	Adds 1 to the operand value.
`DEC` *operand*	Subtracts 1 from the operand value.
`NOT` *operand*	Logically negates the operand value (flips all bits).
`NEG` *operand*	Performs two's-complement negation (flips all bits and adds 1; essentially multiplies by -1).

Binary instructions (which make up the majority of x86 arithmetic), on the other hand, are syntactically similar to the `MOV` instruction. They require two operands and have similar operand limitations. Unlike `MOV`, however, their destination operand serves a second purpose: it is also the left-hand value in the calculation. For example, the assembly operation `ADD EAX,EBX` equates to `EAX = EAX + EBX` or `EAX += EBX` in C++. Table 4-8 shows the common x86 binary arithmetic instructions.

Table 5-8: Binary Arithmetic Instructions

Instruction syntax	Function	Operand notes
`ADD` *destination, source*	*destination* `+=` *source*	
`SUB` *destination, source*	*destination* `-=` *source*	
`AND` *destination, source*	*destination* `&=` *source*	
`OR` *destination, source*	*destination* `|=` *source*	
`XOR` *destination, source*	*destination* `^=` *source*	
`SHL` *destination, source*	*destination* `=` *destination* `<<` *source*	Source must be CL or an 8-bit immediate value.
`SHR` *destination, source*	*destination* `=` *destination* `>>` *source*	Source must be CL or an 8-bit immediate value.
`IMUL` *destination, source*	*destination* `*=` *source*	Destination must be a register; source cannot be an immediate value.

Of these arithmetic instructions, `IMUL` is special because you can pass it a third operand, in the form of an immediate value. With this prototype, the destination operand is no longer involved in the calculation, which instead takes place between the remaining operands. For example, the assembly command `IMUL EAX,EBX,4h` equates to `EAX = EBX * 0x4 in C++`.

You can also pass a single operand to `IMUL`.[3] In this case, the operand acts as the source and can be either a memory address or a register. Depending on the size of the source operand, the instruction will use different parts of the EAX register for inputs and output, as shown in Table 4-9.

Table 5-9: Possible `IMUL` Register Operands

Source size	Input	Output
8 bits	AL	16 bit, stored in AH:AL (which is AX).
16 bits	AX	32 bit, stored in DX:AX (bits 0–15 in AX and bits 16–31 in DX).
32 bits	EAX	64 bit, stored in EDX:EAX (bits 0–31 in EAX and bits 31–63 in EDX).

Notice that even though the input is only one register, each output uses two registers. That's because in multiplication, the result generally is larger than the inputs.

Let's look at an example calculation using `IMUL` with a single 32-bit operand:

```
IMUL [BADFOODh] ; 32-bit operand is at address 0xBADFOOD
```

This command behaves like the following pseudocode:

```
EDX:EAX = EAX * [BADFOODh]
```

Similarly, here's an operation that uses `IMUL` with a single 16-bit operand:

```
IMUL CX ; 16-bit operand is stored in CX
```

And its corresponding pseudocode:

```
DX:AX = AX * CX
```

Finally, this is an `IMUL` command with a single 8-bit operand:

```
IMUL CL ; 8-bit operand is stored in CL
```

3. There is also an unsigned multiplication instruction, `MUL`, which only works with a single operand.

And its corresponding pseudocode:

```
AX = AL * CL
```

x86 assembly language has division as well, through the `IDIV` instruction.[4] The `IDIV` instruction accepts a single source operand and follows register rules similar to those for `IMUL`. As Table 4-10 shows, `IDIV` operations require two inputs and two outputs.

Table 5-10: Possible `IDIV` Register Operands

Source size	Input	Output
8 bit	16 bit, stored in AH:AL (which is AX).	Remainder in AH; quotient in AL.
16 bit	32 bit, stored in DX:AX.	Remainder in DX; quotient in AX.
32 bit	64 bit, stored in EDX:EAX.	Remainder in EDX; quotient in EAX.

In division, the inputs are generally larger than the output, so here the inputs take two registers. Moreover, division operations must store a remainder, which gets stored in the first input register. For example, here's how a 32-bit `IDIV` calculation would look:

```
MOV EDX, 0          ; there's no high-order DWORD
                    ;   in the input, so EDX is 0
MOV EAX, inputValue ; 32-bit input value
IDIV ECX; divide EDX:EAX by ECX
```

And here's some pseudocode that expresses what happens under the hood:

```
EAX = EDX:EAX / ECX ; quotient
EDX = EDX:EAX % ECX ; remainder These details of IDIV and IMUL are important
to remember, as the behavior can otherwise be quite obfuscated when you're
simply looking at the commands.
```

Branching

After evaluating an expression, programs can decide what to execute next based on the result, typically using constructs such as `if()` statements or `switch()` statements. These control flow statements don't exist at the assembly level, however. Instead, assembly code uses the EFLAGS register to make decisions and jump operations to execute different blocks; this process is called *branching*.

To get the proper value in EFLAGS, assembly code uses one of two instructions: `TEST` or `CMP`. Both compare two operands, set the status bits of

4. Like `IMUL` and `MUL`, `DIV` is the unsigned counterpart to `IDIV`.

EFLAGS, and then discard any results. TEST compares the operands using a logical AND, while CMP uses signed subtraction to subtract the latter operand from the former.

In order to branch properly, the code has a jump command immediately following the comparison. Each type of jump instruction accepts a single operand that specifies the address of the code to jump to. How a particular jump instruction behaves depends on the status bits of EFLAGS. Table 4-11 describes some x86 jump instructions.

Table 5-11: Common x86 Jump Instructions

Instruction	Name	Behavior
JMP dest	Unconditional jump	Jumps to dest (sets EIP to dest).
JE dest	Jump if equal	Jumps if ZF (zero flag) is 1.
JNE dest	Jump if not equal	Jumps if ZF is 0.
JG dest	Jump if greater	Jumps if ZF is 0 and SF (sign flag) is equal to OF (overflow flag).
JGE dest	Jump if greater or equal	Jumps if SF is equal to OF.
JA dest	Unsigned JG	Jumps if CF (carry flag) is 0 and ZF is 0.
JAE dest	Unsigned JGE	Jumps if CF is 0.
JL dest	Jump if less	Jumps if SF is not equal to OF.
JLE dest	Jump if less or equal	Jumps if ZF is 1 or SF is not equal to OF.
JB dest	Unsigned JL	Jumps if CF is 1.
JBE dest	Unsigned JLE	Jumps if CF is 1 or ZF is 1.
JO dest	Jump if overflow	Jumps if OF is 1.
JNO dest	Jump if not overflow	Jumps if OF is 0.
JZ dest	Jump if zero	Jumps if ZF is 1 (identical to JE).
JNZ dest	Jump if not zero	Jumps if ZF is 0 (identical to JNE).

Remembering which flags control which jump instructions can be a pain, but their purpose is clearly expressed in their name. A good rule of thumb is that a jump preceded by a CMP is the same as its corresponding operator. For example, Table 4-11 lists JE as "jump if equal," so when JE follows a CMP operation, it's the same as the == operator. Similarly, JGE would be >=, JLE would be >=, and so on.

As an example, consider the high-level code shown in Listing 4-8:

```
//{.. unrelated code ..}
if (EBX > EAX)
    ECX = EDX;
else
    ECX = 0;
//{.. unrelated code ..}
```

Listing 5-8: A simple conditional statement

This `if()` statement just checks whether `EBX` is greater than `EAX`, and sets `ECX` based on the result. In assembly, the same statement may look something like this:

```
    {.. unrelated code ..}
    CMP EBX, EAX  ; if (EBX > EAX)
    JG label1     ; jump to label1 if EBX > EAX
    MOV ECX, 0    ; ECX = 0 (else block)
    JMP label2    ; jump over the if block
label1:
❶   MOV ECX, EDX ; ECX = EDX (if block)
label2:
    {.. unrelated code ..}
```

The assembly for the `if()` statement in Listing 4-8 begins with a `CMP` instruction and branches if `EBX` is greater than `EAX`. If the branch is taken, `EIP` is set to the `if` block at ❶ courtesy of the `JG` instruction. If the branch is not taken, the code continues executing linearly and hits the `else` block immediately after the `JG` instruction. When the `else` block finishes executing, an unconditional `JMP` sets `EIP` to `0x7`, skipping over the `if` block.

Function Calls

In assembly code, functions are isolated blocks of commands executed through the `CALL` instruction. The `CALL` instruction, which takes a function address as the only operand, pushes a return address onto the stack and sets EIP to its operand value. The following pseudocode shows a `CALL` in action, with memory addresses on the left in hex:

```
0x1: CALL EAX
0x2: {.. some stuff ..}
```

When `CALL EAX` is executed, the next address is pushed to the stack and `EIP` is set to `EAX`, showing that `CALL` is essentially a `PUSH` and `JMP`. The following pseudocode underscores this point:

```
0x1: PUSH 3h
0x2: JMP EAX
0x3: {.. some stuff ..}
```

While there's an extra address between the `PUSH` instruction and the code to execute, the result is the same: before the block of code at `EAX` is executed, the address of the code that follows the branch is pushed to the stack. This happens so the *callee* (the function being called) knows where to jump to in the *caller* (the function doing the call) when it returns.

If a function without parameters is called, a `CALL` command is all that's necessary. If the callee takes parameters, however, the parameters must first

be pushed onto the stack in reverse order. The following code shows how a function call with three parameters might look:

```
PUSH 300h   ; arg3
PUSH 200h   ; arg2
PUSH 100h   ; arg1
CALL ECX    ; call
```

When the callee is executed, the top of the stack contains a return address that points to the code after the call. The first parameter, `0x100`, is below the return address on the stack. The second parameter, `0x200`, is below that, followed by the third parameter, `0x300`. The callee sets up its stack frame, using memory offsets from `EBP` to reference each parameter. Once the callee has finished executing, it restores the caller's stack frame and executes the `RET` instruction, which pops the return address off the stack and jumps to it.

Since the parameters are not a part of the callee's stack frame, they remain on the stack after `RET` is executed. If the caller is responsible for cleaning the stack, it adds 12 (3 parameters, at 4 bytes each) to ESP immediately after `CALL ECX` completes. If the callee is responsible, it cleans up by executing `RET 12` instead of `RET`. This responsibility is determined by the callee's *calling convention*.

A function's calling convention tells the compiler how the assembly code should pass parameters, store instance pointers, communicate the return value, and clean the stack. Different compilers have different calling conventions, but the ones listed in Table 4-12 are the only four that a game hacker is likely to encounter.

Table 5-12: Calling Conventions to Know for Game Hacking

Directive	Cleaner	Notes
`__cdecl`	caller	Default convention in Visual Studio.
`__stdcall`	callee	Convention used by Win32 API functions.
`__fastcall`	callee	First two `DWORD` (or smaller) parameters are passed in ECX and EDX.
`__thiscall`	callee	Used for member functions. The pointer to the class instance is passed in ECX.

The Directive column in Table 4-12 gives the name of the calling convention, and the Cleaner column tells you whether the caller or callee is responsible for cleaning the stack given that directive. In the case of these four calling conventions, parameters are always pushed right to left, and return values are always stored in EAX. This is a standard, but not a rule; it can differ across other calling conventions.

Closing Thoughts

My goal in writing this chapter was to help you understand memory and assembly in a general sense, before we dig into game-hacking specifics. With your newfound ability to think like a computer, you should be adequately armed to start tackling more advanced memory forensics tasks. If you're itching for a peek at how you'll apply all of this to something real, flip to "Hooking Adobe Air" on page XX or "Hooking Direct3D" on page XX in Chapter 8.

If you want some hands-on time with memory, compile this chapter's example code and use Cheat Engine or OllyDbg to inspect, tweak, and poke at the memory until you've got the hang of it. This is important, as the next chapter will build on these skills by teaching you advanced memory forensic techniques.

5

ADVANCED MEMORY FORENSICS

Whether you hack games as a hobby or a business, you'll eventually find yourself between a rock and . . . an unintelligible memory dump. Be it a race with a rival bot developer to release a highly requested feature, a battle against a game company's constant barrage of updates, or a struggle to locate some complex data structure in memory, you'll need top-notch memory forensics skills to prevail.

Successful bot development is frailly balanced atop speed and skill, and tenacious hackers must rise to the challenge by swiftly releasing ingenious features, promptly responding to game updates, and readily searching for even the most elusive pieces of data. Doing this, however, requires a comprehensive understanding of common memory patterns, advanced data structures, and the purpose of different pieces of data.

Those three aspects of memory forensics are perhaps the most effective weapons in your arsenal, and this chapter will teach you how to use them. First, I'll discuss advanced memory-scanning techniques that focus on searching for data by understanding its purpose and usage. Next, I'll teach you how to use memory patterns to tackle game updates and tweak your bots without having to relocate all of your addresses from scratch. To wrap up, I'll dissect the four most common complex data structures in the C++ standard library (`std::string`, `std::vector`, `std::list`, and `std::map`) so you can recognize them in memory and enumerate their contents. By the end of the chapter, my hope is that you'll have a deep understanding of memory forensics and be able to take on any challenge related to memory scanning.

Advanced Memory Scanning

Within a game's source code, each piece of data has a cold, calculated definition. When the game is being played, however, all of that data comes together to create something new. Players only experience the beautiful scenery, visceral sounds, and intense adventures; the data that drives these experiences is irrelevant.

With that in mind, imagine Hacker A has just started tearing into his favorite game, wanting to automate some of the boring bits with a bot. He doesn't have a complete understanding of memory yet, and to him, the data is nothing but assumptions. He thinks, "I have 500 health, so I can find the health address by telling Cheat Engine to look for a 4-byte integer with a value of 500." Hacker A has an accurate understanding of data: it's just information (values) stored at particular locations (addresses) using defined structures (types).

Now imagine Hacker B, who already understands the game both inside and out; she knows how playing the game alters its state in memory, and the data no longer has any secrets. She knows that every defined property of the data can be determined given its purpose. Unlike Hacker A, Hacker B has an understanding of data that transcends the confines of a single variable declaration: she considers the data's *purpose* and *usage*. In this section, we'll discuss both.

Each piece of data in a game has a purpose, and the assembly code of the game must, at some point, reference the data to fulfill that purpose. Finding the unique code that uses a piece of data means finding a version-agnostic marker that persists across game updates until the data is either removed or its purpose is changed. Let me show you why this is important.

Deducing Purpose

So far, I've only shown you how to blindly search memory for a given piece of data without considering how it's being used. This method can be effective, but it is not always efficient. In many cases, it's much quicker to deduce the purpose of data, determine what code might use that data, and then locate that code to ultimately find the address of the data.

This might not sound easy, but neither does "scan the game's memory for a specific value of a specific data type, and then continuously filter the result list based on changing criteria," which is what you've learned to do thus far. So let's look at how we might locate the address for health given its purpose.

Consider the code in Listing 5-1.

```
struct PlayerVital {
    int current, maximum;
};
PlayerVital health;
{..}
printString("Health: %d of %d\n", health.current, health.maximum);
```

Listing 6-1: A structure containing the player's vitals, and a function that displays them

If you pretend that `printString()` is a fancy function to draw text on an in-game interface, then this code is pretty close to what you might find in a game. The `PlayerVital` structure has two properties: the `current` value and a `maximum` value. The value `health` is a `PlayerVital` structure, so it has these properties, too. Based on the name alone, you can deduce that `health` exists to display information about the player's health, and you can see this purpose fulfilled when `printString()` uses the data.

Even without the code, you can intuitively draw similar conclusions by just looking at the health text displayed in the game's interface; a computer can't do anything without code, after all. Aside from the actual `health` variable, there are a few code elements that need to exist to show a player this text. First, there needs to be some function to display text. Second, the strings `Health` and `of` must be nearby.

NOTE *Why do I assume the text is split into two separate strings instead of one? The game interface shows that the current health value is between these two strings, but there are many ways that could happen, including format strings, `strcat()`, or text aligned with multiple display text calls. When you're analyzing data, it's best to keep your assumptions broad to account for all possibilities.*

To find `health` without using a memory scanner, we could utilize these two distinct strings. We probably wouldn't have a clue what the function to display text looks like, where it is, or how many times it's called, though. Realistically, the strings are all we would know to look for, and that's enough. Let's walk through it.

Finding the Player's Health with OllyDbg

I'll walk you through how to track down the `health` structure in this section, but I've also included the binary I analyze in the book's resource files. To follow along and get some hands-on practice, use the file */GameHackingExamples/Chapter5_AdvancedMemoryForensics_Scanning.exe*.

First, open OllyDbg and attach it to the executable. Then, open OllyDbg's Executable Modules window and double-click the main module; in my example, the main module is the only *.exe* in the module's window. The CPU window should pop up. Now, right-click in the Disassembler pane and select **Search For ▸ All Referenced Text Strings**. This should open the References window, shown in Figure 5-1.

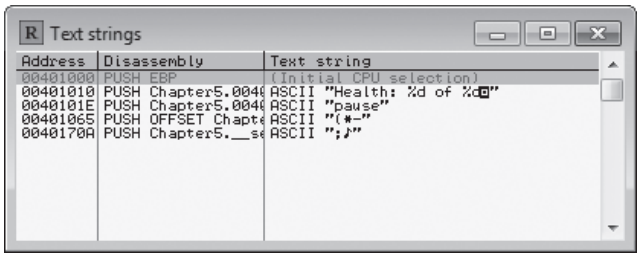


Figure 6-1: OllyDbg's References window, showing only a list of strings. There would be a lot more than four in a real game.

From this window, right-click and select **Search For Text**. A search dialog appears. Enter the string you're looking for, as shown in Figure 5-2, and make the search as broad as possible by disabling **Case Sensitive** and enabling **Entire Scope**.

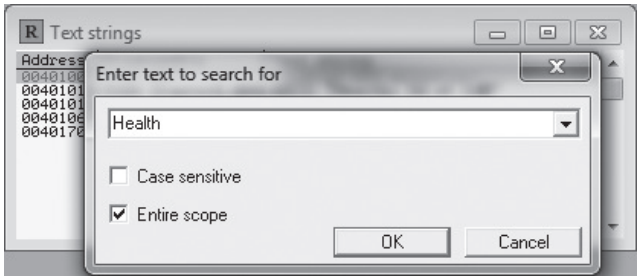


Figure 6-2: Searching for strings in OllyDbg

Click **OK** to execute the search. The References window comes back into focus with the first match highlighted. Double-click the match to see the assembly code that uses the string inside the CPU window. The Disassembler pane focuses on the line of code at 0x401030, which pushes the format string parameter to `printString()`. You can see this line in Figure 5-3, where I've highlighted the entire function call block.

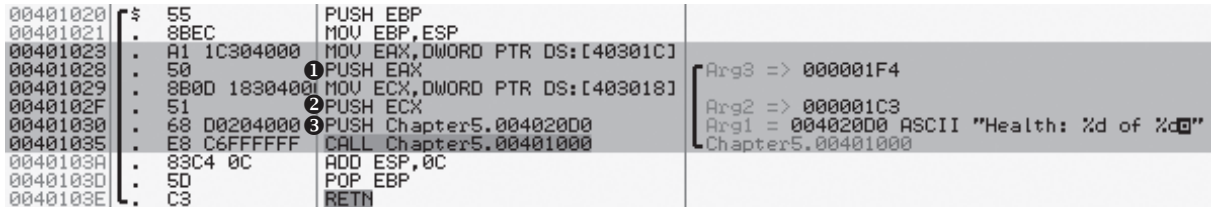


Figure 6-3: Viewing the `printString()` call in the CPU window's Disassembler pane

By reading the assembly code, you can get a very accurate understanding of exactly what the game is doing. The black bracket on the left shows that the string `Health` is inside a function call. Notice the arguments to that function. In order, these are EAX ❶, ECX ❷, and the format string at 0x4020D0 ❸. EAX is the value at 0x40301C, ECX is the value at 0x403018, and the format string contains `Health`. Since the string contains two format placeholders, you can assume that the remaining two parameters are the arguments for those placeholders.

Knowing what the arguments are and that they are pushed in reverse order, you can work backward and conclude that the original code looked something like Listing 5-2.

```
int currentHealth; //Value at 0x403018
int maxHealth;     //Value at 0x40301C
{..}
someFunction("Health: %d of %d\n",
    currentHealth, maxHealth);
```

Listing 6-2: How a game hacker might interpret the assembly that Figure 5-3 compiles to

The values stored in EAX and ECX are adjacent in memory, which means they may be part of a structure. To keep it simple, though, this example just shows them as variable definitions. Either way, these are the two numbers used to display the player's health. Because both of these important values were displayed in the game's UI, it was easy to make assumptions about the underlying code that displays them. When you know the purpose of a piece of data, you can quickly find the code responsible for fulfilling it; in this case, that knowledge helped us quickly find both addresses.

AUTOMATICALLY FIND CURRENTHEALTH AND MAXHEALTH

In "Searching for Assembly Patterns" on page 19 and "Searching for Strings" on page 21, I showed a few Cheat Engine Lua scripts and explained how they worked. Using the `findString()` function in these examples, you can make Cheat Engine automatically locate the address of the format string that we just found manually in OllyDbg. Next, you can write a small function to scan for this address following byte 0x68 (the byte for the `PUSH` command, as you can see beside it at 0x401030 in Figure 5-3) to locate the address of the code that pushes it to the stack. Then, you can read 4 bytes from `pushAddress - 5` and `pushAddress - 12` to locate `currentHealth` and `maxHealth`, respectively.

This may not seem useful since we've already found the addresses, but if this were a real game, these addresses would change when an update is released. Using this knowledge to automate finding them can be very helpful. If you're up to the challenge, think of this as a lab and give it a whirl!

In many cases, finding addresses can be this easy, but some pieces of data have such complex purposes that it's harder to guess what to look for. Figuring out how to search for map data or character locations in OllyDbg, for instance, can be pretty tricky.

Strings are far from the only markers that you can use to find the data you want to change in a game, but they are definitely the easiest to teach without giving contrived examples. Moreover, some games have logging or error strings embedded in their code, and poking around in the Referenced Text Strings window of OllyDbg can be a quick way to determine whether these strings are present. If you become familiar with a game's logging practices, you'll be able to find values even more easily.

Determining New Addresses After Game Updates

When application code is modified and recompiled, a brand new binary that reflects the changes is produced. This binary might be very similar to the previous one, or they might be nothing alike; the difference between the two versions has a direct correlation to the complexity of the high-level changes. Small changes, like modified strings or updated constants, can leave binaries nearly identical and often have no effect on the addresses of code or data. But more complex changes—like added features, a new user interface, refactored internals, or new in-game content—often cause shifts in the location of crucial memory.

Due to constant bug fixes, content improvements, and feature additions, online games are among the most rapidly evolving types of software. Some games release updates as often as once a week, and game hackers often spend a majority of their time reverse-engineering the new binaries in order to accordingly update their bots.

TIPS FOR WINNING THE UPDATE RACE

In saturated markets, being the first bot developer to release a stable update is critical to success. The race starts the second the game updates, and hackers determined to be the fastest will spend hundreds of hours preparing. These are the most common ways to stay on top:

Create update alarms By writing software that alerts you as soon as the game patches, you can begin working on your updates as soon as possible.

Automate bot installs Games often schedule expected updates at times when the fewest players are online. Botters hate waking up and downloading new software before they bot, but they love waking up to have it silently installed while the game is patching.

Use fewer addresses The less there is to update, the better. Consolidating related data into structures and eliminating unnecessary memory address usage can save a bunch of time.

Have great test cases Data changes, and hackers make mistakes. Having ways to quickly test every feature can be the difference between a stable bot and one that randomly crashes, gets users killed, or even leads to their characters being banned from the game.

Attacking updates with these practices will give you a sizable head start, but they might not always be enough to lead you to victory. Above all else, strive to understand reverse engineering as much as possible, and use that understanding to your advantage.

If you create advanced bots, they will become increasingly supported by a foundation of memory addresses. When an update comes, determining the new addresses for a large number of values and functions is the most time-consuming inevitability you will face. Relying on the "Tips for Winning the Update Race" on page 102 can be very beneficial, but the tips won't help you locate the updated addresses. You can automatically locate some addresses using Cheat Engine scripts, but that won't always be the case, either. Sometimes you'll have to do the dirty work by hand.

If you try to reinvent the wheel and find these addresses the same way you did initially, you'll be wasting your time. You actually have a big advantage, though: the old binary and the addresses themselves. Using these two things, it is possible to find every single address you need to update in a fraction of the time.

Figure 5-4 shows two different disassemblies: a new game binary on the left, and the previous version on the right. I have taken this image from an actual game (which will remain nameless) in order to give you a realistic example.

```
0047B653  .  57           PUSH EDI                                  0047B523  .  57           PUSH EDI
0047B654  .  50           PUSH EAX                                  0047B524  .  50           PUSH EAX
0047B655  .  8D45 F4      LEA EAX,DWORD PTR SS:[EBP-C]              0047B525  .  8D45 F4      LEA EAX,DWORD PTR SS:[EBP-C]
0047B658  .  64:A3 000000 MOV DWORD PTR FS:[0],EAX                  0047B528  .  64:A3 000000 MOV DWORD PTR FS:[0],EAX
0047B65E  .  8965 F0      MOV DWORD PTR SS:[EBP-10],ESP             0047B52E  .  8965 F0      MOV DWORD PTR SS:[EBP-10],ESP
0047B661  .  C785 E0FDFFF MOV DWORD PTR SS:[EBP-220],-1             0047B531  .  C785 E0FDFFF MOV DWORD PTR SS:[EBP-220],-1
0047B66B  .  C745 FC 0000 MOV DWORD PTR SS:[EBP-4],0                0047B538  .  C745 FC 0000 MOV DWORD PTR SS:[EBP-4],0
0047B672  >  E8 298A0B00  CALL      005340A0                        0047B542  >  E8 D9890B00  CALL      00533F20
0047B677  .  8BF8         MOV EDI,EAX                               0047B547  .  8BF8         MOV EDI,EAX
0047B679  .  89BD E0FDFFF MOV DWORD PTR SS:[EBP-220],EDI            0047B549  .  89BD E0FDFFF MOV DWORD PTR SS:[EBP-220],EDI
0047B67F  .  83FF FF      CMP EDI,-1                                0047B54F  .  83FF FF      CMP EDI,-1
0047B682  .v 75 1F        JNZ SHORT       0047B6A3                  0047B552  .v 75 1F        JNZ SHORT       0047B573
```

Figure 6-4: Side-by-side disassemblies of two versions of one game

My bot modifies the code at 0x047B542 (right), and I needed to find the corresponding code in the new version, which I discovered at 0x047B672 (left). This function call invokes a packet-parsing function when a packet has been received. In order to find this address originally (and by "originally," I mean about 100 updates previous), I figured out how the game's network protocol worked, set breakpoints on many network-related API calls, stepped through execution, and inspected data on the stack until I found something that looked similar to what I expected given my knowledge of the protocol.

I could have followed the same steps for each of the 100+ updates since then, but that would have been unnecessary. The code stayed relatively the same throughout the years, which let me use patterns from the old code to find that function call's address in the new code.

Now, consider this chunk of assembly code:

```
PUSH EDI
PUSH EAX
LEA EAX,DWORD PTR SS:[EBP-C]
MOV DWORD PTR FS:[0],EAX
MOV DWORD PTR SS:[EBP-10],ESP
MOV DWORD PTR SS:[EBP-220],-1
MOV DWORD PTR SS:[EBP-4],0
```

Does it look familiar? Compare it to Figure 5-4, and you'll see that this exact code exists right above the highlighted function call in both versions of the game. Regardless of what it does, the combination of operations looks pretty unique; because of the number of different offsets it's using relative to EBP, it's unlikely that an identical chunk of code exists in any other part of the binary.

Every time I have to update this address, I open the old binary in OllyDbg, highlight this chunk of operations, right-click, and select Asm2Clipboard ▸ Copy fixed asm to clipboard. Then, I open the new binary in OllyDbg, navigate to the CPU Window, press CTRL-S, paste the assembly code, and hit Find. In 9.5 cases out of 10, this places me directly above the function call I need to find in the new version.

When an update comes, you can use the same method to find nearly all of your known addresses. It should work for every address you can find easily in assembly code. There are a few caveats, though:

1. OllyDbg limits search to eight operations, so you must find code markers of that size or smaller.
2. The operations you use cannot contain any other addresses, as those addresses have likely changed.
3. If parts of the game have changed that use the address you're looking for, the code might be different.
4. If the game changes compilers or switches optimization settings, almost all code will be entirely different.

As "Automatically Find currentHealth and maxHealth" on page XX discusses, you can benefit from writing scripts that carry out these tasks for you. Serious game hackers work very hard to automatically locate as many addresses as possible, and some of the best bots are engineered to automatically detect their addresses at runtime, every time. It can be a lot of work initially, but the investment can definitely pay off.

Identifying Complex Structures in Game Data

Chapter 4 described how a game might store data in static structures. This knowledge will suffice when you're trying to find simple data, but it falls short for data that is stored through dynamic structures. This is because dynamic structures might be scattered in different memory locations, follow long pointer chains, or require complex algorithms to actually extract the data from them.

This section explores common dynamic structures you'll find in video game code, and how to read data from them once they're found. To begin, I'll talk about the underlying composition of each dynamic structure. Next, I'll outline the algorithms needed to read the data from these structures. (For simplicity, each algorithm discussion assumes you have a pointer to an instance of the structure as well as some way to read memory.) Lastly, I'll cover tips and tricks that can help you determine when a value you're searching for in memory is actually encapsulated in one of these structures, so you'll know when to apply this knowledge. I'll focus on C++, as its object-oriented nature and heavily used standard library are typically responsible for such structures.

NOTE *Some of these structures might differ slightly from machine to machine based on compilers, optimization settings, or standard library implementations, but the basic concepts will remain the same. Also, in the interest of brevity, I will be omitting irrelevant parts of these structures, such as custom allocators or comparison functions. Working example code can be found in* /GameHackingExamples/Chapter5_AdvancedMemoryForensics_Scanning *in this book's resource files.*

The std::string Class

Instances of `std::string` are among the most common culprits of dynamic storage. This class from the C++ *Standard Template Library (STL)* abstracts string operations away from the developer while preserving efficiency, making it widely used in all types of software. A video game might use `std::string` structure for any string data, such as creature names.

Examining the Structure of a std::string

When you strip away the member functions and other nondata components of the `std::string` class, this is the structure that remains:

```
class string {
    union {
        char* dataP;
        char dataA[16];
    };
    int length;
};

// point to a string in memory
string* _str = (string*)stringAddress;
```

The class reserves 16 characters that are presumably used to store the string in place. It also, however, declares that the first 4 bytes can be a pointer to a character. This might seem odd, but it's a result of optimization. At some point, the developers of this class decided that 15 characters (plus a null terminator) was a suitable length for many strings, and chose to save on memory allocations and frees by reserving 16 bytes of memory in advance. To accommodate longer strings, they allowed the first 4 bytes of this reserved memory to be used as a pointer to the characters of these longer strings.

NOTE *If the code were compiled to 64 bits, then it would actually be the first 8 (not 4) bytes that point to a character. Throughout this example, however, you can assume 32-bit addresses, and that* `int` *is the size of an address.*

Accessing string data this way takes some overhead. The function to locate the right buffer looks something like this:

```
const char* c_str() {
    if (_str->length <= 15)
        return (const char*)&_str->dataA[0];
    else
        return (const char*)_str->dataP;
}
```

The fact that a `std::string` can be either a complete string or a pointer to a longer string makes this particular structure quite tricky from a game hacking respective. Some games may use `std::string` to store strings that only rarely exceed 15 characters. When this is the case, you might implement bots that rely on these strings, never knowing that the underlying structure is in fact more complicated than a simple string.

Overlooking a std::string Can Ruin Your Fun

Not knowing the true nature of the structure containing the data you need can lead you to write a bot that works only some of the time, and fails when it counts. Imagine, for example, that you're trying to figure out how a game stores creature data. In your hypothetical search, you find that all the creatures in the game are stored in an array of structures that look something like Listing 5-3.

```
struct creatureInfo {
    int uniqueID;
    char name[16];
    int nameLength;
    int healthPercent;
    int xPosition;
    int yPosition;
    int modelID;
```

```
    int creatureType;
};
```

Listing 6-3: How you might interpret creature data found in memory

After scanning the creature data in memory, say you notice that the first 4 bytes of each structure are unique for each creature, so you call those bytes the `uniqueID` and assume they form a single `int` property. Looking further in the memory, you find that the creature's `name` is stored right after `uniqueID`, and after some deduction, you figure out the name is 16 bytes long. The next value you see in memory turns out to be the `nameLength`; it's a bit strange that a null-terminated string has an associated length, but you ignore that oddity and continue analyzing the data in memory. After further analysis, you determine what the remaining values are for, define the structure shown in Listing 5-3, and write a bot that automatically attacks creatures with certain names.

After weeks of testing your bot while hunting creatures with names like *Dragon*, *Cyclops*, *Giant*, and *Hound*, you decide it's time to give your bot to your friends. For the inaugural use, you gather everyone together to kill a boss named *Super Bossman Supreme*. The entire team sets the bot to attack the boss first and target lesser creatures like a *Demon* or *Grim Reaper* when the boss goes out of range.

Once your team arrives at the boss's dungeon. . .you're all slowly obliterated.

What went wrong in this scenario? Your game must be storing creature names with `std::string`, not just a simple character array. The `name` and `nameLength` fields in `creatureInfo` are, in fact, part of a `std::string` field, and the `name` character array is a union of `dataA` and `dataP` members. *Super Bossman Supreme* is longer than 15 characters, and because the bot was not aware of the `std::string` implementation, it didn't recognize the boss. Instead, it constantly retargeted summoned *Demon* creatures, effectively keeping you from targeting the boss while he slowly drained your health and supplies.

Determining Whether Data Is Stored in a std::string

Without knowing how the `std::string` class is structured, you'd have trouble tracking down bugs like the hypothetical one I just described. But pair what you've learned here with experience, and you can avoid these kinds of bugs entirely. When you find a string like `name` in memory, don't just assume it's stored in a simple array. To figure out whether a string is in fact a `std::string`, ask yourself these questions:

1. Why is the string length present for a null-terminated string? If you can't think of a good reason, then you may have a `std::string` on your hands.
2. Do some creatures (or other game elements, depending on what you're looking for) have names longer than 16 letters, but you find room for only 16 characters in memory? If so, the data is almost definitely stored in a `std::string`.

3. Is the name stored in place, requiring the developer to use `strcpy()` to modify it? It's probably a `std::string`, because working with raw C strings in this way is considered bad practice.

Finally, keep in mind that there is also a class called `std::wstring` that is used to store wide strings. The implementation is very similar, but `wchar_t` is used in place of every `char`.

The std::vector<T> Class

Games must keep track of many dynamic arrays of data, but managing dynamically sized arrays can be very tricky. For speed and flexibility, game developers often store such data using a templated STL class called `std::vector` instead of a simple array.

Examining the Structure of a std::vector

A declaration of this class looks something like Listing 5-4.

```
template<typename T>
class vector {
    T* begin;
    T* end;
    T* reservationEnd;
};
```

Listing 6-4: An abstracted `std::vector` object

This template adds an extra layer of abstraction, so I'll continue this description using a `std::vector` declared with the `DWORD` type. Here's how a game might declare that vector:

```
std::vector<DWORD> _vec;
```

Now, let's dissect what a `std::vector` of `DWORD` objects would look like in memory. If you had the address of `_vec` and shared the same memory space, you could re-create the underlying structure of the class and access `_vec` as shown in Listing 5-5.

```
class vector {
    DWORD* begin;
    DWORD* end;
    DWORD* tail;
};
// point to a vector in memory
vector* _vec = (vector*)vectorAddress;
```

Listing 6-5: A `DWORD std::vector` object

You can treat the member `begin` like a raw array, as it points to the first element in the `std::vector` object. There is no array length member, though,

so you must calculate the vector's length based on `begin` and `end`, which is an empty object following the final object in the array. The length calculation code looks like this:

```
int length() {
    return ((DWORD)_vec->end - (DWORD)_vec->begin) / sizeof(DWORD);
}
```

This function simply subtracts the address stored in `begin` from the address stored in `end` to find the number of bytes between them. Then, to calculate the number of objects, it divides the number of bytes by the number of bytes per object.

Using `begin` and this `length()` function, you can safely access elements in `_vec`. That code would look something like this:

```
DWORD at(int index) {
    if (index >= _vec->length())
        throw new std::out_of_range();
    return _vec->begin[index];
}
```

Given an index, this code will fetch an item from the vector. But if the index is greater than the vector's length, a `std::out_of_range` exception will be thrown. Adding values to a `std::vector` would be very expensive if the class couldn't reserve or reuse memory, though. To remedy this, the class implements a function called `reserve()` that tells the vector how many objects to leave room for.

The absolute size of a `std::vector` (its *capacity*) is determined through an additional pointer, which is called `tail` in the vector class we've re-created. The calculation for the capacity resembles the length calculation:

```
int capacity() {
    return ((DWORD)_vec->tail - (DWORD)_vec->begin) / sizeof(DWORD);
}
```

To find the capacity of a `std::vector`, instead of subtracting the `begin` address from the `end` address, as you would to calculate length, this function subtracts the `begin` address from `tail`. Additionally, you can use this calculation a third time to determine the number of free elements in the vector by using `tail` and `end` instead:

```
int freeSpace() {
    return ((DWORD)_vec->tail - (DWORD)_vec->end) / sizeof(DWORD);
}
```

Given proper memory reading and writing functions, you can use the declaration in Listing 5-4 and the calculations that follow to access and manipulate vectors in the memory of a game. Chapter 6 discusses reading memory in detail, but for now, let's look at ways you can determine if data you're interested in is stored in a `std::vector`.

Determining Whether Data Is Stored in a std::vector

Once you've found an array of data in a game's memory, there are a few steps you can follow to determine if it is stored in a `std::vector`. First, you can be sure that the array is not stored in a `std::vector` if it has a static address, because `std::vector` objects require pointer paths to access the underlying array. If the array *does* require a pointer path, having a final offset of `0` would indicate a `std::vector`. To confirm, you can change the final offset to `4` and check if it points to the final object in the array instead of the first one. If so, you're almost definitely looking at a vector, as you've just confirmed the `begin` and `end` pointers.

The std::list<T> Class

Similar to `std::vector`, `std::list` is a class that you can use to store a collection of items in a linked list. The main differences are that `std::list` doesn't require a contiguous storage space for elements, cannot directly access elements by their index, and can grow in size without affecting any previous elements. Due to the overhead required to access items, it is rare to see this class used in games, but it shows up in some special cases, which I'll discuss in this section.

Examining the Structure of a std::list

The `std::list` class looks something like Listing 5-6.

```
template<typename T>
class listItem {
    listItem<T>* next;
    listItem<T>* prev;
    T value;
};

template<typename T>
class list {
    listItem<T>* root;
    int size;
};
```

Listing 6-6: An abstracted `std::list` object

There are two classes here: `listItem` and `list`. To avoid extra abstraction while explaining how `std::list` works, I'll describe this object as it would look when the type is `DWORD`. Here's how a game would declare a `std::list` of the `DWORD` type:

```
std::list<DWORD> _lst;
```

Given that declaration, the `std::list` is structured like the code in Listing 5-7.

```
class listItem {
    listItem* next;
    listItem* prev;
    DWORD value;
};
class list {
    listItem* root;
    int size;
};
// point to a list
list* _lst = (list*)listAddress;
```

Listing 6-7: A `DWORD std::list` object

The class `list` represents the list header, while `listItem` represents a value stored in the list. Instead of being stored contiguously, the items in the list are stored independently. Each item contains a pointer to the item that comes after it (`next`) and the one that comes before it (`prev`), and these pointers are used to locate items in the list. The `root` item acts as a marker for the end of the list; the `next` pointer of the last item points to `root`, as does the `prev` pointer of the first item. The `root` item's `next` and `prev` pointers also point to the first item and the last item, respectively. Figure 5-5 shows what this looks like.

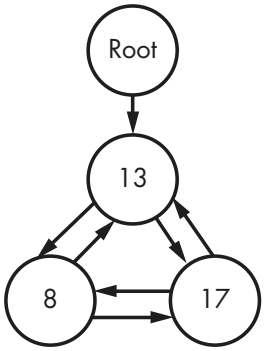


Figure 6-6: Visualizing a `std::map` as a circular linked list

Given this structure, you can use the following code to iterate over a `std::list` object:

```
// iterate forward
listItem* it = _lst->root->next;
for (; it != _lst->root; it = it->next)
    printf("Value is %d\n", it->value);

// iterate backward
listItem* it = _lst->root->prev;
for (; it != _lst->root; it = it->prev)
    printf("Value is %d\n", it->value);
```

The first loop starts at the first item (`root->next`) and iterates forward (`it = it->next`) until it hits the end marker (`root`). The second loop starts at the last item (`root->pres`) and iterates backward (`it = it->prev`) until it hits the end marker (`root`). This iteration relies on `next` and `prev` because unlike an array, the `std::list` objects are not contiguous. Since the memory of each object in a `std::list` is not contiguous, there's no quick-and-dirty way to calculate the size. Instead, the class just defines a `size` member. Additionally, the concept of reserving space for new objects is irrelevant for lists, so there's no variable or calculation to determine one's capacity.

Determining Whether Game Data Is Stored in a std::list

Identifying objects stored in the `std::list` class can be tricky, but there are a few hints you can watch for. First, items in a `std::list` cannot have static addresses, so if the data you seek has a static address, then you're in the clear. Items that are obviously part of a collection may, however, be part of a `std::list` if they're not contiguous in memory.

Also consider that objects in a `std::list` can have infinitely long pointer chains (think `it->prev->next->prev->next->prev` . . .), and pointer scanning for them in Cheat Engine will show many more results when No Looping Pointers is turned off.

You can also use a script to detect when a value is stored in a linked list. Listing 5-8 shows a Cheat Engine script that does just this.

```
function _verifyLinkedList(address)
    local nextItem = readInteger(address) or 0
    local previousItem = readInteger(address + 4) or 0
    local nextItemBack = readInteger(nextItem + 4)
    local previousItemForward = readInteger(previousItem)

    return (address == nextItemBack
            and address == previousItemForward)
end

function isValueInLinkedList(valueAddress)
    for address = valueAddress - 8, valueAddress - 48, -4 do
        if (_verifyLinkedList(address)) then
            return address
        end
    end
    return 0
end

local node = isValueInLinkedList(addressOfSomeValue)
if (node > 0) then
    print(string.format("Value in LL, top of node at 0x0%x", node))
end
```

Listing 6-8: Listing 5-8: Determining if data is in a `std::list` using a Cheat Engine Lua script

There's quite a bit of code here, but what it's doing is actually pretty simple. The `isValueInLinkedList()` function takes an address of some value, and then looks backward for up to 40 bytes (10 integer objects, in case the value is in some larger structure), starting 8 bytes above the address (two pointers must be present, and they are 4 bytes each). Because of memory alignment, this loop iterates in steps of 4 bytes.

On each iteration, the address is passed to the `_verifyLinkedList()` function, which is where the magic happens. If we look at it in terms of linked list structure as defined in this chapter, the function simply does this:

```
return (node->next->prev == node && node->prev->next == node)
```

That is, the function basically assumes the memory address it's given points to a linked list, and makes sure the supposed node has valid next and previous nodes. If the nodes are valid, the assumption was correct and the address is that of a linked list node. If the nodes don't exist or don't point to the right locations, the assumption was wrong and the address is not part of a linked list.

Keep in mind that this script won't give you the address of the list's root node, but simply the address of the node containing the value you've given it. To properly traverse a linked list, you'll need to scan for a valid pointer path to the root node, so you'll need its address.

Finding that address can require some searching of memory dumps, a lot of trial and error, and a ton of head scratching, but it's definitely possible. The best way to start is to follow the chain of `prev` and `next` nodes until you find a node with data that is either blank, nonsensical, or filled with the value `0xBAADF00D` (some, but not all, standard library implementations use this value to mark root nodes).

This investigation can also be made easier if you know exactly how many nodes are in the list. Even without the list header, you can determine the amount of nodes by continuously following the next pointer until you end up back at your starting node, as in Listing 5-9.

```
function countLinkedListNodes(nodeAddress)
    local counter = 0
    local next = readInteger(nodeAddress)
    while (next ~= nodeAddress) do
        counter = counter + 1
        next = readInteger(next)
    end
    return counter
end
```

Listing 6-9: Determining the size of an arbitrary `std::list` using a Cheat Engine Lua script

First, this function creates a counter to store the number of nodes and a variable to store the next node's address. The `while` loop then iterates over the nodes until it ends up back at the initial node. Finally, it returns the counter variable, which was incremented on every iteration of the loop.

FIND THE ROOT NODE WITH A SCRIPT

It's actually possible to write a script that can find the root node, but I'll leave it as an optional exercise for you. How does it work? Well, the root node must be in the chain of nodes, the list header points to the root, and the size of the list will immediately follow the root in memory. Given this information, you can write a script that will search for any memory containing a pointer to one of the list's nodes, followed by the size of the list. More often than not, this piece of memory is the list header, and the node it points to is the root node.

The std::map<T, T> Class

Like a `std::list`, a `std::map` uses links between elements to form its structure. Unique to `std::map`, however, is the fact that each element stores two pieces of data (a key and a value), and sorting the elements is an inherent property of the underlying data structure: a red-black tree. The following code shows the structures that compose a `std::map`.

```
template<typename keyT, typename valT>
struct mapItem {
    mapItem<keyT, valT>* left;
    mapItem<keyT, valT>* parent;
    mapItem<keyT, valT>* right;
    keyT key;
    valT value;
};

template<typename keyT, typename valT>
struct map {
    DWORD irrelevant;
    mapItem<keyT, valT>* rootNode;
    int size;
}
```

A red-black tree is a self-balancing binary search tree, so a `std::map` is, too. In the STL's `std::map` implementation, each element (or node) in the tree has three pointers: `left`, `parent`, and `right`. In addition to the pointers, each node also has a `key` and a `value`. The nodes are arranged in the tree based on a comparison between their keys. The `left` pointer of a node points to a node with a smaller key, and the `right` pointer points to a node with a larger key. The `parent` points to the upper node. The first node in the tree is called the `rootNode`, and nodes that lack children point to it.

Visualizing a std::map

Figure 5-6 shows a `std::map` that has the keys 1, 6, 8, 11, 13, 15, 17, 22, 25, and 27.

The top node (holding the value `13`) is pointed to by the `parent` of `rootNode`. The top node will always have the median `key`. Everything to the left of it has a smaller `key`, and everything to the right has a greater `key`. This is true for any node in the tree, and this truth enables efficient key-based search. While not represented in the image, the left pointer of the root node will point to the leftmost node (`1`) and the right pointer will point to the rightmost node (`27`).

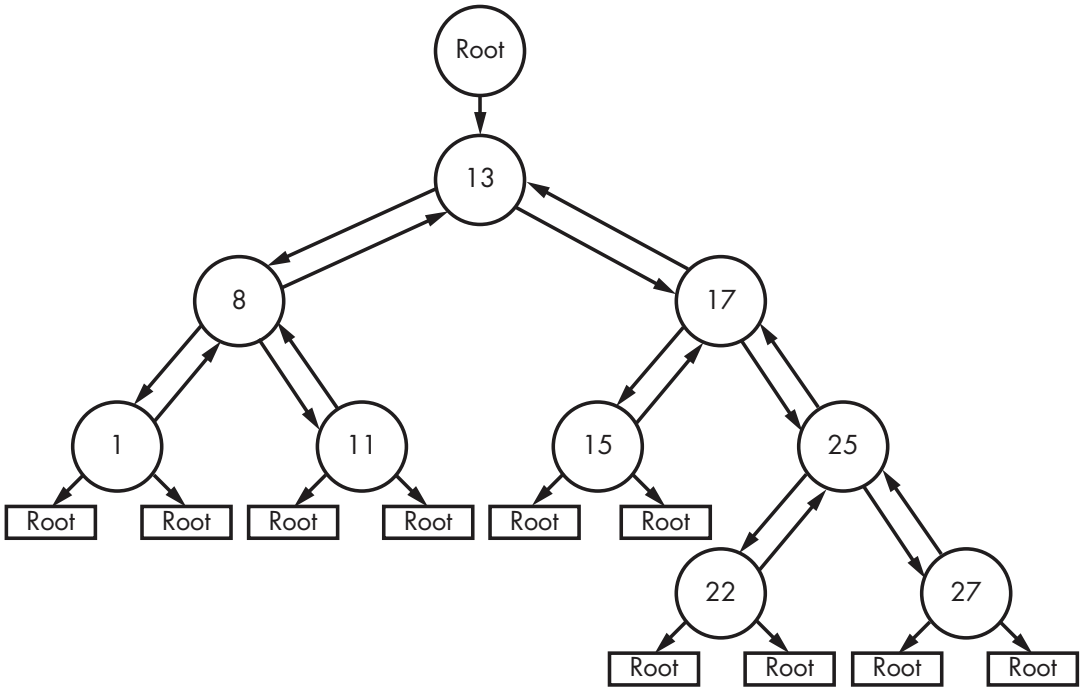


Figure 6-5: A red-black tree

Accessing Data in a std::map

Once again, I'll use a static `std::map` definition when discussing how to extract data from the structure. Since the template takes two types, I'll also use some pseudotypes to keep things obvious. Here's the declaration for the `std::map` object I'll reference for the rest of the section:

```
typedef int keyInt;
typedef int valInt;
std::map<keyInt, valInt> myMap;
```

With this declaration, the structure of `myMap` becomes:

```
struct mapItem {
    mapItem* left;
    mapItem* parent;
    mapItem* right;
    keyInt key;
    valInt value;
};
struct map {
    DWORD irrelevant;
    mapItem* rootNode;
    int size;
}
map* _map = (map*)mapAddress;
```

There are some important algorithms that you might need to access the data in a `std::map` structure in a game. First, blindly iterating over every item in the map can be useful if you just want to see all of the data. To do this sequentially, you could write an iteration function like this:

```
void iterateMap(mapItem* node) {
    if (node == _map->rootNode) return;
    iterateMap(node->left);
    printNode(node);
    iterateMap(node->right);
}
```

A function to iterate over an entire map would first read the current node and check whether it's the `rootNode`. If not, it would recurse left, print the node, and recurse right.

To call this function, you'd have to pass a pointer to the `rootNode` as follows:

```
iterateMap(_map->rootNode->parent);
```

The purpose of a `std::map`, however, is to store keyed data in a quickly searchable way. When you need to locate a node given a specific `key`, mimicking the internal search algorithm is preferable to scanning the entire tree. The code for searching a `std::map` looks something like this:

```
mapItem* findItem(keyInt key, mapItem* node) {
    if (node != _map->rootNode) {
        if (key == node->key)
            return node;
        else if (key < node->key)
            return findItem(key, node->left);
        else
            return findItem(key, node->right);
    } else return NULL;
}
```

Starting at the top of the tree, you'd simply recurse left if the current key is greater than the search key, and recurse right if it is smaller. If the keys are equal, you'd return the current node. If you reach the bottom of the tree and don't find the key, you'd return `NULL` because the key isn't stored in the map.

Here's one way you might use this `findItem()` function:

```
mapItem* ret = findItem(someKey, _map->rootNode->parent);
if (ret)
    printNode(ret);
```

As long as `findItem()` doesn't return `NULL`, this code should print a node from `_map`.

Determining Whether Game Data Is Stored in a std::map

Typically, I don't even consider whether data could be in a `std::map` until I know the collection is not an array, a `std::vector`, or a `std::list`. If you rule out all three options, then as with a `std::list`, you can look at the three integer values before the value and check if they point to memory that could possibly be other map nodes.

Once again, this can be done with a Lua script in Cheat Engine. The script is similar to the one I showed for lists, looping backward over memory to see if a valid node structure is found before the value. Unlike the list code, though, the function that verifies a node is much trickier. Take a look at the code in Listing 5-10, and then I'll dissect it.

```
function _verifyMap(address)
    local parentItem = readInteger(address + 4) or 0

    local parentLeftItem = readInteger(parentItem + 0) or 0
    local parentRightItem = readInteger(parentItem + 8) or 0

❶   local validParent =
        parentLeftItem == address
        or parentRightItem == address
    if (not validParent) then return false end

    local tries = 0
    local lastChecked = parentItem
    local parentsParent = readInteger(parentItem + 4) or 0
❷   while (readInteger(parentsParent + 4) ~= lastChecked and tries < 200) do
        tries = tries + 1
        lastChecked = parentsParent
        parentsParent = readInteger(parentsParent + 4) or 0
    end

    return readInteger(parentsParent + 4) == lastChecked
end
```

Listing 6-10: Determining if data is in a `std::map` using a Cheat Engine Lua script

Given `address`, this function checks if `address` is in a map structure. It first checks if there's a valid parent node and, if so, it checks whether that parent node points to `address` on either side ❶. But this check isn't enough. If the check passes, the function will also climb up the line of `parent` nodes until it reaches a node that is the parent of its own parent ❷, trying 200 times before calling it quits. If the climb succeeds in finding a node that is its own grandparent, then `address` definitely points to a map node. This works because, as I outlined in "Visualizing a std::map" on page 114 , at the top of every map is a root node whose parent points to the first node in the tree, and that node's parent points back to the root node.

NOTE *I bet you didn't expect to run into the grandfather paradox from time travel when reading a game hacking book!*

Using this function and a slightly modified back-tracking loop from Listing 5-8, you can automatically detect when a value is inside a map:

```
function isValueInMap(valueAddress)
    for address = valueAddress - 12, valueAddress - 52, -4 do
        if (_verifyMap(address)) then
            return address
        end
    end
    return 0
end

local node = isValueInMap(addressOfSomeValue)
if (node > 0) then
    print(string.format("Value in map, top of node at 0x0%x", node))
end
```

Aside from function names, the only change in this code from Listing 5-8 is that it starts looping 12 bytes before the value instead of 8, because a map has three pointers instead of the two in a list. One good consequence of a map's structure is that it's easy to obtain the root node. When the `_verifyMap` function returns true, the `parentsParent` variable will contain the address of the root node. With some simple modifications, you could return this to the main call and have everything you need to read the data from a `std::map` in one place.

Closing Thoughts

Memory forensics is the most time-consuming part of hacking games, and its obstacles can appear in all shapes and sizes. Using purpose, patterns, and a deep understanding of complex data structures, however, you can quickly overcome these obstacles. If you're still a bit confused about what's going on, make sure to download and play with the example code provided, as it contains proofs of concept for all of the algorithms covered in this chapter.

In the next chapter, we'll start diving in to the code you need to read and write a game's memory from your own programs, so you can take the first step in putting to work all of this information about memory structures, addresses, and data.