

# Взлом игр

*Разработка  
автономных ботов  
для онлайн игр*



# **ВЗЛОМ ИГР**

**Разработка автономных  
ботов для онлайн-игр**

**Ник Кано**



**no starch  
press**

Сан-Франциско

## **ОБ АВТОРЕ**

Ник Кано написал свои первые скрипты для игровых серверов с открытым исходным кодом, когда ему было 12 лет, и начал бизнес по продаже своих ботов, когда ему было 16 лет. С тех пор он является частью сообщества хакеров и консультирует разработчиков и дизайнеров игр по лучшим практикам защиты своих игр от ботов. Ник также имеет многолетний опыт в обнаружении и защите от вредоносных программ, и он говорил на многих конференциях о своих исследованиях и инструментах.

## **Технический рецензент**

Стивен Лоулер - основатель и президент небольшой консалтинговой фирмы по компьютерному программному обеспечению и безопасности. Он активно работает в области информационной безопасности более 10 лет, в основном в области реверс-инжиниринга, анализа вредоносных программ и исследований уязвимостей. Он был членом команды по анализу вредоносных программ Mandiant и помогал в громких компьютерных вторжениях, затрагивающих несколько компаний из списка Fortune 100. Стивен также разработал и преподает практический курс по эксплуатации ARM, который предлагался на BlackHat и нескольких других конференциях по безопасности в течение последних пяти лет

# Краткое содержание

## Оглавление

Предисловие .....	11
Благодарность .....	12
Введение .....	13
Необходимые условия для чтения .....	13
Краткая история взлома игр .....	14
Зачем взламывать игры? .....	15
Как организована эта книга .....	16
Об онлайн-ресурсах .....	17
Как использовать эту книгу .....	18
Часть 1: Инструменты ремесла .....	19
1) Сканирование памяти с использованием Cheat Engine.....	20
Почему важны сканеры памяти .....	20
Сканирование памяти .....	21
Сканер памяти Cheat Engine .....	21
Типы сканирования .....	22
Запуск первого сканирования .....	23
Следующие сканы .....	23
Когда невозможно получить единственный результат .....	24
Таблицы читов .....	25
Модификация памяти в играх .....	25
Ручная модификация с помощью Cheat Engine .....	26
Генератор трейнеров .....	26
Сканирование указателя .....	29
Цепочки указателей .....	29
Основы сканирования с помощью указателя .....	30
pointerScan() .....	31
rScan() .....	32
Сканирование указателей с помощью Cheat Engine .....	32
Ключевые параметры .....	33
Опции, зависящие от ситуации .....	34
Повторное сканирование указателя .....	36
Среда сценариев Lua .....	38
Поиск шаблонов в ассемблере .....	38
Поиск строк .....	40
Заключительные мысли .....	42
2) Отладка игр с помощью OllyDbg .....	43
Краткий обзор пользовательского интерфейса OllyDbg .....	44
Окно CPU в OllyDbg .....	47
Просмотр и навигация по ассемблерному коду игры .....	47
Просмотр и редактирование содержимого регистров (Viewing and Editing Register Contents) .....	50
Просмотр и поиск памяти игры (Viewing and Searching a Game's Memory) ..	50
Просмотр стека вызовов игры (Viewing a Game's Call Stack) .....	51
Многоклиентское патчинг (Multiclient Patching) .....	51
Создание патчей кода (Creating Code Patches) .....	52
Внимание к размеру патча .....	52
Определение размера патча (Determining Patch Size) .....	54
Выражения в OllyDbg (OllyDbg's Expression Engine) .....	55

<b>Использование выражений в точках останова (Using Expressions in Breakpoints) .....</b>	<b>56</b>
<b>Использование операторов в движке выражений (Using Operators in the Expression Engine).....</b>	<b>57</b>
<b>Работа с базовыми элементами выражений (Working with Basic Expression Elements).....</b>	<b>58</b>
<b>Доступ к содержимому памяти с помощью выражений (Accessing Memory Contents with Expressions).....</b>	<b>59</b>
<b>Выражения OllyDbg в действии (OllyDbg Expressions in Action) .....</b>	<b>60</b>
<b>Приостановка выполнения при выводе имени конкретного игрока (Pausing Execution When a Specific Player's Name Is Printed).....</b>	<b>60</b>
<b>Определение места для паузы (Figuring Out Where to Pause) .....</b>	<b>60</b>
<b>Создание условной точки останова (Crafting the Conditional Breakpoint)....</b>	<b>61</b>
<b>Приостановка выполнения при снижении здоровья персонажа (Pausing Execution When Your Character's Health Drops) .....</b>	<b>62</b>
<b>Написание выражения для проверки здоровья (Writing an Expression to Check Health).....</b>	<b>63</b>
<b>Определение места для начала трассировки (Figuring Out Where to Start the Trace) .....</b>	<b>64</b>
<b>Активация трассировки (Activating the Trace).....</b>	<b>65</b>
<b>Плагины OllyDbg для игровых хакеров (OllyDbg Plug-ins for Game Hackers)</b>	<b>66</b>
<b>Копирование ассемблерного кода с помощью Asm2Clipboard .....</b>	<b>67</b>
<b>Добавление Cheat Engine в OllyDbg с помощью Cheat Utility (Adding Cheat Engine to OllyDbg with Cheat Utility) .....</b>	<b>67</b>
<b>Управление OllyDbg через командную строку (Controlling OllyDbg Through the Command Line) .....</b>	<b>68</b>
<b>Визуализация потока управления с OllyFlow (Visualizing Control Flow with OllyFlow).....</b>	<b>69</b>
<b>Заключительные мысли (Closing Thoughts) .....</b>	<b>72</b>
<b>3) Разведка с использованием Process Monitor и Process Explorer .....</b>	<b>73</b>
<b>Process Monitor .....</b>	<b>73</b>
<b>Логирование внутриигровых событий.....</b>	<b>74</b>
<b>Анализ событий в журнале Process Monitor .....</b>	<b>76</b>
<b>Отладка игры для сбора дополнительных данных.....</b>	<b>78</b>
<b>Process Explorer .....</b>	<b>80</b>
<b>Параметры управления дескрипторами (Handle Manipulation Options) .....</b>	<b>84</b>
<b>Закрытие мьютексов (Closing Mutexes).....</b>	<b>85</b>
<b>Анализ доступа к файлам (Inspecting File Accesses).....</b>	<b>86</b>
<b>Заключительные мысли (Closing Thoughts) .....</b>	<b>87</b>
<b>Часть 2: Анализ игры.....</b>	<b>88</b>
<b>4) От кода в памяти: общий вводный курс .....</b>	<b>89</b>
<b>Как переменные и другие данные отображаются в памяти (How Variables and Other Data Manifest in Memory) .....</b>	<b>90</b>
<b>Числовые данные (Numeric Data) .....</b>	<b>91</b>
<b>Строковые данные (String Data) .....</b>	<b>94</b>
<b>Структуры данных (Data Structures) .....</b>	<b>96</b>
<b>Порядок элементов в структуре и выравнивание (Structure Element Order and Alignment) .....</b>	<b>96</b>
<b>Как работают структуры.....</b>	<b>98</b>
<b>Unions (Объединения) .....</b>	<b>98</b>
<b>Класс с виртуальными функциями (A Class with Virtual Functions) .....</b>	<b>100</b>
<b>Экземпляры классов и таблицы виртуальных функций (Class Instances and Virtual Function Tables).....</b>	<b>101</b>

<b>Интенсивный курс по ассемблеру x86 (x86 Assembly Crash Course) .....</b>	<b>103</b>
<b>Инструкции (Instructions).....</b>	<b>104</b>
<b>Команды ассемблера (Assembly Commands) .....</b>	<b>106</b>
<b>Регистры процессора (Processor Registers) .....</b>	<b>106</b>
<b>Общие регистры (General Registers).....</b>	<b>107</b>
<b>Индексные регистры (Index Registers).....</b>	<b>108</b>
<b>Регистр индекса выполнения (The Execution Index Register).....</b>	<b>109</b>
<b>Регистр EFLAGS (The EFLAGS Register) .....</b>	<b>109</b>
<b>Регистры сегментов (Segment Registers).....</b>	<b>110</b>
<b>Стек вызовов .....</b>	<b>112</b>
<b>Структура .....</b>	<b>112</b>
<b>Кадр стека (The Stack Frame) .....</b>	<b>113</b>
<b>Важные инструкции x86 для взлома игр .....</b>	<b>115</b>
<b>Изменение данных (Data Modification) .....</b>	<b>115</b>
<b>Арифметика (Arithmetic).....</b>	<b>116</b>
<b>Бинарные инструкции (Binary Instructions) .....</b>	<b>116</b>
<b>Разветвление (Branching) .....</b>	<b>119</b>
<b>Вызовы функций .....</b>	<b>121</b>
<b>Заключительные мысли (Closing Thoughts) .....</b>	<b>123</b>
<b>5) Продвинутый анализ памяти.....</b>	<b>124</b>
<b>Продвинутое сканирование памяти (Advanced Memory Scanning) .....</b>	<b>125</b>
<b>Определение назначения (Deducing Purpose) .....</b>	<b>125</b>
<b>Поиск здоровья игрока с помощью OllyDbg .....</b>	<b>127</b>
<b>Определение новых адресов после обновлений игры.....</b>	<b>129</b>
<b>Выявление сложных структур в игровых данных .....</b>	<b>132</b>
<b>The std::string Class .....</b>	<b>133</b>
<b>Изучение структуры строки std::string .....</b>	<b>133</b>
<b>Непонимание структуры std::string может испортить вам удовольствие ...</b>	<b>134</b>
<b>Определение, хранятся ли данные в std::string.....</b>	<b>135</b>
<b>Класс std::vector .....</b>	<b>136</b>
<b>Определение хранения данных в std::vector .....</b>	<b>138</b>
<b>Класс std::list .....</b>	<b>138</b>
<b>Исследование структуры std::list.....</b>	<b>138</b>
<b>Определение, хранится ли игровая информация в std::list.....</b>	<b>140</b>
<b>Класс std::map.....</b>	<b>143</b>
<b>Доступ к данным в std::map .....</b>	<b>144</b>
<b>Определение хранения игровых данных в std::map.....</b>	<b>146</b>
<b>Заключительные мысли .....</b>	<b>148</b>
<b>6) Чтение из и запись в память. Запись в память игры.....</b>	<b>149</b>
<b>Получение идентификатора процесса игры .....</b>	<b>149</b>
<b>Получение дескрипторов процессов .....</b>	<b>151</b>
<b>Работа с OpenProcess() .....</b>	<b>151</b>
<b>Доступ к памяти .....</b>	<b>152</b>
<b>Работа с ReadProcessMemory() и WriteProcessMemory() .....</b>	<b>152</b>
<b>Доступ к значению в памяти с использованием ReadProcessMemory() и WriteProcessMemory() .....</b>	<b>153</b>
<b>Запись шаблонных функций доступа к памяти .....</b>	<b>154</b>
<b>Защита памяти .....</b>	<b>155</b>
<b>Различие атрибутов защиты памяти в x86 Windows .....</b>	<b>155</b>
<b>Различие атрибутов защиты памяти в x86 Windows .....</b>	<b>155</b>
<b>СПЕЦИАЛЬНЫЕ ТИПЫ ЗАЩИТЫ.....</b>	<b>157</b>
<b>Изменение защиты памяти .....</b>	<b>157</b>
<b>Рандомизация размещения адресного пространства (ASLR).....</b>	<b>159</b>
<b>Отключение ASLR для упрощения разработки ботов .....</b>	<b>160</b>

<b>Обход ASLR в продакшене .....</b>	<b>160</b>
<b>Часть 3: Управление процессами .....</b>	<b>163</b>
<b>7) Инъекция кода .....</b>	<b>164</b>
<b>Внедрение кодовых пещер (<i>code caves</i>) с инъекцией потока.....</b>	<b>164</b>
<b>Создание ассемблерной кодовой пещеры .....</b>	<b>165</b>
<b>Перевод ассемблера в шеллкод .....</b>	<b>166</b>
<b>Запись кодовой пещеры в память .....</b>	<b>167</b>
<b>Использование инъекции потока для выполнения кодовой пещеры.....</b>	<b>169</b>
<b>Захват основного потока игры для выполнения кодовых пещер.....</b>	<b>170</b>
<b>Создание кодовой пещеры на ассемблере .....</b>	<b>170</b>
<b>Генерация базового шелл-кода и выделение памяти .....</b>	<b>172</b>
<b>Нахождение и заморозка основного потока.....</b>	<b>173</b>
<b>Инъекция DLL для полного контроля .....</b>	<b>175</b>
<b>Коварный способ заставить процесс загрузить вашу DLL .....</b>	<b>175</b>
<b>Обман процесса для загрузки вашей DLL .....</b>	<b>175</b>
<b>Доступ к памяти во внедренной DLL.....</b>	<b>178</b>
<b>Обход ASLR во внедренной DLL .....</b>	<b>179</b>
<b>Заключительные мысли .....</b>	<b>180</b>
<b>8) Манипулирование потоком управления в игре .....</b>	<b>181</b>
<b>NOPing для удаления ненужного кода .....</b>	<b>181</b>
<b>Когда применять NOP .....</b>	<b>182</b>
<b>Как использовать NOP .....</b>	<b>183</b>
<b>ПРАКТИКА NOP-ИНГА .....</b>	<b>184</b>
<b>Перехват для перенаправления выполнения кода игры.....</b>	<b>185</b>
<b>Перехват вызовов (Call Hooking).....</b>	<b>185</b>
<b>Работа с Near Call в памяти .....</b>	<b>186</b>
<b>Перехват ближнего вызова .....</b>	<b>186</b>
<b>Очистка стека.....</b>	<b>187</b>
<b>Написание перехвата вызова .....</b>	<b>188</b>
<b>Перехват таблицы виртуальных функций (VF Table Hooking).....</b>	<b>189</b>
<b>Написание перехвата таблицы VF .....</b>	<b>190</b>
<b>Написание перехвата таблицы VF .....</b>	<b>190</b>
<b>Использование перехвата VF-таблицы .....</b>	<b>192</b>
<b>IAT Hooking .....</b>	<b>194</b>
<b>Оплата за переносимость (Paying for Portability).....</b>	<b>194</b>
<b>Проход по IAT .....</b>	<b>196</b>
<b>Размещение IAT-хука.....</b>	<b>197</b>
<b>Использование IAT-хука для синхронизации с потоком игры.....</b>	<b>199</b>
<b>Почему это работает? .....</b>	<b>200</b>
<b>Перехват прыжком (Jump Hooking) .....</b>	<b>201</b>
<b>Размещение прыжка.....</b>	<b>201</b>
<b>Написание функции трамплина .....</b>	<b>203</b>
<b>Завершение перехвата прыжка .....</b>	<b>204</b>
<b>Применение перехватов вызовов (Call Hooks) к Adobe AIR .....</b>	<b>205</b>
<b>Доступ к золотой жиле RTMP (Accessing the RTMP Goldmine) .....</b>	<b>205</b>
<b>Хук на функцию encode() в RTMPS .....</b>	<b>207</b>
<b>Перехват функции RTMPS decode() .....</b>	<b>209</b>
<b>Размещение хуков (Placing the Hooks) .....</b>	<b>210</b>
<b>Применение Jump-хуков и VF-хуков к Direct3D (Applying Jump Hooks and VF Hooks to Direct3D) .....</b>	<b>212</b>
<b>Цикл отрисовки (The Drawing Loop).....</b>	<b>213</b>
<b>Нахождение устройства Direct3D (Finding the Direct3D Device) .....</b>	<b>215</b>
<b>Перехват EndScene() с помощью Jump Hook (Jump Hooking EndScene()) ..</b>	<b>215</b>

<b>Размещение и удаление Jump хука (Placing and Removing the Jump Hook)</b>	217
Написание обратного вызова и "трамплина" (Writing the Callback and Trampoline) .....	218
Написание хука для <i>EndScene()</i> (Writing a Hook for <i>EndScene()</i> ) .....	220
Написание хука для <i>Reset()</i> (Writing a Hook for <i>Reset()</i> ).....	221
Что дальше? (What's Next?) .....	222
Заключительные мысли (Closing Thoughts) .....	223
Часть 4: Создание бота .....	224
<b>9) Использование экстрасенсорного восприятия для устранения тумана войны</b>	225
Основные знания ( <i>Background Knowledge</i> ) .....	226
Раскрытие скрытых деталей с помощью Lighthacks (Revealing Hidden Details with Lighthacks) .....	226
Добавление центрального источника окружающего света (Adding a Central Ambient Light Source) .....	226
Увеличение абсолютного окружающего света (Increasing the Absolute Ambient Light) .....	227
Создание других типов Lighthacks (Creating Other Types of Lighthacks)....	228
Обнаружение скрытых врагов с помощью Wallhacks (Revealing Sneaky Enemies with Wallhacks).....	229
Рендеринг с Z-буферизацией ( <i>Rendering with Z-Buffering</i> ) .....	229
Дословный перевод:.....	231
Создание <i>Direct3D Wallhack</i> ( <i>Creating a Direct3D Wallhack</i> ) .....	231
Переключение Z-буферизации ( <i>Toggling Z-Buffering</i> ) .....	231
Изменение текстуры врага ( <i>Changing an Enemy Texture</i> ) .....	232
Фингерпринтинг модели, которую вы хотите раскрыть ( <i>Fingerprinting the Model You Want to Reveal</i> ) .....	232
Использование NOPing Zoomhacks .....	234
Поверхностный взгляд на Hooking Zoomhacks ( <i>Scratching the Surface of Hooking Zoomhacks</i> ) .....	235
Отображение скрытых данных с помощью HUD ( <i>Displaying Hidden Data with HUDs</i> ).....	235
Создание HUD опыта ( <i>Creating an Experience HUD</i> ).....	236
Использование хуков для поиска данных ( <i>Using Hooks to Locate Data</i> ) ...	237
Обзор других ESP-хаков .....	238
Заключительные мысли .....	239
<b>10) Отзывчивые хаки.....</b>	240
Наблюдение за игровыми событиями .....	240
Мониторинг памяти.....	240
Обнаружение визуальных подсказок ( <i>Detecting Visual Cues</i> ) .....	242
Перехват сетевого трафика ( <i>Intercepting Network Traffic</i> ) .....	243
Обычная функция разбора пакетов ( <i>A Typical Packet-Parsing Function</i> )....	244
Точный код для каждой игры может отличаться .....	244
Несколько хитростей для поиска функции парсинга .....	245
Более сложный парсер.....	245
Гибридная система парсинга ( <i>A Hybrid Parsing System</i> ) .....	247
Взлом парсера ( <i>A Parser Hack</i> ) .....	249
Выполнение игровых действий ( <i>Performing In-Game Actions</i> ) .....	249
Эмуляция клавиатуры ( <i>Emulating the Keyboard</i> ) .....	250
Функция <i>SendInput()</i> ( <i>The SendInput() Function</i> ) .....	250
Функция <i>SendMessage()</i> .....	251
Отправка пакетов ( <i>Sending Packets</i> ) .....	254
Связываем все воедино .....	258

<b>Создание идеального целителя</b> .....	<b>258</b>
<b>Сопротивление атакам контроля толпы (Crowd-Control)</b> .....	<b>258</b>
<b>Избегание напрасной траты маны (Avoiding Wasted Mana)</b> .....	<b>259</b>
<b>Заключительные мысли (Closing Thoughts) .....</b>	<b>260</b>
<b>11) Собираем всё воедино: написание автономных ботов .....</b>	<b>261</b>
<b>Теория управления и взлом игр.....</b>	<b>261</b>
<b>Конечные автоматы.....</b>	<b>262</b>
<b>Объединение теории управления и конечных автоматов.....</b>	<b>265</b>
<b>Простой конечный автомат целителя.....</b>	<b>265</b>
<b>Гипотетическая сложная конечная машина .....</b>	<b>268</b>
<b>Исправление ошибок .....</b>	<b>270</b>
<b>Корректировка для постоянного коэффициента .....</b>	<b>271</b>
<b>Реализация адаптируемой коррекции ошибок .....</b>	<b>271</b>
<b>Поиск пути с алгоритмами поиска .....</b>	<b>274</b>
<b>Два распространенных метода поиска.....</b>	<b>274</b>
<b>Как препятствия нарушают поиск.....</b>	<b>276</b>
<b>Алгоритм поиска A*</b> .....	<b>276</b>
<b>Создание узла A*</b> .....	<b>277</b>
<b>Написание функции поиска A*</b> .....	<b>280</b>
<b>Создание списка пути.....</b>	<b>282</b>
<b>Когда A* поиск особенно полезен.....</b>	<b>283</b>
<b>Предсказание движений врага .....</b>	<b>283</b>
<b>Распространенные и крутые автоматизированные хаки .....</b>	<b>284</b>
<b>Грабеж с пещерными ботами.....</b>	<b>284</b>
<b>Депонирование золота и пополнение запасов .....</b>	<b>285</b>
<b>Использование персонажа в качестве приманки .....</b>	<b>286</b>
<b>Позволение игрокам писать скрипты для пользовательского поведения</b> <b>286</b>	
<b>Автоматизация боя с варботами .....</b>	<b>287</b>
<b>Autowall Bots (Боты-автостены)</b> .....	<b>287</b>
<b>Autosnipe Bots (Боты-автоснайперы)</b> .....	<b>288</b>
<b>Autokite Bots (Боты-автокайтеры)</b> .....	<b>288</b>
<b>Заключительные мысли (Closing Thoughts) .....</b>	<b>288</b>
<b>12) Оставаться скрытым .....</b>	<b>289</b>
<b>Известное программное обеспечение для защиты от читов (Prominent Anti-Cheat Software)</b> .....	<b>289</b>
<b>Набор инструментов PunkBuster (The PunkBuster Toolkit)</b> .....	<b>290</b>
<b>Обнаружение по сигнатурам (Signature-Based Detection)</b> .....	<b>290</b>
<b>Скриншоты (Screenshots).....</b>	<b>291</b>
<b>Проверка хэша (Hash Validation)</b> .....	<b>291</b>
<b>Набор инструментов ESEA Anti-Cheat (The ESEA Anti-Cheat Toolkit)</b> .....	<b>291</b>
<b>Набор инструментов VAC (The VAC Toolkit)</b> .....	<b>292</b>
<b>Сканирование кеша DNS (DNS Cache Scans)</b> .....	<b>292</b>
<b>Проверка бинарных файлов (Binary Validation)</b> .....	<b>292</b>
<b>Ложные срабатывания (False Positives).....</b>	<b>292</b>
<b>Набор инструментов GameGuard (The GameGuard Toolkit)</b> .....	<b>293</b>
<b>Руткит на уровне пользователя (User-Mode Rootkit)</b> .....	<b>293</b>
<b>Руткит на уровне ядра (Kernel-Mode Rootkit)</b> .....	<b>294</b>
<b>Набор инструментов Warden (The Warden Toolkit)</b> .....	<b>294</b>
<b>Тщательное управление следами бота (Carefully Managing a Bot's Footprint)</b> .....	<b>295</b>
<b>Минимизация следа бота (Minimizing a Bot's Footprint)</b> .....	<b>295</b>
<b>Скрытие следов (Masking Your Footprint)</b> .....	<b>296</b>
<b>Обучение бота обнаружению отладчиков (Teaching a Bot to Detect Debuggers)</b> .....	<b>296</b>

<b>Вызов CheckRemoteDebuggerPresent() (Calling</b>	<b>296</b>
CheckRemoteDebuggerPresent()) .....	296
<b>Проверка обработчиков прерываний (Checking for Interrupt Handlers) .....</b>	<b>297</b>
<b>Проверка аппаратных точек останова (Checking for Hardware Breakpoints)</b>	<b>298</b>
.....	298
<b>Вывод отладочных строк (Printing Debug Strings) .....</b>	<b>298</b>
<b>Проверка обработчиков DBG_RIPEXCEPTION (Checking for</b>	<b>298</b>
DBG_RIPEXCEPTION Handlers) .....	298
<b>Измерение времени выполнения критически важных функций (Timing</b>	<b>299</b>
Control-Critical Routines) .....	299
<b>Проверка отладочных драйверов (Checking for Debug Drivers) .....</b>	<b>299</b>
<b>Техники защиты от отладки (Anti-Debugging Techniques).....</b>	<b>300</b>
<b>Создание неизбежного бесконечного цикла (Causing an Unavoidable Infinite</b>	<b>301</b>
Loop) .....	301
<b>Переполнение стека (Overflowing the Stack).....</b>	<b>301</b>
<b>Вызов BSOD (Causing a BSOD) .....</b>	<b>302</b>
<b>Обход детекции на основе сигнатур (Defeating Signature-Based Detection)</b>	<b>303</b>
.....	303
<b>Обход детекции на основе сигнатур (Defeating Signature-Based Detection)</b>	<b>304</b>
.....	304
<b>Обход детекции через скриншоты (Defeating Screenshots).....</b>	<b>305</b>
<b>Обход проверки бинарных файлов (Defeating Binary Validation) .....</b>	<b>306</b>
<b>Обход руткита античита (Defeating an Anti-Cheat Rootkit).....</b>	<b>309</b>
<b>Обход эвристического анализа (Defeating Heuristics) .....</b>	<b>310</b>
<b>Заключительные мысли (Closing Thoughts) .....</b>	<b>311</b>

## **Предисловие**

Ник замечательный. Как вы можете себе представить, мы сразу же нашли общий язык, как в хорошем, так и в плохом смысле этого слова. Я давно работаю в сфере безопасности, он немного моложе. У меня было школьное образование, в то время как он не очень любит колледж. Я верующий человек, а он - нет. Самое интересное, что все это не имеет значения: мы все равно получили огромное удовольствие. Возраст, раса, пол, степень - когда речь идет об играх, хакинге и кодинге, это никого не волнует!

Ник все сделает. Он веселый. Он великолепен. Он трудолюбив. И, пожалуй, самое важное: он один из немногих, кто понимает взаимосвязь игр, хакинга и кодинга. Он работал в этой нише и создавал прибыльных ботов.

В этой первой в своем роде книге Ник расскажет вам, что значит разбирать игры на части. Он научит вас инструментам для изучения программного обеспечения и хитростям профессии. Вы узнаете о внутреннем устройстве игр, о том, как их разбирать и модифицировать. Например, Ник учит, как обойти античит, чтобы автоматизировать игру. Разве не здорово иметь собственного бота, который собирает опыт, золото, предметы и многое другое - и все это в ваше отсутствие?

Вы когда-нибудь задумывались, как мошенники обманывают? Вы когда-нибудь хотели поставить патч или защитить свою игру? Возьмите кофе, откройте ноутбук и наслаждайтесь.

Благословений вам и вашим близким, доктор Джаред ДеМотт

Эксперт по безопасности и разработчик программного обеспечения

## **Благодарность**

Написание этой книги было удивительным путешествием, и я не смог бы сделать это в одиночку. Издательство No Starch Press оказалось мне огромную поддержку и тесно сотрудничало со мной, чтобы превратить эту книгу из концепции в реальность. В частности, я хотел бы поблагодарить моего редактора-разработчика Дженинфер Гриффит-Дельгадо и редактора-производителя Лорел Чун. Билл Поллок, Тайлер Ортман, Элисон Лоу и остальные члены команды No Starch - замечательные люди, и я рад, что работал с ними.

Спасибо копирайтеру Рейчел Монаган, корректору Поле Л. Флеминг и техническому рецензенту Стивену Лоулеру. Спасибо также моим друзьям Кэвитту "synt4x" Gloverу и Вадиму Котову, которые нашли время, чтобы просмотреть некоторые главы перед отправкой, а также Джареду ДеМотту за написание предисловия к книге.

Я хотел бы поблагодарить всех людей на TPForums, которые приняли меня, когда я был еще наивным ребенком, и помогли мне научиться взламывать игры. В частности, я должен поблагодарить Джозефа "jo3bingham" Бингхема, Яна Обермиллера и jegeremic, которые оказали значительное влияние на мое становление как хакера, а также основателя TPForums Джоша "Zurphrus" Хартцелла, который помог мне обрести уверенность и навыки, когда мое будущее казалось самым мрачным.

Спасибо также всему персоналу форума и каждому клиенту, который когда-либо пользовался моими ботами. И, наконец, спасибо моей семье, друзьям и коллегам, которые с удовольствием и поддержкой помогали мне стать тем человеком, которым я являюсь сегодня.



## Введение

Распространенным заблуждением в мире онлайн-игр является идея о том, что единственная игра, в которую вы можете играть, — это та, которая указана в названии.

На самом деле хакерам нравится играть в игру, которая скрывается за занавесом: это игра в кошки-мышки между ними и разработчиками игр. В то время как хакеры занимаются реинжинирингом двоичных файлов игр, автоматизацией игровых процессов и модификацией игрового окружения, а разработчики игр борются с инструментами, созданными хакерами (обычно называемыми боты), используя методы защиты от реверсинга, алгоритмы обнаружения ботов и эвристический анализ данных.

По мере развития борьбы между хакерами и разработчиками игр технические методы, применяемые обеими сторонами, — многие из которых напоминают методы, используемые разработчиками вредоносных программ и производителями антивирусов, — эволюционировали и становились все более сложными. Эта книга рассказывает о борьбе, которую ведут игровые хакеры, и о передовых методах, которые они разработали, чтобы манипулировать играми, одновременно ускользая от разработчиков игр в темных закоулках собственного программного обеспечения. Хотя книга посвящена обучению разработке инструментов, которые, скорее всего, будут расценены игровыми компаниями как неприятные или даже вредоносные, вы обнаружите, что многие из методов полезны для разработки инструментов, которые являются совершенно доброкачественными и нейтральными. Кроме того, знание того, как реализуются эти приемы, является ключевым для разработчиков игр, стремящихся предотвратить их использование.

### Необходимые условия для чтения

Эта книга не ставит своей целью научить вас разрабатывать программное обеспечение, и поэтому предполагает, что вы, как минимум, имеете солидный багаж знаний в области разработки программного обеспечения. Этот опыт должен включать

знакомство с разработкой под Windows, а также небольшой опыт разработки игр и управления памятью. Хотя этих навыков вам будет достаточно для изучения этой книги, опыт работы с ассемблером x86 и внутренним устройством Windows позволит вам не упустить детали более продвинутых реализаций.

Более того, поскольку все продвинутые хаки, о которых пойдет речь в этой книге, основаны на внедрении кода, умение писать код на родном языке, таком как С или С++, является обязательным условием. Весь код примеров в этой книге написан на С++ и может быть скомпилирован с помощью Microsoft Visual C++ Express Edition. (Вы можете загрузить MSVC++ Express Edition с сайта <http://www.visualstudio.com/en-US/products/visual-studio-express-vs>.)

#### **НОТЕ**

*Другие языки, компилирующие в-native код, такие как Delphi, также способны к инжектированию, но в этой книге я не буду их рассматривать.*

## **Краткая история взлома игр**

С момента появления онлайн-компьютерных игр в начале 1980-х годов идет непрерывная борьба умов между хакерами и разработчиками игр. Эта бесконечная битва подталкивает разработчиков игр посвящать огромное количество времени защите своих продуктов от вмешательства хакеров. В свою очередь, хакеры занимаются реверс-инжинирингом игровых файлов, автоматизацией процессов и изменением игрового окружения. Их мотивация разнообразна: улучшение графики, повышение производительности, удобство использования, автономная игра, получение внутриигровых ресурсов и, конечно, реальный финансовый доход.

Конец 1990-х и начало 2000-х годов стали золотым веком взлома игр, поскольку онлайн-игры к тому времени стали достаточно популярными, чтобы привлекать большую аудиторию, но ещё были простыми для модификации и манипуляций. Игры того периода, такие как Tibia (1997), Runescape (2001) и Ultima Online (1997), оказались в центре внимания разработчиков ботов. Создатели этих игр до сих пор с трудом контролируют многочисленные сообщества пользователей и разработчиков ботов. Пассивность разработчиков игр и упорство хакеров не только полностью разрушили внутриигровую экономику, но и породили прибыльную индустрию, специализирующуюся на разработке и защите ботов.

Конец 1990-х — начало 2000-х годов стал «золотым веком» взлома игр. В этот период онлайн-игры стали достаточно сложными, чтобы привлечь большую аудиторию, но в то же время оставались простыми для перепрограммирования. В результате бездействия игровых компаний и упорства хакеров игровые экономики были разрушены, а разработка и защита ботов превратились в полноценную коммерческую индустрию.

Со временем крупные игровые компании стали серьёзнее относиться к борьбе с ботами. Сегодня они имеют специализированные команды по борьбе с ботами, активно банят игроков за использование ботов и подают судебные иски против

их разработчиков. Поэтому игровые хакеры вынуждены создавать сложные методы скрытности для защиты своих пользователей.

Эта борьба продолжается и сегодня, и число участников с обеих сторон растёт с распространением онлайн-игр. Крупнейшие разработчики преследуют хакеров с неослабевающей решительностью и даже предъявляют многомиллионные иски против самых успешных взломщиков. В ближайшие годы взлом игр и создание ботов будет продолжать развиваться, превращаясь в еще более прибыльную отрасль для тех хакеров, которые готовы рискнуть.

## Зачем взламывать игры?

Помимо очевидного увлечения и интереса к самому процессу, взлом игр также имеет практическое применение. Ежедневно тысячи начинающих программистов экспериментируют с небольшими игровыми взломами, чтобы автоматизировать скучные и повторяющиеся задачи. Такие начинающие пользователи, или «скрипт-кидди», используют простые средства автоматизации вроде AutoIt для своих небольших и относительно безобидных хаков. В то же время профессиональные хакеры, обладающие обширным инструментарием и многолетним опытом программирования, тратят сотни часов на создание продвинутых игровых хаков. Именно эти сложные хаки подробно рассматриваются в данной книге, поскольку часто разрабатываются с целью получения реального дохода.

Игровая индустрия является огромным рынком, оборот которого в 2014 году составил 22,4 миллиарда долларов по данным Entertainment Software Association. Из десятков миллионов ежедневных игроков около 20% играют в массовые многопользовательские онлайн-ролевые игры (MMORPG). В таких играх тысячи игроков торгуют виртуальными предметами внутри активно развивающейся игровой экономики. Игроки часто нуждаются во внутриигровых ресурсах и готовы покупать их за реальные деньги, что приводит к формированию больших сообществ, предлагающих услуги обмена игрового золота на реальные деньги. В некоторых случаях эти сообщества даже устанавливают официальные курсы обмена игровой валюты на реальные деньги.

Хакеры используют это, создавая ботов, которые автоматически добывают золото и повышают уровень персонажей. В зависимости от цели хакера, такие боты используются либо для массовых «ферм» золота, прибыль от которых затем продаётся, либо продаются сами программы игрокам, желающим быстро и без усилий получать ресурсы и уровни. Благодаря массовости сообщества вокруг популярных MMORPG, годовой доход таких хакеров может достигать шестизначных и даже семизначных цифр.

Хотя MMORPG дают наибольшее пространство для атаки, их аудитория всё же относительно невелика. Около 38% геймеров предпочитают стратегии в реальном времени (RTS) и

многопользовательские онлайн-арены (МОВА), ещё 6% предпочитают шутеры от первого лица (FPS). Эти соревновательные игры формата «игрок против игрока» (PvP) занимают 44% игрового рынка и приносят значительную прибыль настойчивым хакерам.

Игры PvP часто носят эпизодический характер, и каждый матч – это отдельная игра. Здесь нет большого смысла использовать ботов для долгосрочной автоматизации, например, когда пользователь отходит от клавиатуры (AFK). Вместо золотых ферм или автономных ботов здесь обычно создают реактивных ботов, помогающих игрокам во время боевых действий.

Игроки участвуют в таких играх, прежде всего, для того, чтобы проявить своё мастерство. В результате число игроков, заинтересованных в покупке ботов для PvP, значительно ниже по сравнению с MMORPG, где требуется длительная рутина. Однако хакеры могут получать неплохой доход и от продажи PvP-ботов, которые, к тому же, обычно гораздо проще в разработке, чем полноценные автономные боты.

## Как организована эта книга

Книга состоит из четырёх частей, каждая из которых посвящена отдельному аспекту взлома игр. В первой части — «Рабочие инструменты» — вы получите набор инструментов, которые помогут вам эффективно взламывать игры.

**Глава 1: Сканирование памяти с помощью Cheat Engine** научит вас находить важные значения в памяти игр, используя популярный инструмент Cheat Engine.

**Глава 2: Отладка игр с помощью OllyDbg** познакомит вас с основами отладки и реверс-инжиниринга игр с помощью OllyDbg.

В завершение части, **Глава 3: Разведка с помощью Process Monitor и Process Explorer** покажет, как использовать два мощных инструмента для анализа того, как игры взаимодействуют с файлами, процессами, сетью и операционной системой.

Для каждой главы первой части предусмотрены онлайн-ресурсы, содержащие специально подготовленные бинарные файлы для безопасного тестирования и отработки новых навыков.

Когда вы освоитесь с базовыми инструментами, переходите к второй части книги — «Анализ игры», где вы научитесь глубже понимать устройство игр.

**Глава 4: Из кода в память: Общий курс** объяснит, как выглядят исходный код и игровые данные после компиляции в исполняемый файл.

**Глава 5: Продвинутый анализ памяти** развивает знания, полученные в предыдущей главе. Вы узнаете, как с помощью сканирования и отладки легко находить сложные структуры и классы в памяти.

**Глава 6: Чтение и запись в игровую память** покажет, как считывать и изменять данные непосредственно в процессе работы игры.

Эти главы содержат подробные примеры, с помощью которых

вы сможете проверить свои знания на практике.

В третьей части — «Управление процессом» — вы станете настоящим кукловодом, узнав, как превратить любую игру в свою марионетку.

На основе навыков из предыдущих частей, **Глава 7: Инъекция кода** расскажет, как внедрять и запускать собственный код в адресном пространстве игры.

**Глава 8: Манипуляция потоком управления в игре** научит использовать инъекцию кода для перехвата, изменения или отключения любых вызовов функций игры. Здесь вы найдете примеры из реальной практики для популярных библиотек, таких как Adobe AIR и Direct3D.

Эти главы дополнены тысячами строк готового к использованию кода, которые могут служить основой при создании ваших ботов.

Четвёртая часть — «Создание ботов» — покажет, как объединить полученные знания и навыки, чтобы создавать мощных игровых ботов.

**Глава 9: Использование экстрасенсорного восприятия для борьбы с туманом войны** научит раскрывать полезную информацию, скрытую игровым интерфейсом, такую как расположение врагов или количество зарабатываемого опыта.

**Глава 10: Адаптивные хаки** продемонстрирует шаблоны кода, которые позволяют ботам мгновенно реагировать на внутриигровые события быстрее человека.

**Глава 11: Собираем всё вместе: Создание автономных ботов** раскроет принципы работы полностью автономных игровых ботов, которые могут играть без участия человека. В этой главе вы узнаете основы теории управления, машин состояний, поисковых алгоритмов и математических моделей.

**Глава 12: Оставаясь незамеченным** познакомит вас с методами, позволяющими вашим ботам избегать обнаружения системами защиты игр.

Каждая глава сопровождается множеством примеров кода. Некоторые хаки основаны на коде из предыдущих глав, а другие демонстрируют простые и эффективные шаблоны проектирования, которые можно использовать для разработки собственных ботов.

## Об онлайн-ресурсах

Помимо книги, вы найдёте множество дополнительных материалов на сайте <https://www.nostarch.com/gamehacking/>. Эти ресурсы включают специально созданные бинарные файлы для безопасного тестирования ваших навыков, большое количество примеров кода, а также готовые к использованию фрагменты кода для взлома игр. Данные ресурсы тесно связаны с материалами книги, и без них изучение будет неполным. Обязательно скачайте эти ресурсы перед продолжением чтения.

## Как использовать эту книгу

Эта книга предназначена в первую очередь как руководство, которое шаг за шагом обучает навыкам взлома игр. Каждая глава последовательно знакомит вас с новыми знаниями и умениями, основываясь на предыдущих материалах. По мере изучения глав я рекомендую вам активно экспериментировать с приведёнными примерами кода и проверять свои навыки на реальных играх. Такой подход важен, так как некоторые темы становятся понятными только после практического применения.

После прохождения материала книги, вы сможете использовать её и как справочник. Например, если вам понадобится помочь в разборе структуры данных, вы сможете вернуться к главе 5. Если вы занимаетесь реверс-инжинирингом формата карт игры и готовы приступить к созданию алгоритма поиска пути, то всегда можете обратиться к главе 11 и использовать её примеры в качестве отправной точки.

Хотя невозможно предугадать все проблемы, с которыми вы столкнётесь, я постарался обеспечить вас достаточным количеством примеров и рекомендаций, чтобы помочь справиться с наиболее частыми трудностями.

### Заметка от издателя

Эта книга не поддерживает пиратство, нарушение закона DMCA, нарушение авторских прав или условий использования игр. Хакеры, занимающиеся взломом игр, могут быть навсегда заблокированы в играх, привлечены к судебной ответственности на многомиллионные суммы или даже попасть в тюрьму за свою деятельность.

# **Часть 1: Инструменты ремесла**

# 1) Сканирование памяти с использованием Cheat Engine



Лучшие игровые хакеры мира годами создают и персонализируют обширные арсеналы инструментов собственной разработки. Такие мощные наборы инструментов позволяют им эффективно анализировать игры, быстро создавать прототипы и разрабатывать ботов. В основе любого такого набора всегда лежат четыре ключевых компонента: сканер памяти, отладчик на уровне ассемблера, монитор процессов и шестнадцатеричный редактор.

Сканирование памяти является важнейшим этапом взлома игр, и в этой главе вы познакомитесь с Cheat Engine — мощным сканером памяти, который позволяет искать в оперативной памяти игры такие значения, как уровень игрока, здоровье или внутриигровые деньги. Сначала мы разберём базовые принципы сканирования и модификации памяти, а также методы работы с указателями. Затем вы научитесь использовать встроенный в Cheat Engine скриптовый движок Lua.

**NOTE** Вы можете загрузить Cheat Engine с официального сайта: <http://www.cheatengine.org/>. Будьте внимательны при установке, так как инсталлятор может предложить установить дополнительные панели инструментов и другие ненужные программы. При желании вы можете отключить эти опции во время установки.

## Почему важны сканеры памяти

Понимание состояния игры является ключевым моментом для создания эффективных хаков. Любая игра содержит данные, описывающие её текущее состояние, в числовом виде в оперативной памяти компьютера. Хакеры используют специальные инструменты для сканирования памяти, чтобы найти нужные значения и адреса, после чего их программы могут

считывать эти данные для анализа текущего состояния игры.

Например, программа, которая автоматически лечит персонажа, когда его здоровье опускается ниже 500 единиц, должна отслеживать состояние его здоровья. Для этого хакер сначала находит соответствующий адрес в памяти игры с помощью сканера памяти, а затем регулярно считывает текущее значение, чтобы понять состояние персонажа.

Пример такого алгоритма в псевдокоде:

```
// выполняйте это в некотором цикле
Health = readMemory(game, HEALTH_LOCATION)
if (Health < 500)
    pressButton(HEAL_BUTTON)
```

Сканер памяти позволяет найти `HEALTH_LOCATION` чтобы ваше программное обеспечение могло запросить его позже.

## Сканирование памяти

Сканер памяти является самым базовым, но при этом наиболее важным инструментом для начинающего игрового хакера. Как и в любой другой программе, все игровые данные хранятся в памяти по абсолютным адресам. Если представить память как большой набор байтов, то каждая порция данных находится по конкретному адресу памяти. Сканер памяти проходит по всем адресам в памяти игры и ищет те значения, которые совпадают с искомым значением `x`. Каждый раз, когда сканер находит такое значение, он добавляет адрес в список результатов.

Однако из-за большого размера памяти игры значение `x` может встречаться в сотнях различных мест. Представьте, что `x` – это текущее здоровье игрока. Оно может быть, например, равным 500. Но значение «500» случайным образом может оказаться в памяти игры во многих местах, не связанных с настоящим здоровьем игрока. Чтобы отфильтровать эти ненужные совпадения, сканер памяти позволяет выполнить повторное сканирование списка результатов, удаляя адреса, которые больше не содержат того же значения, что и `x` (неважно, равно ли `x` по-прежнему 500 или изменилось).

Чтобы повторное сканирование было эффективным, общее состояние игры должно изменяться между сканированиями. Представьте, что после первого сканирования вы изменяете состояние игры, например, теряя здоровье или получая урон. Последующие сканирования позволяют отсеять ложные совпадения и выявить истинный адрес искомого значения `x`.

## Сканер памяти Cheat Engine

Этот раздел познакомит вас с возможностями сканера памяти Cheat Engine, который поможет находить адреса значений состояния игры в оперативной памяти. Вы сможете опробовать сканер на практике в разделе «Базовое редактирование памяти» на странице 11, а пока откройте Cheat Engine и ознакомьтесь с интерфейсом. Все основные функции сканирования памяти

компактно расположены в главном окне программы, как показано на рисунке 1-1.

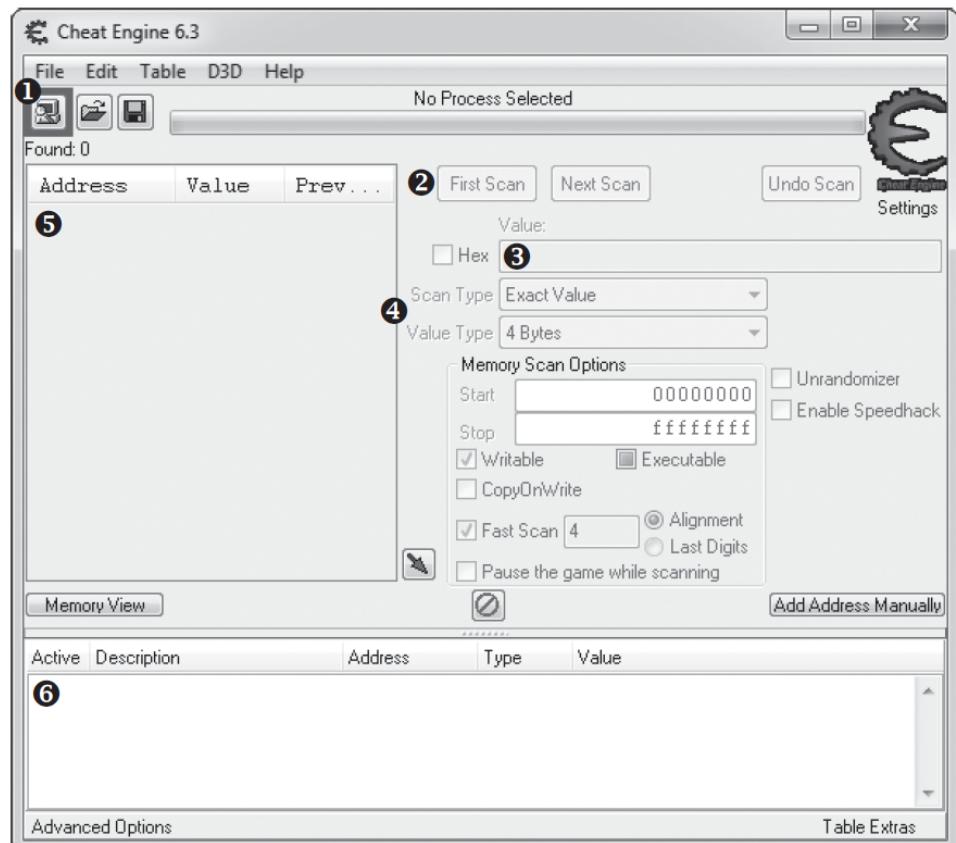


Рисунок 1-1: Главный экран Cheat Engine

Чтобы начать сканирование памяти игры, нажмите на значок присоединения к процессу (Attach icon) и выберите нужный процесс. Затем введите искомое значение (обозначенное ранее как  $x$  в нашем концептуальном примере)  $w$ . Присоединяя процесс, мы даём Cheat Engine команду подготовиться к работе именно с ним, а в данном случае – к сканированию его памяти. Также полезно указать Cheat Engine тип сканирования, который будет рассмотрен далее.

## Типы сканирования

Cheat Engine позволяет выбрать два различных параметра сканирования: **Scan Type** (тип сканирования) и **Value Type** (тип значения) (8).

Параметр **Scan Type** указывает сканеру, каким образом сравнивать искомое значение со значениями в памяти при сканировании, и включает следующие типы:

**Exact Value (Точное значение)** – возвращает адреса,

указывающие на значения, равные искомому. Выбирайте этот вариант, если искомое значение не меняется в процессе сканирования; например, здоровье, мана и уровень обычно соответствуют этой категории.

**Bigger Than (Больше чем)** – возвращает адреса, указывающие на значения, которые больше заданного искомого значения. Эта опция полезна, когда вы ищете значения, которые постоянно увеличиваются, например таймеры.

**Smaller Than (Меньше чем)** – возвращает адреса, указывающие на значения, которые меньше заданного искомого значения. Как и предыдущая опция, она полезна для поиска таймеров (в данном случае, тех, которые ведут обратный отсчёт).

**Value Between (Значение в диапазоне)** – возвращает адреса, значения которых попадают в указанный диапазон. Эта опция сочетает в себе возможности «Больше чем» и «Меньше чем», предоставляя дополнительное поле ввода, позволяющее указать гораздо более узкий диапазон значений.

**Unknown Initial Value (Неизвестное начальное значение)** – возвращает все адреса в памяти программы, позволяя при повторных сканированиях отслеживать изменения значений относительно их исходного состояния. Эта опция полезна для поиска типов предметов или существ, поскольку внутренние значения, используемые разработчиками игр для этих объектов, обычно неизвестны заранее.

Параметр **Value Type** определяет тип переменной, которую сканер Cheat Engine должен искать.

## Запуск первого сканирования

После того, как оба параметра сканирования установлены, нажмите кнопку First Scan («Первое сканирование») ②, чтобы запустить начальное сканирование значений, после чего сканер заполнит список результатов ⑤. Все адреса, отображаемые в списке зелёным цветом, являются статическими (static), что означает, что они сохраняются неизменными при перезапуске программы. Адреса, указанные чёрным цветом, расположены в динамически выделяемой памяти (dynamically allocated memory), то есть в памяти, выделяемой во время выполнения программы.

Когда список результатов заполняется впервые, он показывает адрес и текущее значение каждого результата. При повторных сканированиях будет отображаться также значение каждого результата на момент предыдущего сканирования. (Любые значения в реальном времени обновляются с интервалом, который вы можете настроить в меню Edit → Settings → General Settings → Update interval.)

## Следующие сканы

Как только список результатов заполнен, сканер активирует кнопку Next Scan («Следующее сканирование») ②, которая предоставляет шесть новых типов сканирования. Эти дополнительные типы позволяют сравнивать адреса из списка

результатов с их предыдущими значениями, полученными во время последнего сканирования, что поможет точнее определить, какой именно адрес содержит нужное игровое значение. Эти типы следующие:

Increased Value (Значение увеличилось) – возвращает адреса, указывающие на значения, которые увеличились. Этот тип дополняет тип «Больше чем» тем, что сохраняет то же минимальное значение и удаляет любые адреса, значения которых уменьшились.

Increased Value By (Значение увеличилось на) – возвращает адреса, значения которых увеличились на указанное число. Данный тип обычно возвращает значительно меньше ложных совпадений, но использовать его стоит, только если известно, на сколько именно увеличилось значение.

Decreased Value (Значение уменьшилось) – противоположность типу «Значение увеличилось».

Decreased Value By (Значение уменьшилось на) – противоположность типу «Значение увеличилось на».

Changed Value (Значение изменилось) – возвращает адреса, указывающие на значения, которые изменились. Этот тип полезен, когда известно, что значение должно измениться, но неизвестно, в какую сторону.

Unchanged Value (Значение не изменилось) – возвращает адреса, указывающие на значения, которые не изменились. Данный тип помогает исключить ложные совпадения, поскольку можно легко увеличить энтропию игрового окружения и в то же время гарантировать, что нужное значение остаётся неизменным.

Обычно необходимо использовать несколько разных типов сканирования, чтобы сузить большой список результатов до одного правильного адреса. Устранение ложных совпадений часто зависит от правильного создания энтропии (как описано в разделе «Базовое сканирование памяти» на стр. 4), тактического изменения параметров сканирования, последовательного нажатия кнопки «Next Scan» и повторения этого процесса до тех пор, пока не останется один точный адрес.

## Когда невозможно получить единственный результат

Иногда невозможно точно определить единственный результат с помощью Cheat Engine, и в таких случаях правильный адрес приходится устанавливать экспериментальным путём. Например, если вы ищете текущее значение здоровья вашего персонажа и не можете сузить список результатов до менее пяти адресов, вы можете попробовать изменять значение каждого адреса вручную (как описано в разделе «Ручная модификация с помощью Cheat Engine» на странице 8) до тех пор, пока не увидите изменение здоровья на экране или пока другие значения не начнут автоматически соответствовать установленному вами.

## Таблицы читов

Как только вы найдёте правильный адрес, вы можете дважды щёлкнуть по нему, чтобы добавить его в панель таблицы читов (cheat table pane) ❶. Адреса в панели таблицы читов можно изменять, отслеживать и сохранять в файлы таблиц читов для последующего использования.

Для каждого адреса в панели таблицы читов вы можете добавить описание, дважды щёлкнув по столбцу «Description» («Описание»), и назначить цвет, щёлкнув правой кнопкой мыши и выбрав пункт «Change Color» («Изменить цвет»). Вы также можете отображать значения адресов в шестнадцатеричном или десятичном формате, щёлкнув правой кнопкой мыши и выбрав соответственно «Show as hexadecimal» («Отображать в шестнадцатеричном виде») или «Show as decimal» («Отображать в десятичном виде»). Наконец, вы можете изменить тип данных каждого значения, дважды щёлкнув по столбцу «Type» («Тип»), или непосредственно изменить само значение, дважды щёлкнув по столбцу «Value» («Значение»).

Основная цель панели таблицы читов — позволить игровому хакеру удобно отслеживать адреса, и её можно динамически сохранять и загружать. Чтобы сохранить текущую таблицу читов в документ .ct, содержащий адреса и их типы, описания, цвета и форматы отображения, перейдите в меню File → Save («Файл → Сохранить») или File → Save As («Файл → Сохранить как»). Для загрузки сохранённых документов .ct используйте меню File → Load («Файл → Загрузить»). (Множество готовых таблиц читов для популярных игр вы можете найти на сайте <http://cheatengine.org/tables.php>.)

Теперь, когда вы знаете, как находить текущее значение состояния игры, в следующем разделе я расскажу, как изменять это значение непосредственно в памяти.

## Модификация памяти в играх

Боты используют изменение значений в памяти игрового процесса для обмана игровой системы, что позволяет, например, получать большое количество внутриигровых денег, изменять здоровье персонажа, его местоположение и многое другое. В большинстве онлайн-игр основные параметры персонажа (такие как здоровье, мана, навыки и позиция) хранятся в памяти, но управляются игровым сервером и передаются в локальный клиент игры через Интернет. Поэтому модификация таких значений во время онлайн-игры носит лишь косметический характер и не влияет на реальные данные. (Любое значимое изменение памяти в онлайн-игре требует куда более сложного взлома, который выходит за рамки возможностей Cheat Engine.). Однако в локальных играх, где нет удалённого сервера, вы можете свободно изменять все эти значения.

## Ручная модификация с помощью Cheat Engine

Мы будем использовать Cheat Engine, чтобы понять, как именно работает изменение памяти. Чтобы вручную изменить данные в памяти, выполните следующие шаги:

Подключите Cheat Engine к игре. Либо выполните сканирование, чтобы найти нужный адрес, либо загрузите таблицу читов, содержащую его.

Дважды щёлкните по столбцу Value (значение) для нужного адреса, чтобы открыть окно ввода, в котором можно задать новое значение.

Если вы хотите, чтобы новое значение не могло быть перезаписано, установите галочку в столбце Active (активный) рядом с нужным адресом, чтобы заморозить его. Это приведёт к тому, что Cheat Engine будет постоянно перезаписывать заданное значение при любых изменениях.

Этот метод отлично подходит для быстрых и простых хакинговых приёмов, но постоянное ручное изменение значений утомительно. Гораздо удобнее было бы автоматизировать этот процесс.

## Генератор трейнеров

Генератор трейнеров в Cheat Engine позволяет автоматизировать весь процесс модификации памяти без написания кода. Чтобы создать трейнер (простой бот, который привязывает действия по изменению памяти к комбинациям клавиш), перейдите в File → Create generic trainer Lua script from table («Файл → Создать универсальный Lua-скрипт трейнера из таблицы»). Это откроет окно генератора трейнеров, похожее на показанное на рисунке 1-2.

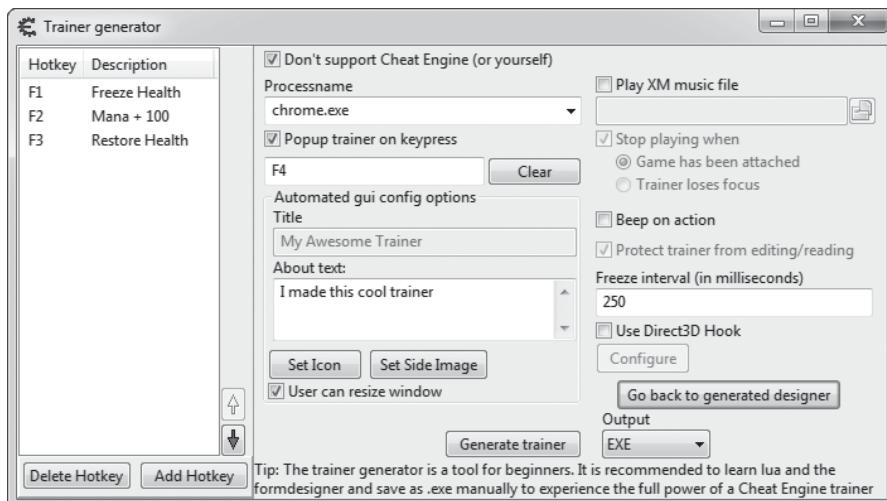


Рисунок 1-2: Диалоговое окно генератора Cheat Engine Trainer

Описание полей для редактирования:

Processname (Имя процесса) – название исполняемого файла, к которому должен подключаться трейнер. Это имя отображается в списке процессов, когда Cheat Engine присоединяется к процессу, и оно должно автоматически заполняться названием процесса, к которому подключен Cheat Engine.

Pop up trainer on keypress (Отображение трейнера по нажатию клавиши) – опционально включает горячую клавишу (устанавливается путём ввода комбинации в поле рядом с чекбоксом), которая открывает главное окно трейнера.

Title (Заголовок) – имя трейнера, которое будет отображаться в его интерфейсе. Это поле необязательно.

About text (Описание трейнера) – текстовое описание трейнера, которое будет отображаться в его интерфейсе. Это также необязательное поле.

Freeze interval (Интервал заморозки в миллисекундах) – интервал времени, через который операция заморозки (фиксации значения) перезаписывает его. Оставьте значение 250 мс по умолчанию, так как более низкие значения могут излишне нагружать систему, а более высокие – работать слишком медленно.

После настройки этих значений нажмите Add Hotkey («Добавить горячую клавишу»), чтобы задать комбинацию клавиш для активации вашего трейнера. Вам будет предложено выбрать значение из таблицы читов. Введите значение, и вы попадёте в окно установки/изменения горячих клавиш, похожее на рисунок 1-3.

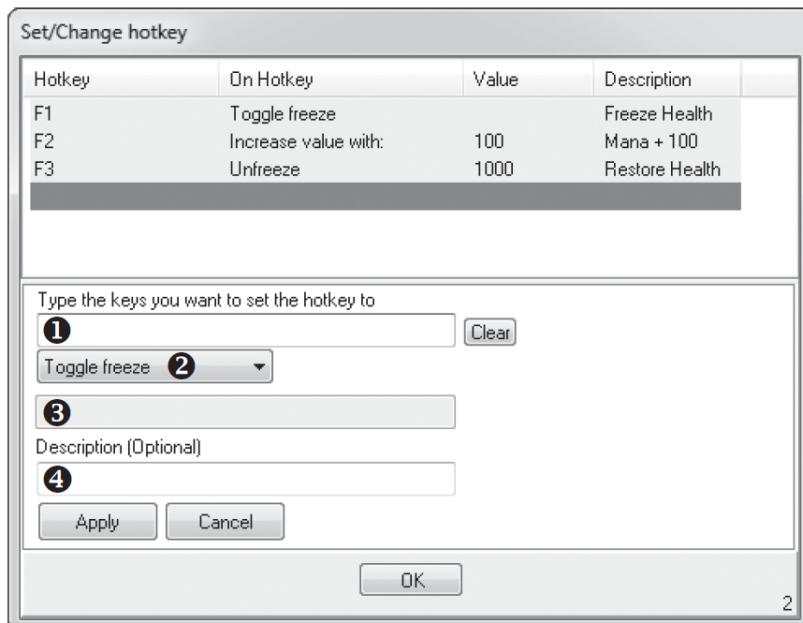


Рисунок 1-3: Экран установки/изменения горячих клавиш Cheat Engine

На этом экране установите курсор в поле с надписью Type the keys you want to set the hotkey to (Ведите клавиши, которые хотите назначить на горячую клавишу) ① и введите нужную комбинацию клавиш. Затем выберите желаемое действие из выпадающего списка ②. Ваши варианты должны быть представлены в следующем порядке:

Toggle freeze (Переключить заморозку) – переключает состояние заморозки адреса.

Toggle freeze and allow increase (Переключить заморозку и разрешить увеличение) – замораживает адрес, но позволяет увеличивать значение. Если значение уменьшается, трейнер перезаписывает его предыдущим значением. Увеличенные значения не будут изменены.

Toggle freeze and allow decrease (Переключить заморозку и разрешить уменьшение) – выполняет действие, противоположное Toggle freeze and allow increase.

Freeze (Заморозить) – устанавливает адрес в замороженное состояние, если он ещё не заморожен.

Unfreeze (Разморозить) – снимает заморозку с адреса, если он был заморожен.

Set value to (Установить значение в) – устанавливает значение, указанное в поле Value box ③.

Decrease value with (Уменьшить значение на) – уменьшает значение на количество, указанное в поле Value box ③.

Increase value with (Увеличить значение на) – выполняет действие, противоположное Decrease value with.

Наконец, вы можете задать описание для действия ④. Нажмите Apply («Применить»), затем OK, и ваше действие появится в списке на экране генератора трейнера. В этот момент Cheat Engine запустит трейнер в фоновом режиме, и вы сможете просто нажимать назначенные горячие клавиши для выполнения действий в памяти.

Чтобы сохранить ваш трейнер в портативный исполняемый файл, нажмите Generate trainer («Создать трейнер»). Запуск этого исполняемого файла после старта игры автоматически подключит трейнер к игре, так что вы сможете использовать его без необходимости открывать Cheat Engine.

Теперь, когда вы освоились с Cheat Engine, его сканером памяти и генератором трейнеров, попробуйте самостоятельно изменить некоторые данные в памяти.

#### БАЗОВОЕ РЕДАКТИРОВАНИЕ ПАМЯТИ

Скачайте файлы для этой книги с <https://www.nostarch.com/gamehacking/> и запустите BasicMemory.exe. Затем откройте Cheat Engine и подключитесь к этому исполняемому файлу. После этого, используя только Cheat Engine, найдите адреса, отвечающие за х и у-координаты серого шара. (Подсказка: используйте тип данных 4 Bytes).

После того как вы найдёте нужные значения, измените их, чтобы разместить шар поверх чёрного квадрата. Если вы всё сделали правильно, игра уведомит вас об успехе, отобразив текст «Good job!» (Отличная работа!).

(Подсказка: каждый раз при перемещении шара его позиция — хранящаяся в виде 4-байтового целого числа — изменяется на 1. Также ищите только статические (зелёные) результаты).

## Сканирование указателя

Как уже упоминалось, онлайн-игры часто хранят значения в динамически выделенной памяти. Адреса, ссылающиеся на динамическую память, сами по себе бесполезны, однако некоторые статические адреса могут всегда указывать на другой адрес, который, в свою очередь, указывает на следующий адрес, и так далее, пока последний адрес в цепочке не приведёт к динамическому значению, которое нас интересует.

В Cheat Engine можно находить такие цепочки адресов с помощью метода, называемого сканированием указателей (pointer scanning).

В этом разделе я познакомлю вас с цепочками указателей и расскажу, как работает их сканирование в Cheat Engine. После того как вы разберётесь с интерфейсом, вы сможете попрактиковаться в разделе «Pointer Scanning» на странице 18.

## Цепочки указателей

Цепочка смещений, которую я только что описал, называется цепочкой указателей (pointer chain) и выглядит следующим образом:

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

Первое значение в этой цепочке указателей (start) называется указателем памяти (memory pointer). Это адрес, с которого начинается цепочка. Остальные значения (offset1, offset2 и так далее) формируют путь к нужному значению и называются путём указателя (pointer path).

Этот псевдокод показывает, как можно обработать цепочку указателей:

---

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

---

Этот код создаёт функцию `readPointerPath()`, которая принимает цепочку указателей `chain` в качестве параметра.

Функция `readPointerPath()` рассматривает путь указателя в `chain` как список смещений памяти, отсчитываемых от адресного регистра, который изначально устанавливается в указатель на память ❶. Затем эта функция выполняет цикл по этим смещениям, обновляя значение `ret` при каждой итерации на результат выполнения `read(ret + offset)`. После завершения цикла функция возвращает конечное значение `ret`.

Этот псевдокод демонстрирует, как выполняется вызов `readPointerPath()`, если развернуть цикл:

```
list<int> chain = {0xDEADBEEF, 0xAB, 0x10, 0xCC}
value = readPointerPath(chain)

// Развернутый вызов функции:
ret = read(0xDEADBEEF) // начальный указатель [chain[0]]
ret = read(ret + 0xAB)
ret = read(ret + 0x10)
ret = read(ret + 0xCC)
int value = ret
```

В результате выполнения функция вызывает `read()` четыре раза на четырёх различных адресах, по одному разу для каждого элемента в цепочке указателей.

**НОТЕ** *Многие игровые хакеры предпочитают писать код чтения цепочек указателей прямо в коде, вместо того чтобы инкапсулировать его в отдельные функции, такие как `readPointerPath()`.*

## Основы сканирования с помощью указателя

Цепочки указателей существуют, потому что каждый участок динамически выделенной памяти должен иметь соответствующий статический адрес, который код игры может использовать в качестве ссылки.

Игровые хакеры могут получить доступ к этим участкам памяти, определяя цепочки указателей, которые на них ссылаются. Однако из-за их многоуровневой структуры цепочки указателей нельзя найти с помощью линейного поиска, который используют обычные сканеры памяти, поэтому хакеры разработали новые методы для их обнаружения.

С точки зрения реверс-инжиниринга, можно проанализировать ассемблерный код, чтобы выяснить, какой путь указателя используется для доступа к нужному значению. Однако этот процесс очень трудоёмкий, требует много времени и продвинутых инструментов.

Сканеры указателей (Pointer scanners) решают эту проблему, используя метод перебора (brute-force), который рекурсивно перебирает все возможные цепочки указателей, пока не найдётся та, которая приведёт к целевому адресу памяти.

Псевдокод из Листинга 1-1 должен дать вам общее представление о том, как работает сканер указателей.

---

```
list<int> pointerScan(target, maxAdd, maxDepth) {
    ❶    for address = BASE, 0x7FFFFFFF, 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 {}
}
```

---

Листинг 1-1: Псевдокод для сканера указателей

Этот код включает функции `pointerScan()` и `rScan()`.

## pointerScan()

Функция `pointerScan()` является точкой входа в процесс сканирования. Она принимает следующие параметры:

`target` – динамический адрес памяти, который нужно найти,

`maxAdd` – максимальное значение любого смещения,

`maxDepth` – максимальная длина пути указателя.

Затем функция выполняет цикл по каждому 4-байтово выровненному адресу ❶ в памяти игры и вызывает `rScan()`,

передавая ей параметры:

address – текущий адрес в итерации,  
target, maxAdd, maxDepth и curDepth (глубина пути, которая в  
этом случае всегда равна 1).

## rScan()

Функция rScan() считывает данные из памяти с каждого 4-  
байтого выровненного смещения в диапазоне от 0 до maxAdd ❷ и  
возвращает результат, если он совпадает с target ❸.

Если rScan() не находит совпадение в первой итерации и  
глубина рекурсии не слишком велика ❹, она увеличивает  
curDepth и снова выполняет цикл по каждому смещению ❺,  
вызывая саму себя для каждой итерации.

Если рекурсивный вызов возвращает частичный путь указателя  
❻, функция rScan() добавляет текущее смещение в этот путь и  
возвращает цепочку рекурсии ❼ до тех пор, пока не достигнет  
pointerScan().

Когда вызов rScan() из pointerScan() возвращает полную  
цепочку указателей, функция pointerScan() добавляет текущий  
адрес в начало цепочки и возвращает её как полный путь.

## Сканирование указателей с помощью Cheat Engine

В предыдущем примере был показан базовый процесс  
сканирования указателей, но представленная реализация является  
примитивной. Помимо того, что её выполнение чрезвычайно  
медленное, она также порождает бесчисленное количество ложных  
совпадений.

Сканер указателей в Cheat Engine использует ряд продвинутых  
интерполяционных методов, которые ускоряют сканирование и  
делают его более точным. В этом разделе я познакомлю вас с  
широким набором доступных параметров сканирования.

Чтобы запустить сканирование указателей в Cheat Engine,  
щелкните правой кнопкой мыши по адресу в динамической  
памяти в вашей таблице читов и выберите Pointer scan for this  
address (Сканировать указатели для этого адреса).

Когда вы инициируете сканирование указателей, Cheat Engine  
запросит, где сохранить результаты сканирования в виде файла  
.ptr. После того как вы укажете путь, откроется диалоговое окно  
Pointerscanner scanoptions, похожее на то, что представлено на  
Рисунке 1-4.

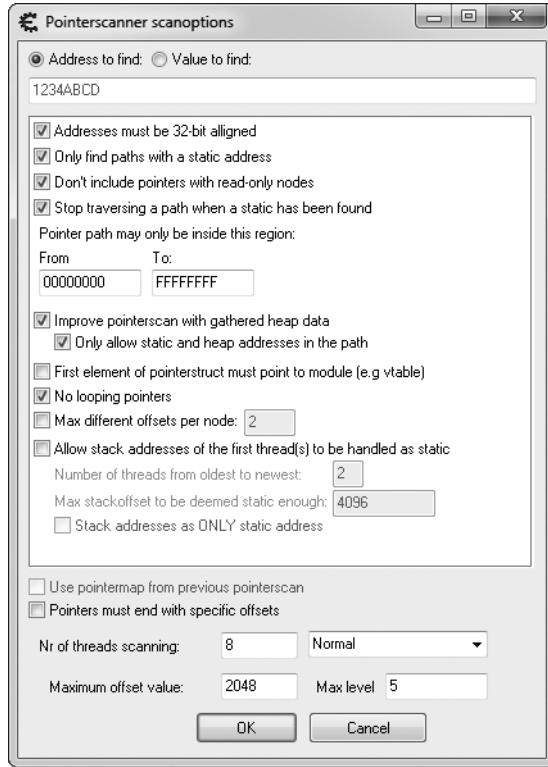


Рисунок 1-4: Cheat Engine Pointerscanner диалог параметров сканирования

В поле ввода Адрес для поиска в верхней части отображается адрес вашей динамической памяти. Теперь тщательно выберите одну из многочисленных опций сканирования Cheat Engine.

## Ключевые параметры

Некоторые параметры сканирования Cheat Engine обычно сохраняют свои значения по умолчанию. Вот эти параметры:

**Addresses must be 32-bits aligned** – Указывает Cheat Engine сканировать только те адреса, которые кратны 4, что значительно увеличивает скорость сканирования. Как вы узнаете в главе 4, компиляторы автоматически выравнивают данные, так что большинство адресов уже будут кратными 4. Вам редко понадобится отключать этот параметр.

**Only find paths with a static address** – Ускоряет процесс сканирования, предотвращая поиск путей, начинающихся с динамического указателя. Этот параметр всегда должен быть включён, так как поиск путей от другого динамического адреса может быть бесполезным.

**Don't include pointers with read-only nodes** – Должен быть всегда включён. Динамически выделенная память, содержащая изменяемые данные, никогда не должна быть доступна только для чтения.

Stop traversing a path when a static has been found – Останавливает сканирование, если найден указатель с начальным статическим адресом. Этот параметр помогает снизить количество ложных срабатываний и ускорить процесс поиска.

Pointer path may only be inside this region – Обычно этот параметр можно оставить без изменений. Остальные настройки позволяют компенсировать широкий диапазон сканирования, разумно сужая область поиска.

First element of pointerstruct must point to module – Запрещает Cheat Engine сканировать фрагменты кучи, где нет таблиц виртуальных функций, предполагая, что игра использует объектно-ориентированную архитектуру. Эта настройка может значительно ускорить сканирование, но она крайне ненадёжна и почти всегда должна оставаться отключённой.

No looping pointers – Отклоняет пути, где указатели ссылаются сами на себя, устранивая неэффективные результаты, но слегка замедляя процесс поиска. Обычно этот параметр должен быть включён.

Max level – Определяет максимальную длину пути указателя. (Помните переменную maxDepth в примере кода из листинга 1-1?)  
Оптимальное значение — 6 или 7.

Конечно, иногда вам может понадобиться изменить эти настройки. Например, если поиск не даёт надёжных результатов, отключение No looping pointers или изменение Max level может помочь, особенно если значение, которое вы ищете, находится в динамической структуре данных (например, связанном списке, бинарном дереве или векторе).

Другой пример — параметр Stop traversing a path when a static has been found, который в редких случаях может мешать поиску надёжных результатов.

## Опции, зависящие от ситуации

В отличие от предыдущих параметров, настройки оставшихся опций будут зависеть от вашей ситуации. Вот как определить наилучшую конфигурацию для каждой из них:

Improve pointerscan with gathered heap data – Позволяет Cheat Engine использовать записи о распределении памяти (heap allocation record) для определения пределов смещений, эффективно ускоряя процесс сканирования за счёт устранения множества ложных совпадений. Если игра использует кастомный менеджер памяти (что становится всё более распространённым явлением), эта опция может привести к результату, прямо противоположному тому, для чего она предназначена. Вы можете оставить этот параметр включённым при начальном сканировании, но это первое, что следует отключить, если вам не удаётся найти надёжные пути.

Only allow static and heap addresses in the path – Исключает все пути, которые не могут быть оптимизированы с использованием данных кучи, делая этот метод ещё более агрессивным.

**Max different offsets per node** – Ограничивает количество одинаковых значений, которые проверяет сканер. Если  $n$  различных адресов указывают на 0x0BADF00D, эта опция указывает Cheat Engine учитывать только первые  $m$  адресов. Это может быть крайне полезно, если вам не удаётся сузить список результатов. В других случаях отключение этой опции может помочь избежать пропуска множества допустимых путей.

**Allow stack addresses of the first thread(s) to be handled as static** – Сканирует стеки вызовов первых  $m$  потоков в игре, учитывая первые  $n$  байт в каждом из них. Это позволяет Cheat Engine сканировать параметры и локальные переменные функций в цепочке вызовов игры (обычно целью является поиск переменных, используемых главным игровым циклом). Пути, найденные с этой настройкой, могут быть крайне нестабильными, но в то же время чрезвычайно полезными. Я использую эту опцию только тогда, когда не удаётся найти heap-адреса.

**Stack addresses as only static address** – Расширяет действие предыдущей настройки, позволяя использовать в путях указателей только адреса из стека.

**Pointers must end with specific offsets** – Может быть полезно, если вам известны смещения в конце допустимого пути. Эта опция позволяет указать конкретные смещения, начиная с последнего, что значительно сужает область поиска.

**Nr of threads scanning** – Определяет, сколько потоков будет использовать сканер. В большинстве случаев наилучший вариант – это количество ядер вашего процессора. В раскрывающемся меню можно выбрать приоритет сканирования:

**Idle** – если хотите, чтобы сканирование проходило медленно и не нагружало систему.

**Normal** – стандартный вариант для большинства сканирований.

**Time critical** – полезен для долгих сканирований, но может сделать ваш компьютер непригодным для работы на время выполнения процесса.

**Maximum offset value** – Определяет максимальное значение смещения в пути. (Помните переменную `maxAdd` в Листинге 1-1?) Обычно рекомендуется начинать с низкого значения и увеличивать его только в случае неудачного поиска. Хорошее стартовое значение – 128. Помните, что этот параметр в основном игнорируется, если используются опции heap-оптимизации.

**Н О Т Е** Что, если одновременно включить параметры *Only allow static and heap addresses in the path* и *Stack addresses as only static address*? Окажется ли сканирование пустым? Кажется, это забавный, хотя и бесполезный эксперимент.

Как только вы определили параметры сканирования, нажмите OK, чтобы начать сканирование указателей. Когда сканирование завершится, появится окно результатов со списком найденных цепочек указателей. Этот список часто содержит тысячи результатов, включая как реальные цепочки, так и ложные

совпадения.

## Повторное сканирование указателя

Сканер указателей имеет функцию повторного сканирования, которая может помочь вам устраниить ложные совпадения. Чтобы начать, нажмите CTRL+R в окне результатов, чтобы открыть диалоговое окно Rescan pointerlist, как показано на рисунке 1-5.

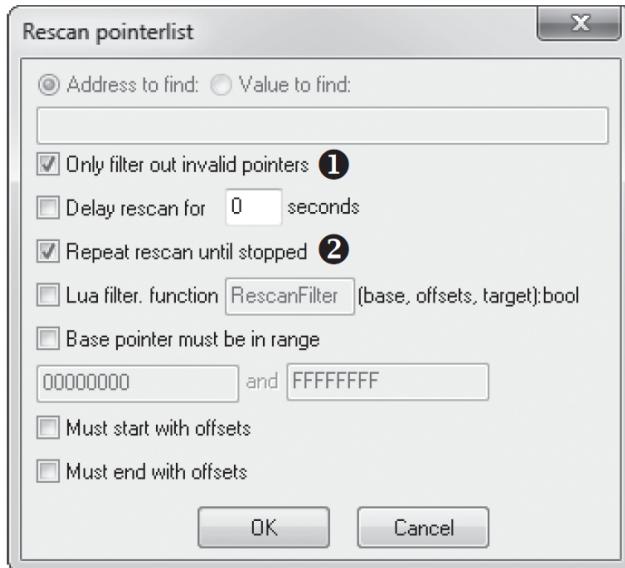


Рисунок 1-5: Cheat Engine Диалог сканирования списка указателей

Существует два основных параметра, которые следует учитывать при повторном сканировании в Cheat Engine:

**Only filter out invalid pointers** Если установить этот флагок ①, повторное сканирование отбросит только цепочки указателей, которые ведут к недопустимым областям памяти, что полезно, если исходный список результатов слишком велик. Отключите этот параметр, чтобы также отфильтровать пути, которые не ведут к конкретному адресу или значению (как показано на рисунке).

**Repeat rescan until stopped** Если установить этот флагок ②, повторное сканирование будет выполняться в непрерывном цикле. В идеале следует включить этот параметр и позволить сканеру работать в фоне, пока вы создаёте максимальную энтропию в памяти.

Для начального повторного сканирования включите оба параметра Only filter out invalid pointers и Repeat rescan until stopped, затем нажмите OK, чтобы запустить процесс.

Окно повторного сканирования закроется, а в окне результатов появится кнопка Stop rescan loop.

Список результатов будет автоматически обновляться, пока вы не нажмёте Stop rescan loop.

Перед этим рекомендуется потратить несколько минут на

создание энтропии в памяти, изменяя игровую среду.

В редких случаях повторное сканирование с помощью цикла повторного сканирования может оставить вас с длинным списком возможных путей. Когда это происходит, вам может понадобиться перезапустить игру, найти адрес, который содержит ваше значение (он мог измениться!), и использовать функцию повторного сканирования для этого адреса, чтобы дополнительно сузить результаты. В этом сканировании оставьте Only filter out invalid pointers неотмеченным и введите новый адрес в поле Address to find.

**НОТЕ**

*Если вам пришлось закрыть окно результатов, вы можете снова открыть его и загрузить список результатов, перейдя в главное окно Cheat Engine и нажав Memory View под панелью результатов. Это должно открыть окно дампа памяти. Когда появится это окно, нажмите CTRL+P, чтобы открыть список результатов сканирования указателей. Затем нажмите CTRL+O, чтобы открыть .ptr файл, в котором был сохранен скан указателей.*

Если ваши результаты все еще недостаточно узкие, попробуйте запустить то же самое сканирование после перезапуска системы или даже на разных системах. Если это по-прежнему дает большой набор результатов, каждый результат можно безопасно считать статическим, так как более чем одна цепочка указателей может разрешаться в один и тот же адрес.

Когда вы сужите свой набор результатов, дважды щелкните на подходящей цепочке указателей, чтобы добавить её в свою таблицу читов. Если у вас есть несколько, казалось бы, одинаковых цепочек, выберите ту, у которой меньше всего смещений. Если вы находите несколько цепочек с идентичными смещениями, начинающихся с одного и того же указателя, но расходящихся после определенного момента, значит, ваши данные могут храниться в динамической структуре данных.

Вот и всё, что касается сканирования указателей в Cheat

**Сканирование указателей**

Перейдите по ссылке <https://www.nostarch.com/gamehacking/> и скачайте MemoryPointers.exe. В отличие от последнего задания, где вам нужно было выиграть всего один раз, это задание требует, чтобы вы выиграли 50 раз за 10 секунд. После каждой победы адрес памяти для координат x и y будет меняться, что позволит вам заморозить значение только в том случае, если вы нашли правильный путь указателя. Начните это упражнение таким же образом, как и предыдущее, но как только вы найдете адреса, используйте функцию поиска указателей, чтобы найти пути указателей к ним. Затем поместите мяч в верхнюю часть черного квадрата, заморозьте значение на месте и нажмите TAB, чтобы начать тест. Как и раньше, игра сообщит вам, когда вы победите. (Подсказка: попробуйте установить максимальный уровень на 5 и максимальное смещение на 512. Также поэкспериментируйте с параметрами, позволяющими учитывать адреса стека, завершите сканирование, когда статический адрес найден, и улучшите поиск указателей с помощью данных heap. Посмотрите, какая комбинация параметров дает наилучшие результаты.)

Engine. Попробуйте сами!

## Среда сценариев Lua

Исторически сложилось так, что разработчики ботов редко использовали Cheat Engine для обновления своих адресов, когда игра выпускала патч, потому что это было гораздо проще сделать в OllyDbg. Это делало Cheat Engine бесполезным для игровых хакеров, кроме как для первоначального исследования и разработки — то есть до тех пор, пока вокруг мощной среды сканирования Cheat Engine не была реализована встроенная среда сценариев на основе Lua.

Хотя этот движок был создан для упрощения разработки простых ботов в Cheat Engine, профессиональные игровые хакеры обнаружили, что его также можно использовать для написания сложных сценариев, которые автоматически находят адреса в разных версиях бинарного файла игры — задача, которая в противном случае могла бы занять часы.

**NOTE** Вы найдете больше информации о движке сценариев Lua в Cheat Engine в вики по адресу <http://wiki.cheatengine.org/>.

Чтобы начать использовать движок Lua, нажмите CTRL+ALT+L в главном окне Cheat Engine. Когда откроется окно, напишите свой сценарий в текстовом поле и нажмите Execute script, чтобы выполнить его. Сохраните сценарий с помощью CTRL+S и откройте сохраненный сценарий с помощью CTRL+O.

Движок сценариев имеет сотни функций и бесконечные варианты использования, поэтому я дам вам лишь общее представление о его возможностях, разбирая два сценария. Каждая игра отличается, и каждый игровой хакер пишет сценарии для достижения уникальных целей, поэтому эти сценарии полезны только для демонстрации концепций.

## Поиск шаблонов в ассемблере

Этот первый сценарий находит функции, которые формируют исходящие пакеты и отправляют их на игровой сервер. Он работает путем поиска в ассемблерном коде игры функций, содержащих определенную последовательность кода.

---

```

❶ BASEADDRESS = getAddress("Game.exe")
❷ function LocatePacketCreation(packetType)
❸   for address = BASEADDRESS, (BASEADDRESS + 0xffffffff) 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
   if (match) then return address + 1 end
end
return 0
end

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

```

---

Код начинается с получения базового адреса модуля, к которому Cheat Engine прикреплен ❶. После того как базовый адрес получен, определяется функция LocatePacketCreation() ❷. Эта функция выполняет цикл по первым 0x2FFFFF байтам памяти в игре ❸, ищет последовательность, которая представляет этот x86 ассемблерный код:

---

```

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

```

---

Функция проверяет, что тип равен packetType, но ей не важно, какое значение имеет смещение функции ❹. Как только эта последовательность найдена, функция возвращается.

Далее определяется функция LocateFunctionHead() ❺. Функция выполняет обратный проход на расстояние до 0x1FFF байт от данного адреса ❻, и на каждом адресе проверяет наличие фрагмента ассемблерного кода ❼, который выглядит следующим

образом:

---

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

---

Этот фрагмент будет присутствовать в начале каждой функции, поскольку он является частью пролога функции, который устанавливает стековый фрейм функции. Когда код найден, функция возвращает адрес фрагмента плюс 1 ❸ (первый байт, 0xCC, является заполнителем).

Чтобы связать эти шаги вместе, вызывается функция LocatePacketCreation() с packetType, который мы ищем (произвольно 0x64), и полученный адрес передается в функцию LocateFunctionHead() ❹. Это эффективно находит первую функцию, которая помещает packetType в вызов функции, и сохраняет ее адрес в funcAddress. Этот фрагмент кода показывает результат:

```
INT3          ; LocateFunctionHead откатилась назад сюда
PUSH EBP      ; и вернула этот адрес
MOV EBP, ESP
PUSH [-1]
--snip--
PUSH [0x64]   ; LocatePacketCreation вернула этот адрес
CALL [something]
```

Этот 35-строчный скрипт может автоматически находить 15 различных функций менее чем за минуту.

## Поиск строк

Этот следующий сценарий Lua сканирует память игры в поиске текстовых строк. Он работает примерно так же, как сканер памяти Cheat Engine при использовании типа значения строки.

---

```
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 + 0xffffffff), chunkStep do
        local chunk = readBytes(address, chunkSize, true)
        if (not chunk) then break end
❺    for c = 0, chunkSize-len do
        checkForString(address , chunk, c, str, len)
    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
❻    print(string.format("\t0x%x", address + start))
end

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

---

После получения базового адреса определяется функция `findString()` ❶, которая принимает строку `str` в качестве параметра. Эта функция выполняет цикл по памяти игры ❷ блоками по 4 096 байтов. Блоки сканируются последовательно, каждый новый блок начинается на (длину `str`) байтов раньше конца предыдущего ❸, чтобы предотвратить пропуск строки, которая начинается в одном блоке и заканчивается в другом.

Так как `findString()` считывает каждый блок, он проходит по каждому байту до точки перекрытия в блоке ❹, передавая каждый подблок в функцию `checkForString()` ❺. Если `checkForString()` находит совпадение подблока со `str`, она выводит адрес этого подблока в консоль ❻.

Наконец, для поиска всех адресов, которые ссылаются на строки "hello" и "world", вызываются функции `findString("hello")` ❼ и `findString("world")` ➋. Используя этот код для поиска встроенных отладочных строк и сочетая его с предыдущим кодом для поиска заголовков функций, я могу находить большое количество внутренних функций внутри игры за считанные секунды.

### Оптимизация кода работы с памятью

Из-за высокой нагрузки при чтении памяти оптимизация имеет чрезвычайно важное значение, когда вы пишете код, выполняющий множественные операции чтения памяти. В предыдущем примере кода обратите внимание, что функция **findString()** не использует встроенную в Lua функцию **readString()**. Вместо этого она читает блоки памяти и выполняет поиск нужной строки в них. Давайте разберем это в числах.

Сканирование с использованием **readString()** попыталось бы прочитать строку длиной **len** байтов по **каждому** возможному адресу памяти. Это означало бы чтение **(0x2FFFFF + len + 1)** байтов. Однако **findString()** считывает блоки размером **4 096** байтов и выполняет локальный поиск в них. Это означает, что максимум он прочитал бы **(0x2FFFFF ÷ 4 096 + (0x2FFFFF ÷ (4 096 - 1))) \* len** байтов.

При поиске строки длиной **10** байтов число прочитанных байтов составило бы **503 316 480** и **30 458 923 528** соответственно.

**findString()** не только считывает данных на порядок меньше, но и выполняет меньше чтений памяти. Чтение в блоках по **4 096** байтов потребовало бы **(0x2FFFFF ÷ 4 096 - 1)** чтений. Сравните это с поиском через **readString()**, который потребовал бы **0x2FFFFF** операций чтения. Такое сканирование делает чтение памяти **значительно более затратным**, чем просто увеличение размера блока.

(Примечание: Я выбрал 4 096 байтов произвольно. Возможно, такой относительно малый размер блока делает процесс чтения слишком медленным, а чтение сразу четырех страниц памяти ради поиска одной строки может оказаться неэффективным.)

## Заключительные мысли

К этому моменту у вас должно быть базовое понимание Cheat Engine и принципов его работы. Cheat Engine — это очень важный инструмент в вашем арсенале, и я призываю вас получить практический опыт работы с ним, следуя разделам "Основное редактирование памяти" на странице 11 и "Сканирование указателей" на странице 18, а также экспериментируя с ним самостоятельно.

## 2) Отладка игр с помощью OllyDbg



Вы можете лишь поверхностно понять, что происходит во время работы игры, используя Cheat Engine, но с хорошим отладчиком можно копнуть глубже и разобраться в структуре игры и ее процессе выполнения. Это делает OllyDbg незаменимым инструментом в вашем арсенале хакера игр.

Он оснащен множеством мощных инструментов, таких как условные точки останова, поиск ссылок на строки, поиск паттернов в ассемблере и трассировка выполнения, что делает его надежным отладчиком на уровне ассемблера для 32-битных Windows-приложений.

Я подробно разберу структуру низкоуровневого кода в главе 4, но для этой главы предполагается, что вы хотя бы немножко знакомы с современными отладчиками на уровне кода, такими как встроенный в Microsoft Visual Studio. OllyDbg функционально похож на них, за одним важным исключением:

Он работает напрямую с ассемблерным кодом приложения, даже если отсутствует исходный код и/или отладочные символы, что делает его идеальным инструментом для изучения внутренних механизмов игры. Ведь игровые компании редко (или глуко) включают отладочные символы в свои игры!

В этой главе я расскажу об интерфейсе OllyDbg, покажу, как использовать его основные функции отладки, разберу его систему выражений и приведу несколько практических примеров, как применить его в хакинге игр. В завершение я поделюсь полезными советами и предложу вам тестовую игру, специально созданную для отладки в OllyDbg.

**НОТЕ** Эта глава посвящена OllyDbg 1.10 и может быть не совсем точной для более поздних версий. Я использую именно эту версию, потому что на момент написания плагины для OllyDbg 2 всё ещё намного уступают тем, что доступны для OllyDbg 1.

Когда вы почувствуете, что разобрались с интерфейсом и возможностями OllyDbg, вы можете попробовать его в реальной игре, выполнив задание «Изменение оператора if()» на странице 46.

## Краткий обзор пользовательского интерфейса OllyDbg

Перейдите на сайт OllyDbg (<http://www.ollydbg.de/>), скачайте и установите OllyDbg, затем откройте программу. В верхней части окна должна появиться панель инструментов, показанная на Рисунке 2-1.

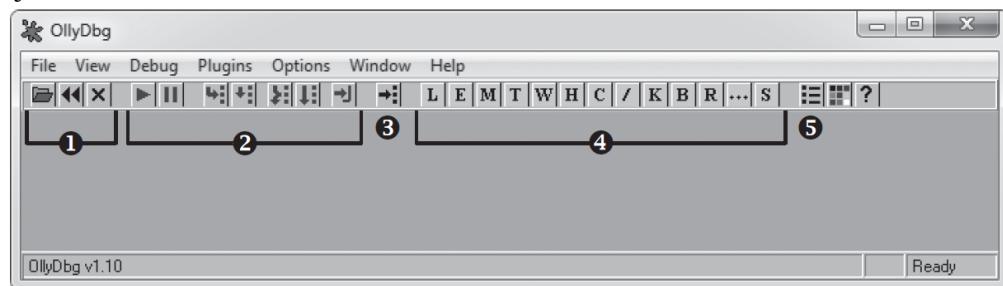


Рисунок 2-1: Главное окно OllyDbg

Эта панель инструментов содержит кнопки управления программой ①, кнопки отладки ②, кнопку перехода ③, кнопки управления окнами ④ и кнопку настроек ⑤.

Три кнопки управления программой позволяют:

Открыть исполняемый файл и присоединиться к создаваемому им процессу

Перезапустить текущий процесс

Завершить выполнение текущего процесса

Эти же функции можно выполнить с помощью горячих клавиш:  
F3

CTRL+F2

ALT+F2

Чтобы присоединиться к уже запущенному процессу, выберите File ▶ Attach.

Кнопки отладки управляют действиями отладчика. В таблице 2-1 описано, что делает каждая кнопка, а также указаны соответствующие горячие клавиши и их функции.

В этой таблице также перечислены три полезные функции отладки, которые отсутствуют на панели инструментов отладчика.

**Таблица 2-1:** Кнопки отладки и другие функции отладчика

Кнопка	Горячая клавиша	Функция
Play	F9	Возобновляет нормальное выполнение процесса.
Pause	F12	Приостанавливает выполнение всех потоков в процессе и открывает окно CPU с инструкцией, которая выполнялась в момент остановки.
Step into	F7	Пошагово выполняет следующую операцию (входит в вызовы функций).
Step over	F8	Переходит к следующей операции в текущей области видимости (пропускает вызовы функций).
Trace into	CTRL+F11	Выполняет глубокий трассировочный запуск, отслеживая все выполняемые операции.
Trace over	CTRL+F12	Выполняет пассивный трассировочный запуск, отслеживая только операции в пределах текущей области видимости.
Execute until return	CTRL+F9	Выполняет код до тех пор, пока не встретит операцию возврата в текущей области.
	CTRL+F7	Автоматически выполняет пошаговое выполнение каждой операции, следуя за выполнением в окне дизассемблирования. Делает выполнение похоже на анимацию.
	CTRL+F8	Также анимирует выполнение, но пропускает вызовы функций вместо пошагового входа в них.
ESC		Останавливает анимацию, приостанавливая выполнение на текущей операции.

Кнопка Go открывает диалоговое окно, запрашивающее шестнадцатеричный адрес. После ввода адреса OllyDbg откроет окно CPU и отобразит дизассемблированный код по указанному адресу. Когда окно CPU активно, вы также можете вывести эту информацию с помощью горячей клавиши CTRL+G.

Кнопки управления окнами открывают различные контрольные окна (control windows), которые отображают полезную информацию о процессе, который вы отлаживаете, а также раскрывают дополнительные функции отладки, такие как возможность устанавливать точки останова (breakpoints).

OllyDbg включает в себя 13 контрольных окон, которые могут быть открыты одновременно внутри многооконного интерфейса.

**Таблица 2-2: Окна управления OllyDbg**

Окно	Горячая клавиша	Функция
Журнал (Log)	ALT+L	Отображает список сообщений журнала, включая отладочные принты, события потоков, события отладчика, загрузку модулей и многое другое.
Модули (Modules)	ALT+E	Отображает список всех исполняемых модулей, загруженных в процесс. Двойной щелчок по модулю откроет его в окне CPU.
Карта памяти (Memory map)	ALT+M	Отображает список всех блоков памяти, выделенных процессу. Двойной щелчок по блоку в списке откроет окно дампа этого блока памяти.
Потоки (Threads)	-	Отображает список потоков, работающих в процессе. Для каждого потока в списке процесс имеет структуру, называемую <b>Thread Information Block (TIB)</b> . OllyDbg позволяет вам просматривать TIB каждого потока: просто кликните правой кнопкой мыши на поток и выберите <b>Dump thread data block</b> .
Окна (Windows)	-	Отображает список оконных дескрипторов, управляемых процессом. Щелкните правой кнопкой мыши по окну в этом списке, чтобы перейти к нему или установить точку останова в классе процесса, которому оно принадлежит (полезно, когда сообщение отправляется в окно).
Дескрипторы (Handles)	-	Отображает список дескрипторов, используемых процессом. (Обратите внимание, что <i>Process Explorer</i> имеет гораздо лучший список дескрипторов, чем OllyDbg, как обсуждается в главе 3.)
CPU (ЦП)	ALT+C	Отображает основной интерфейс дизассемблера и элементы управления анализатором отладчика.
Патчи (Patches)	CTRL+P	Отображает список любых изменений ассемблерного кода, которые были сделаны в модулях внутри процесса.
Стек вызовов (Call stack)	ALT+K	Отображает стек вызовов для активного потока. Окно обновляется при переключении потоков.
Точки останова (Breakpoints)	ALT+B	Отображает список активных точек останова отладчика и позволяет управлять ими в этом окне.
Ссылки (References)	-	Отображает список ссылок, который обычно показывает результаты поиска по многим различным типам данных. Он открывается как собственное окно при запуске поиска.
Трассировка выполнения (Run trace)	-	Отображает список операций, выполненных трассировщиком отладчика.
Исходный код (Source)	-	Отображает исходный код дизассемблированного модуля, если присутствует база данных отладки программы.

Наконец, кнопка **Settings** (Настройки) открывает окно настроек OllyDbg. Пока что оставьте настройки по умолчанию. Теперь, когда вы познакомились с основным окном OllyDbg, давайте подробнее рассмотрим окна CPU, Patches и Run trace. Вы будете активно использовать эти окна в качестве игрового хакера, и умение ориентироваться в них — ключ к успеху.

## Окно CPU в OllyDbg

Окно CPU, показанное на рисунке 2-2, — это место, где игровые хакеры проводят большую часть своего времени в OllyDbg, потому что оно является главным управляемым окном для функций отладки.

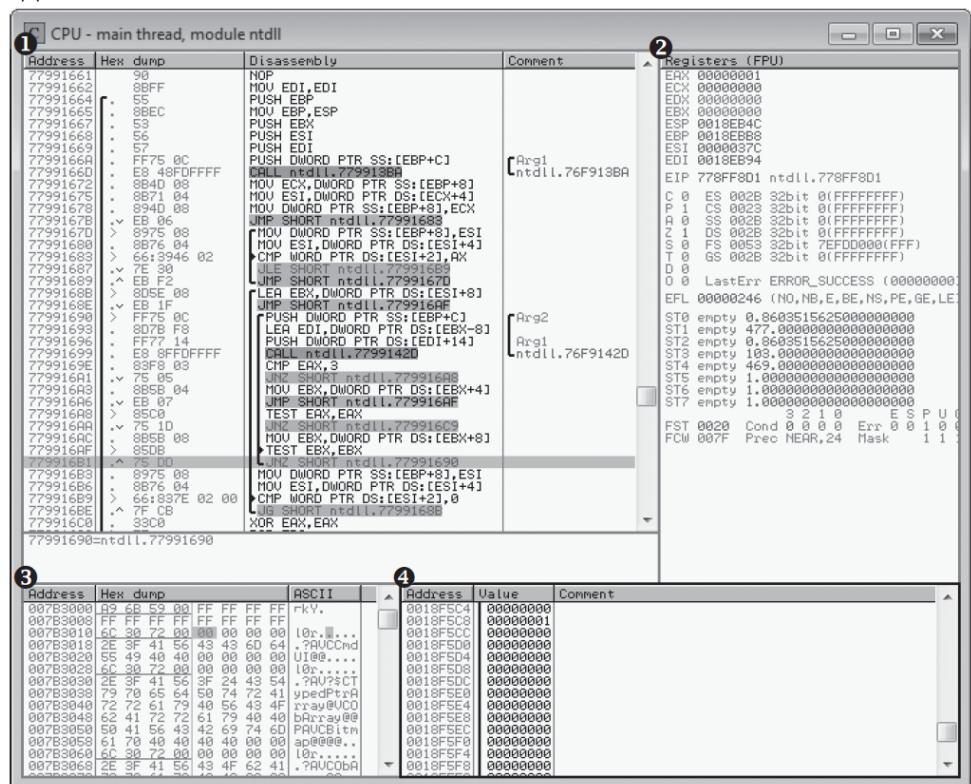


Рисунок 2-2: Окно процессора OllyDbg

Это окно содержит четыре отдельных панели управления: панель дизассемблера ①, панель регистров ②, панель дампа ③ и панель стека ④. Эти четыре панели охватывают основные функции отладки в OllyDbg, поэтому важно знать их досконально.

## Просмотр и навигация по ассемблерному коду игры

Вы будете перемещаться по игровому коду и управлять большинством аспектов отладки через панель дизассемблера OllyDbg. В этой панели отображается ассемблерный код для текущего модуля, а его данные аккуратно представлены в таблице, состоящей из четырёх отдельных столбцов: Адрес, Шестнадцатеричный дамп, Дизассемблированный код и Комментарий.

Столбец Адрес показывает адреса памяти каждой выполняемой операции в игре, к которой вы подключены. Вы можете дважды щёлкнуть по адресу в этом столбце, чтобы переключить его в режим базового адреса. Когда адрес установлен как базовый,

столбец Адрес показывает все остальные адреса как смещения относительно него.

Столбец Шестнадцатеричный дамп отображает байтовый код каждой выполняемой операции, группируя коды операций и параметры соответствующим образом. Чёрные фигурные скобки, охватывающие несколько строк в левой части этого столбца, обозначают границы функций.

Операции, в которые совершаются прыжки, показаны правосторонней стрелкой внутри этих скобок. Операции, выполняющие прыжки, отображаются с верхними или нижними стрелками, в зависимости от направления, в котором осуществляется переход.

Например, в рисунке 2-2 инструкция по адресу 0x779916B1 (выделена серым цветом) имеет стрелку, направленную вверх, что означает безусловный переход вверх. Можно думать о прыжке как об операторе `goto`.

Столбец Дизассемблированный код показывает ассемблерский код каждой операции, выполняемой в игре. Например, вы можете подтвердить, что инструкция по адресу 0x779916B1 в рисунке 2-2 действительно является прыжком, посмотрев на дизассемблированный код, где показана команда `JNZ (jump if nonzero, переход, если не ноль)`.

Чёрные фигурные скобки в этом столбце обозначают границы циклов. Стрелки, направленные вправо, прикреплённые к этим скобкам, указывают на условные выражения, которые контролируют, выполняется ли цикл дальше или завершается. Три правосторонние стрелки в этом столбце на рисунке 2-2 указывают на инструкции `CMP` (сравнение) и `TEST`, которые используются в ассемблере для сравнения значений.

Столбец Комментарий содержит читаемые человеком комментарии о каждой выполняемой операции в игре. Если OllyDbg обнаруживает известную API-функцию, он автоматически вставляет в этот столбец её имя.

Кроме того, если он успешно определяет передачу аргументов в функцию, он также вставит их сюда (например, `Arg1`, `Arg2`, ..., `ArgN`). Вы можете дважды щёлкнуть в этом столбце, чтобы добавить настраиваемый комментарий. Чёрные фигурные скобки в этом столбце отмечают предполагаемые границы параметров вызова функции.

#### НОТЕ

*OllyDbg выводит границы функций, направления переходов, структуры циклов и параметры функций во время анализа кода, поэтому, если в этих столбцах отсутствуют граничные линии или стрелки переходов, просто нажмите `CTRL-A`, чтобы запустить анализ кода в бинарном файле.*

Когда окно дизассемблера (Disassembler pane) активно, есть несколько горячих клавиш, которые можно использовать для быстрого перемещения и управления отладчиком.

Нажмите `F2`, чтобы установить или убрать точку останова. Нажмите `Shift+F12`, чтобы установить условную точку останова.

Используйте - (дефис) для перехода назад и + (плюс) для перехода вперед (эти клавиши работают так же, как в веб-браузере).

Используйте \* (звездочка) для перехода к EIP (указатель инструкции в архитектуре x86). Используйте Ctrl+ (дефис) для перехода к предыдущей функции и Ctrl++ для перехода к следующей функции. Окно дизассемблера (Disassembler pane) также может заполнять окно Ссылки (References window) различными типами найденных ссылок. Если вы хотите изменить содержимое окна Ссылки (References window), щелкните правой кнопкой мыши в окне дизассемблера, наведите курсор на Поиск (Search), чтобы развернуть меню, и выберите один из следующих вариантов:

#### Все межмодульные вызовы (All intermodular calls)

Ищет все вызовы функций в удаленных модулях. Это может, например, позволить вам увидеть, где в игре вызываются Sleep(), PeekMessage() или любая другая функция Windows API, что дает возможность инспектировать их или устанавливать точки останова.

#### Все команды (All commands)

Ищет все вхождения заданной операции, записанной на ассемблере, где добавленные операторы CONST и R32 будут соответственно соответственно постоянному значению или значению регистра. Один из вариантов использования этой опции — поиск команд, таких как MOV [0xDEADBEEF], CONST.

MOV [0xDEADBEEF], R32; и MOV [0xDEADBEEF], [R32+CONST]. Позволяет перечислить все операции, которые изменяют память по адресу 0xDEADBEEF, которым может быть что угодно, включая адрес здоровья вашего персонажа.

#### Все последовательности (All sequences)

Ищет все вхождения заданной последовательности операций. Это похоже на предыдущие опции, но позволяет вам указывать несколько команд.

Все константы (All constants) Ищет все вхождения заданной шестнадцатеричной константы. Например, если вы введете адрес здоровья вашего персонажа, это выведет все команды, которые обращаются к нему напрямую.

#### Все переключатели (All switches)

Ищет все блоки переключателей switch-case.

#### Все найденные строки (All referenced text strings)

Ищет все строки, упомянутые в коде. Вы можете использовать этот вариант для поиска всех найденных строк и увидеть, что код с ними делает. Это может быть полезно для сопоставления внутриигровых сообщений с кодом, который их отображает. Эта опция также удобна для поиска сообщений об ошибках или ведения журналов, которые могут значительно помочь в определении назначения частей кода. Окно дизассемблера (Disassembler pane) также может заполнять окно Имена (Names window) всеми метками в текущем модуле (Ctrl+N) или всеми известными метками во всех модулях (Поиск → Имя в модуле

(Search → Name in all modules)).

Известные API-функции Windows будут автоматически помечены их именами, а также вы можете добавить метку к команде, подсветив ее, нажав Shift+;, и введя метку, когда появится запрос.

Когда к адресу добавлена метка, она будет отображаться вместо самого адреса.

Один из способов использования этой функции — присвоение имен функциям, которые вы проанализировали (например, просто установите метку на первой команде в функции), чтобы затем видеть их имена, когда другие функции вызывают их.

## Просмотр и редактирование содержимого регистров (Viewing and Editing Register Contents)

Окно Регистры (Registers pane) отображает содержимое восьми регистров процессора, всех флагов, шести сегментных регистров, последнего кода ошибки Windows и EIP. Помимо этих значений, это окно может отображать регистры с плавающей точкой (Floating-Point Unit (FPU) registers) или отладочные регистры (debug registers); щелкните по заголовку окна, чтобы выбрать, какие регистры отображать. Значения в этом окне заполняются только в том случае, если вы заморозили процесс (freeze your process).

Значения, выделенные красным, были изменены с момента последнего обновления. Дважды щелкните по значениям в этом окне, чтобы их отредактировать.

## Просмотр и поиск памяти игры (Viewing and Searching a Game's Memory)

Окно Дамп памяти (Dump pane) отображает дамп памяти по указанному адресу. Чтобы перейти к адресу и отобразить содержимое памяти, нажмите Ctrl+G, затем введите адрес в появившемся окне. Вы также можете перейти по адресу из других окон процессора, щелкнув правой кнопкой мыши на столбце Адрес (Address column) и выбрав Перейти в дамп (Follow in dump).

Хотя в окне Дамп памяти (Dump pane) всегда присутствуют три столбца, единственный, который вам следует всегда видеть, — это столбец Адреса (Address column), который ведет себя аналогично аналогичному столбцу в окне дизассемблера. Выбранный тип данных (data display type) определяет содержимое двух других столбцов. Щелкните правой кнопкой мыши по окну Дамп памяти (Dump pane), чтобы изменить тип отображения данных (display type); например, чтобы установить шестнадцатеричный (Hex) и ASCII (8 байт), выберите Hex > Hex/ASCII (8 bytes).

Вы можете установить точку останова в памяти (memory breakpoint) на адресе в окне Дамп памяти (Dump pane), щелкнув правой кнопкой мыши по адресу и развернув подменю Точка останова (Breakpoint submenu). Выберите Memory > On access, чтобы установить точку останова на любое обращение к этому адресу, или Memory > On write, чтобы установить точку останова только на запись по этому адресу. Чтобы удалить точку останова, выберите Удалить точку останова (Remove memory breakpoint) в

том же меню; этот параметр отображается только для адресов, на которых уже установлена точка останова.

При выборе одного или нескольких значений в окне Дамп памяти (Dump pane) можно нажать Ctrl+R, чтобы выполнить поиск ссылок на выбранные значения; результаты поиска появятся в окне Ссылки (References window). Вы также можете выполнить поиск значений в этом окне с помощью Ctrl+B (для поиска бинарных строк) и Ctrl+N (для поиска имен). Чтобы найти следующее совпадение в поиске, нажмите Ctrl+L. Нажатие Ctrl+E позволяет редактировать любое выбранное значение.

**Н О Т Е** *Окна дампа, которые можно открыть из окна Память (Memory window), работают так же, как и окно Дамп памяти (Dump pane).*

## Просмотр стека вызовов игры (Viewing a Game's Call Stack)

Последним окном процессора является Окно стека (Stack pane), и, как следует из названия, оно отображает стек вызовов (call stack). Как и окна Дамп памяти (Dump pane) и Дизассемблер (Disassembler pane), Окно стека (Stack pane) имеет столбец Адреса (Address column). В нем также есть столбец Значений (Value column), который отображает стек в виде массива 32-битных целых чисел, и столбец Комментариев (Comment column), который показывает возвращаемые адреса (return addresses), известные имена функций (known function names) и другие информативные метки.

Окно стека (Stack pane) поддерживает все горячие клавиши, доступные в Окне дампа (Dump pane), за исключением Ctrl+N.

### Многоклиентское патчинг (Multiclient Patching)

Один из типов взлома, называемый **многоклиентским патчем (multiclient patch)**, удаляет ограничение на запуск одной копии игры, перезаписывая соответствующий участок кода в бинарном файле игры **NOP-операцией (no-operation code)**. Это позволяет пользователю запускать несколько игровых клиентов одновременно, даже если это обычно запрещено.

Так как код, который отвечает за ограничение количества запущенных клиентов, обычно выполняется **очень рано после запуска игры**, ботам может быть **почти невозможно внедрить свой патч вовремя**. Самым простым способом обойти это является **применение многоклиентских патчей прямо в OllyDbg и сохранение изменений непосредственно в бинарный файл игры**.

## Создание патчей кода (Creating Code Patches)

Функция Патчи кода (Code patches) в OllyDbg позволяет изменять машинный код игры, устранивая необходимость разрабатывать специальные инструменты для взлома. Это делает прототипирование манипуляций с управлением потоком кода (prototyping control flow hacks) — например, изменение игровой логики, x86 протоколов сборки (x86 assembly protocols) и обычных бинарных конструкций (common binary constructs) — гораздо проще.

Создатели читов обычно включают идеально проработанные патчи (perfected patches) как опциональные функции в набор инструментов бота (bot's tool suite), но в некоторых случаях сохранение патчей (making those features persistent) может быть удобнее для конечного пользователя. К счастью, OllyDbg позволяет изменять, тестировать и навсегда сохранять изменения кода в исполняемом файле игры (permanently save code modifications to an executable binary) без использования сторонних инструментов.

Чтобы применить патч:

Перейдите к строке кода в Окне процессора (CPU window), которую хотите изменить.

Дважды щелкните по инструкции, чтобы изменить ее.

Ведите новую ассемблерную команду (assembly instruction) в появившемся всплывающем окне.

Нажмите Собрать (Assemble).

Как показано на Рисунке 2-3 (Figure 2-3).

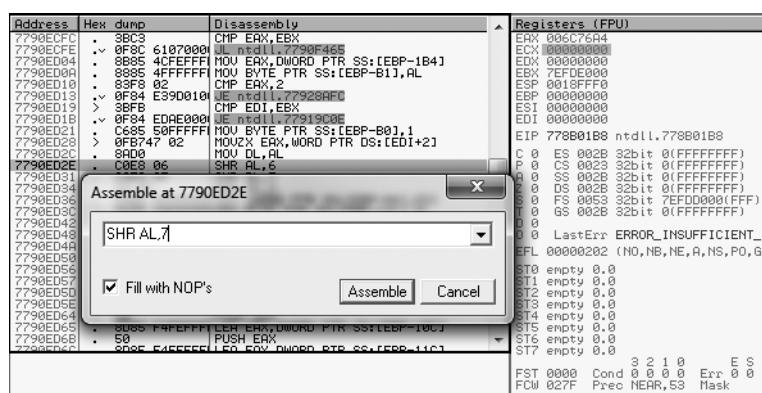


Рисунок 2-3: Размещение патча с помощью OllyDbg

## Внимание к размеру патча

Всегда обращайте внимание на размер вашего патча — нельзя просто произвольно изменять размеры и перемещать машинный код (resize and move around assembled code), как вам захочется.

Если патч больше исходного кода (Patches larger than the code you intend to replace), он может переполнить последующие операции (overflow into subsequent operations), что приведет к удалению важного функционала игры (removing critical

functionality). Если патч меньше исходного кода (Patches smaller than the operations you intend to replace), то это безопасно, при условии, что включена опция Заполнить NOP-ами (Fill with NOPs). Эта опция заменяет неиспользуемые байты на no-operation (NOP) команды, которые являются однобайтовыми операциями, не выполняющими никаких действий.

Все примененные патчи отображаются в Окне патчей (Patches window) вместе с:

- Адресом (address)
- Размером (size)
- Состоянием (state)
- Старым кодом (old code)
- Новым кодом (new code)
- Комментариями (comment)

Чтобы получить доступ к списку горячих клавиш (list of hotkeys), выберите патч в Окне патчей (Patches window). Полный список горячих клавиш приведен в Таблице 2-3 (Table 2-3).

**Таблица 2-3: Горячие клавиши окна патчей (Table 2-3: Patches Window Hotkeys)**

Клавиша (Operator)	Функция (Function)
ENTER	Переход к патчу в дизассемблере.
Пробел (spacebar)	Включает или отключает патч.
F2	Устанавливает точку останова на патче.
Shift+F2	Устанавливает условную точку останова на патче.
Shift+F4	Устанавливает условную логическую точку останова на патче.
DEL	Удаляет запись о патче только из списка.

В OllyDbg вы также можете сохранить патчи напрямую в бинарный файл (save your patches directly to the binary). Щелкните правой кнопкой мыши в окне Дизассемблера (Disassembler pane). Выберите Копировать в исполняемый файл (Copy to executable) > Все изменения (All modifications). Если вы хотите сохранить только некоторые патчи: Выделите их в Окне дизассемблера (Disassembler pane). Выберите Копировать в исполняемый файл (Copy to executable) > Выбранное (Selection).

### **Определение размера патча (Determining Patch Size)**

Есть несколько способов определить, будет ли ваш патч отличаться по размеру от оригинального кода (a different size than the original code).

Например, на Рисунке 2-3 (Figure 2-3) показано изменение команды по адресу 0x7790E2DE:

Исходная команда: SHR AL, 6

Измененная команда: SHR AL, 7

Если посмотреть на байты слева от команды, то можно увидеть, что 3 байта зарезервированы для команды. Это означает, что новая команда должна занимать 3 байта, либо быть дополнена NOPами, если занимает меньше.

Байты в этом случае представлены в двух столбцах:

Первый столбец содержит 0xC0 и 0x08, которые обозначают команду SHR и первый operand (AL).

Второй столбец содержит 0x06, который представляет исходный operand.

Так как второй столбец содержит одноразрядный байт, заменяемый operand должен быть 1 байт (от 0x00 до 0xFF). Если бы этот столбец содержал 0x00000006, то заменяемый operand мог бы быть до 4 байтов.

Обычно патчи кода используют:

NOPы для полного удаления команды (оставляя пустую ячейку, которая заполняется NOPами).

Замену одного операнда (однобайтовое изменение).

Таким образом, этот метод проверки размера патча почти всегда эффективен (this method of checking patch size is almost always effective).

### **Пошаговый трассинг ассемблерного кода (Tracing Through Assembly Code)**

Когда вы запускаете трассировку (trace) программы, OllyDbg выполняет пошаговое выполнение (single-step execution) каждой операции, сохраняя данные о каждой. Когда трассировка завершена, собранные данные отображаются в Окне трассировки (Run trace window), как показано на Рисунке 2-4 (Figure 2-4).

Back	Thread	Module	Address	Command	Modified registers
118.	Main	KERNEL32	7569FA8C	MOV ECX, ESP	EBP=0018F000 EDX=00000000
119.	Main	KERNEL32	7569FA8D	MOV ECX, DWORD PTR FS:[18]	ECX=7EFDDE0000
116.	Main	KERNEL32	7569FA94	MOV ECX, DWORD PTR DS:[ECAX+30]	ECX=7EFDDE0000
115.	Main	KERNEL32	7569FA97	MOV ECX, DWORD PTR DS:[ECAX+10]	ECX=00C21B48
114.	Main	KERNEL32	7569FA99	MOV ECX, DWORD PTR SS:[EBP+8]	EBP=0018F000
113.	Main	KERNEL32	7569FA9D	MOV DWORD PTR DS:[ECAX+44]	EDX=00C2285E
112.	Main	KERNEL32	7569FA9E	MOV DWORD PTR DS:[ECAX+44], EDX	EDX=00C2283E
111.	Main	KERNEL32	7569FA9F	MOV DWORD PTR DS:[ECAX+44], EDX	EDX=00C227D6
110.	Main	KERNEL32	7569FAAC	MOV EDX, DWORD PTR DS:[ECX+7C]	EDX=00C2283E
109.	Main	KERNEL32	7569FAAF	MOV DWORD PTR DS:[ECAX+81]	EDX=00C227D6
108.	Main	KERNEL32	7569FAE2	MOV EDX, DWORD PTR DS:[ECX+74]	EDX=00C227D6
107.	Main	KERNEL32	7569FAE5	MOV DWORD PTR DS:[ECAX+81], EDX	EDX=1F4A6AE7
106.	Main	KERNEL32	7569FAE8	MOV DWORD PTR DS:[ECAX+101]	EDX=00000000
105.	Main	KERNEL32	7569FAE9	MOV ECX, DWORD PTR DS:[ECAX+101]	EDX=FFFFFFFFFF
104.	Main	KERNEL32	7569FAE8	MOV ECX, DWORD PTR DS:[ECAX+50]	EDX=75D36901
103.	Main	KERNEL32	7569FAC1	MOV DWORD PTR DS:[ECAX+143]	EDX=75D36901
102.	Main	KERNEL32	7569FAC4	MOV EDX, DWORD PTR DS:[ECAX+54]	EDX=75D36901
101.	Main	KERNEL32	7569FAE9	MOV DWORD PTR DS:[ECAX+143]	EDX=75D36901
100.	Main	KERNEL32	7569FAE9	MOV EDX, DWORD PTR DS:[ECAX+54]	EDX=75D40AB0
99.	Main	KERNEL32	7569FAE9	MOV DWORD PTR DS:[ECAX+1C1]	EDX=00000000
98.	Main	KERNEL32	7569FAE8	MOV EDX, DWORD PTR DS:[ECX+5C]	EDX=00000000
97.	Main	KERNEL32	7569FA03	MOV DWORD PTR DS:[ECAX+201]	EDX=7790E5F1
96.	Main	KERNEL32	7569FA06	MOV EDX, DWORD PTR DS:[ECX+60]	EDX=00000000
95.	Main	KERNEL32	7569FA07	MOV DWORD PTR DS:[ECAX+201], EDX	EDX=00000000
94.	Main	KERNEL32	7569FA0C	MOV EDX, DWORD PTR DS:[ECX+64]	EDX=01310836
93.	Main	KERNEL32	7569FA0F	MOV DWORD PTR DS:[ECAX+281]	EDX=00000001
92.	Main	KERNEL32	7569FAE2	MOV EDX, DWORD PTR DS:[ECX+68]	EDX=00000001
91.	Main	KERNEL32	7569FAE5	MOV DWORD PTR DS:[ECAX+2C]	EDX=00000001

Рисунок 2-4: Окно Выполнить трассировку

Окно Трассировки выполнения (Run trace window) организовано в виде следующих шести столбцов:

Back – Количество операций, зарегистрированных между выполненной операцией и текущим состоянием выполнения.

Thread – Поток, который выполнил операцию.

Module – Модуль, в котором находится операция.

Address – Адрес операции.

Command – Выполненная операция.

Modified registers – Регистры, измененные операцией, и их новые значения.

При взломе игр я считаю, что функция трассировки (trace feature) в OllyDbg очень эффективна для поиска путей указателей к динамической памяти, когда сканирование с помощью Cheat Engine не дает результата. Это работает потому, что можно проследить лог в окне трассировки (Run trace window backward) – от момента использования памяти до момента, когда она была разрешена из статического адреса.

Эта мощная функция ограничена только креативностью хакера (the creativity of the hacker using it). Хотя я обычно использую ее только для поиска путей указателей (pointer paths), я встречал еще несколько ситуаций, когда она оказывалась незаменимой (invaluable).

Анекдоты из “OllyDbg Expressions in Action” на странице 36 помогут разъяснить функциональность и мощность трассировки (illuminate the functionality and power of tracing).

## Выражения в OllyDbg (OllyDbg's Expression Engine)

OllyDbg включает кастомный движок выражений (custom expression engine), который компилирует и вычисляет сложные выражения (compile and evaluate advanced expressions) с простым синтаксисом (simple syntax). Этот движок выражений удивительно мощный (surprisingly powerful) и, при правильном использовании (when utilized properly), может стать ключевым различием между обычным пользователем OllyDbg и экспертом (the difference

between an average OllyDbg user and an OllyDbg wizard).

Вы можете использовать этот движок для задания выражений во многих функциях, таких как:

Условные точки останова (conditional breakpoints)

Условные трассировки (conditional traces)

Плагин командной строки (command line plug-in)

Этот раздел представляет движок выражений и его возможности (introduces the expression engine and the options it provides).

#### **NOTE**

Части этого раздела основаны на официальной документации по выражениям (*official expressions documentation*) ([http://www.ollydbg.de/Help/i\\_Expressions.htm](http://www.ollydbg.de/Help/i_Expressions.htm)). Однако я обнаружил, что некоторые компоненты, определенные в документации, похоже, не работают (*a few of the components defined in the documentation don't seem to work*), по крайней мере, в OllyDbg v1.10. Два примера — это тип данных INT и тип данных ASCII (*the INT and ASCII data types*), которые должны быть заменены псевдонимами LONG и STRING (*must be substituted with the aliases LONG and STRING*). По этой причине я включаю только те компоненты, которые я лично тестировал и полностью понимаю (*only components that I've personally tested and fully understand*).

## Использование выражений в точках останова (Using Expressions in Breakpoints)

Когда условная точка останова (conditional breakpoint) активирована, OllyDbg предлагает ввести выражение для условия (an expression for the condition); именно здесь чаще всего используются выражения.

Когда эта точка останова (breakpoint) срабатывает, OllyDbg незаметно приостанавливает выполнение (silently pauses execution) и вычисляет выражение (evaluates the expression).

Если результат вычисления ненулевой (nonzero), выполнение остается приостановленным, и вы увидите, что точка останова сработала (got triggered).

Если результат вычисления равен 0, OllyDbg незаметно возобновляет выполнение (silently resumes execution), как будто ничего не произошло.

Так как в игре может выполняться огромное количество инструкций в секунду (huge number of executions within a game every second), часто можно обнаружить, что участок кода выполняется слишком часто (a piece of code is executed in far too many contexts), чтобы использовать его в качестве точки останова для извлечения нужных данных.

Условная точка останова (A conditional breakpoint), в сочетании с хорошим пониманием окружающего кода (paired with a good understanding of the code surrounding it), — это надежный способ избежать таких ситуаций (a foolproof way to avoid these situations).

## Использование операторов в движке выражений (Using Operators in the Expression Engine)

Для числовых типов данных (numeric data types), выражения OllyDbg поддерживают общие операторы в стиле C (OllyDbg expressions support general C-style operators), как показано в Таблице 2-4 (Table 2-4).

Хотя четкой документации по приоритету операторов нет (there is no clear documentation on the operator precedence), OllyDbg, похоже, следует приоритету С и поддерживает использование скобок (OllyDbg seems to follow C-style precedence and can use parenthesized scoping).

**Таблица 2-4: Числовые операторы OllyDbg (Table 2-4: OllyDbg Numeric Operators)**

Оператор (Operator)	Функция (Function)
$a == b$	Возвращает 1, если a равно b, иначе возвращает 0.
$a != b$	Возвращает 1, если a не равно b, иначе возвращает 0.
$a > b$	Возвращает 1, если a больше b, иначе возвращает 0.
$a < b$	Возвращает 1, если a меньше b, иначе возвращает 0.
$a >= b$	Возвращает 1, если a больше или равно b, иначе возвращает 0.
$a <= b$	Возвращает 1, если a меньше или равно b, иначе возвращает 0.
$a \&& b$	Возвращает 1, если a и b оба ненулевые, иначе возвращает 0.
$a    b$	Возвращает 1, если a или b ненулевые, иначе возвращает 0.
$a ^ b$	Возвращает результат XOR(a, b).
$a \% b$	Возвращает результат MODULUS(a, b).
$a \& b$	Возвращает результат AND(a, b).
$a   b$	Возвращает результат OR(a, b).
$a << b$	Возвращает результат сдвига b битов влево.
$a >> b$	Возвращает результат сдвига b битов вправо.
$a + b$	Возвращает сумму a плюс b.
$a - b$	Возвращает разность a минус b.
$a / b$	Возвращает результат деления a на b.
$a * b$	Возвращает произведение a на b.
$+a$	Возвращает signed представление a.
$-a$	Возвращает $a * -1$ .
$!a$	Возвращает 1, если a равно 0, иначе возвращает 0.



Для строк доступны только операторы  $==$  и  $!=$ , которые оба

подчиняются следующим правилам:

Сравнения строк нечувствительны к регистру (String comparisons are case insensitive).

Если один из операндов является строковым литералом, сравнение завершится после достижения длины литерала. В результате выражение [STRING EAX]!="ABC123", где EAX — это указатель на строку ABC123YZ, будет вычисляться в 1 вместо 0 (will evaluate to 1 instead of 0).

Если для операнда в строковом сравнении не указан тип, а другой operand является строковым литералом (например, "MyString"!=EAX), сравнение сначала предположит, что нестроковый operand представлен в ASCII-кодировке (ASCII string), а если сравнение вернет 0, то оно попробует выполнить повторное сравнение, предполагая, что operand является строкой в Unicode (Unicode string).

Конечно, операторы не имеют смысла без operandов (operators aren't much use without operands). Давайте рассмотрим некоторые типы данных, которые можно вычислять в выражениях.

## Работа с базовыми элементами выражений (Working with Basic Expression Elements)

Выражения могут вычислять множество различных элементов, включая:

Регистр процессора (CPU registers): EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI.

Можно также использовать 1-байтовые и 2-байтовые регистры (например, AL для младшего байта и AX для младшего слова EAX).

EIP также можно использовать.

Сегментные регистры (Segment registers): CS, DS, ES, SS, FS, GS.

Регистры FPU (FPU registers): ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7.

Простые метки (Simple labels): могут быть именами API-функций, такими как GetModuleHandle, или пользовательскими метками.

Константы Windows (Windows constants): например, ERROR\_SUCCESS.

Целые числа (Integers): записываются в шестнадцатеричном формате (hexadecimal format) или десятичном формате с суффиксом d (decimal format if followed by a trailing decimal d) (например, FFFF или 65535d).

Числа с плавающей запятой (Floating-point numbers): поддерживают экспоненциальную запись в десятичном формате (allow exponents in decimal format) (например, 654.123e-5).

Строковые литералы (String literals): заключаются в кавычки (quotation marks) (например, "my string").

Движок выражений (The expressions engine) ищет перечисленные элементы в указанном порядке (in the order they're listed here).

Например:

Если у вас есть метка, совпадающая с именем константы Windows (a label that matches the name of a Windows constant), движок использует адрес метки вместо значения константы (the address of the label instead of the constant's value).

Однако если у вас есть метка с тем же именем, что и регистр (a label named after a register), например EAX, движок использует значение регистра, а не имя метки (uses the register value, not the label value).

## Доступ к содержимому памяти с помощью выражений (Accessing Memory Contents with Expressions)

Выражения OllyDbg достаточно мощные, чтобы включать чтение памяти, которое выполняется путем заключения адреса памяти или выражения, вычисляющегося в адрес в квадратные скобки.

Например, [EAX+C] и [401000] представляют содержимое памяти по адресам EAX+C и 401000.

Чтобы читать память как тип, отличный от DWORD, можно указать нужный тип перед скобками, например BYTE [EAX], или внутри них как первый токен, например [STRING ESP+C].

Поддерживаемые типы перечислены в Таблице 2-5.

**Таблица 2-5: Типы данных OllyDbg (OllyDbg Data Types)**

Тип данных	Интерпретация
BYTE	8-битное целое (без знака)
CHAR	8-битное целое (со знаком)
WORD	16-битное целое (без знака)
SHORT	16-битное целое (со знаком)
DWORD	32-битное целое (без знака)
LONG	32-битное целое (со знаком)
FLOAT	32-битное число с плавающей точкой
DOUBLE	64-битное число с плавающей точкой
STRING	Указатель на ASCII-строку (нулем-терминированную)
UNICODE	Указатель на Unicode-строку (нулем-терминированную)

Встраивание данных памяти в выражения OllyDbg чрезвычайно полезно при взломе игр, так как позволяет поручить отладчику проверять параметры персонажа — например, здоровье, имя, золото и другие значения, хранящиеся в памяти перед остановкой выполнения.

Пример использования можно найти в разделе "Приостановка выполнения при печати имени конкретного игрока" (Pausing Execution When a Specific Player's Name Is Printed) на странице 37.

## Выражения OllyDbg в действии (OllyDbg Expressions in Action)

Выражения OllyDbg используют синтаксис, схожий с языками программирования, поддерживая комбинирование и вложенность выражений.

Хакеры (а точнее, все хакеры) часто применяют их для создания условных точек останова (conditional breakpoints), как описано в разделе "Использование выражений в точках останова" (Using Expressions in Breakpoints) на странице 34.

Однако их можно применять и в других случаях, например, в плагине командной строки OllyDbg.

Выражения могут отображать свои результаты прямо в коде, позволяя легко читать память, анализировать значения, вычисляемые ассемблерным кодом, или быстро получать результаты математических выражений.

Кроме того, хакеры могут создавать интеллектуальные точки останова, независимые от расположения кода, комбинируя выражения с функцией трассировки (trace feature).

В этом разделе я поделюсь примерами использования движка выражений, который многократно помогал мне в работе.

Я разберу процесс мышления, покажу весь сеанс отладки и разложу выражения на составные части, чтобы продемонстрировать применение выражений OllyDbg в хакинге игр.

### НОТ

Эти примеры содержат некоторый ассемблерный код, но если у вас мало опыта работы с ассемблером, не переживайте. Просто игнорируйте мелкие детали и помните, что значения, такие как ECX, EAX и ESP, — это регистры процессора, как описано в разделе "Просмотр и редактирование содержимого регистров" (Viewing and Editing Register Contents) на странице 29. Там я объясню все остальное. Если вам что-то непонятно (например, оператор, элемент или тип данных в выражении), обратитесь к разделу "Выражения OllyDbg" (OllyDbg's Expression Engine) на странице 33.

## Приостановка выполнения при выводе имени конкретного игрока (Pausing Execution When a Specific Player's Name Is Printed)

Во время одной из сессий отладки мне нужно было разобраться, что именно происходит, когда игра выводит имена игроков на экран.

Конкретно, мне было необходимо вызвать точку останова до того, как игра нарисует имя "Player 1", игнорируя все другие имена, которые были нарисованы.

## Определение места для паузы (Figuring Out Where to Pause)

В качестве отправной точки я использовал Cheat Engine, чтобы найти адрес имени игрока 1 в памяти. Получив этот адрес, я использовал OllyDbg, чтобы установить точку останова в памяти на первом байте строки.

Каждый раз, когда эта точка останова срабатывала, я быстро

изучал ассемблерный код, чтобы определить, как именно игра использует имя "Player 1". В конечном итоге я обнаружил, что это имя использовалось непосредственно перед вызовом функции, которой я ранее дал имя printText(). Я нашел код, который рисовал имя на экране. Я удалил свою точку останова в памяти и заменил ее на точку останова в коде на вызове printText().

Возникла проблема:

Так как printText() находилась внутри цикла, который проходился по каждому игроку в игре,

Новая точка останова срабатывала каждый раз, когда рисовалось имя, — это происходило слишком часто.

Мне нужно было настроить её так, чтобы она срабатывала только для конкретного игрока.

Изучив ассемблерный код в месте предыдущей точки останова в памяти, я обнаружил, что имена игроков загружались с использованием следующего кода на ассемблере:

---

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

Регистр EAX содержал адрес массива данных игроков; я назову его playerStruct. Размер playerStruct составлял 0x90 байтов, регистр ECX содержал индекс итерации (известную переменную i), и каждое имя игрока

хранилось на смещении 0x50 байтов от начала соответствующей playerStruct. Это означало, что эта инструкция PUSH фактически помещала EAX[ECX].name (имя игрока с индексом i) в стек для передачи в вызов функции printText().

Цикл затем разбивался на следующий псевдокод:

---

```
playerStruct EAX[MAX_PLAYERS]; // this is filled elsewhere
for (int ①ECX = 0; ECX < MAX_PLAYERS; ECX++) {
    char* name = ②EAX[ECX].name;
    breakpoint(); // my code breakpoint was basically right here
    printText(name);
}
```

---

Чисто на основе анализа я определил, что функция playerStruct() содержала данные обо всех играх, а цикл проходил по всему количеству игроков (увеличивая ECX ①), извлекал имя персонажа ② для каждого индекса и выводил его на экран.

## Создание условной точки останова (Crafting the Conditional Breakpoint)

Зная это, чтобы приостановить выполнение только при отрисовке "Player 1", все, что мне нужно было сделать, — проверить текущее имя игрока перед выполнением точки останова.

В псевдокоде новая точка останова выглядела бы так:

---

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

---

Как только я определил форму своей новой точки останова, мне нужно было получить доступ к EAX[ECX].name изнутри цикла.

Вот где движок выражений OllyDbg (OllyDbg's expression engine) оказался полезным: я мог достичь своей цели, внесением небольших изменений (slight modifications) в выражение, которое использовал ассемблерный код, оставив его в следующем виде:

---

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

---

Я удалил точку останова в коде на printText() и заменил ее на условную точку останова, использующую это выражение.

OllyDbg теперь останавливался только в том случае, если строка, хранящаяся по адресу EAX + ECX\*0x90 + 0x50, совпадала с "Player 1". Эта точка останова срабатывала только тогда, когда рисовался "Player 1", что позволяло мне продолжать анализ.

Хотя создание этой точки останова требовало значительных усилий, с опытом этот процесс становится таким же интуитивным, как написание кода. Опытные хакеры могут делать это за считанные секунды.

На практике эта точка останова позволила мне проанализировать определенные значения в функции playerStruct() для "Player 1", как только он появлялся на экране.

Этот метод был важен, так как значения этих состояний были актуальны для моего анализа только в первые несколько кадров после появления игрока.

Творческое использование точек останова позволяет анализировать любые сложные игровые механики.

## Приостановка выполнения при снижении здоровья персонажа (Pausing Execution When Your Character's Health Drops)

Во время одного из сеансов отладки мне нужно было найти первую функцию, которая вызывается после того, как здоровье моего персонажа упало ниже максимума. Я знал два способа решения этой проблемы:

Найти каждый участок кода, который обращается к значению здоровья, и установить условную точку останова, которая будет проверять здоровье в каждом таком месте. Затем, когда одна из этих точек останова срабатывает, поочередно проходить (single-step) через код до следующего вызова функции.

Использовать функцию трассировки (trace function) в OllyDbg, чтобы создать динамическую точку останова, которая остановится точно в нужном месте.

Первый метод требовал большей настройки и был неудобен для повторного использования, в основном из-за огромного количества

необходимых точек останова и того, что мне пришлось бы вручную пошагово выполнять код (single-step by hand).

В отличие от него, второй метод был прост в настройке, и, поскольку он выполнялся автоматически, его можно было легко повторить.

## Написание выражения для проверки здоровья (Writing an Expression to Check Health)

Как и прежде, я начал с Cheat Engine, чтобы найти адрес, в котором хранилось мое здоровье. Используя метод, описанный в "Cheat Engine's Memory Scanner" на странице 5, я определил, что адрес здоровья — 0x40A000.

Далее мне нужно было написать выражение, которое заставит OllyDbg возвращать 1, когда здоровье ниже максимального, и 0 в остальных случаях.

Зная, что мое здоровье хранится по адресу 0x40A000 и что максимальное значение — 500, я изначально составил это выражение:

---

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

Это выражение срабатывало немедленно, как только здоровье падало ниже 500 (напоминаю, что десятичные числа должны оканчиваться на точку в движке выражений).

Однако мне нужно было, чтобы прерывание происходило только тогда, когда вызывается функция. Поэтому я добавил второе выражение с оператором `&&`, чтобы убедиться, что остановка произойдет только при вызове функции:

---

```
[0x40A000] < 500. && [1BYTE EIP] == 0xE8
```

На x86-процессорах регистр EIP хранит адрес выполняемой операции, поэтому я решил проверить первый байт по адресу EIP ❶, чтобы убедиться, что он равен 0xE8.

Это значение сообщает процессору выполнить near function call, а именно этот тип вызова мне и был нужен.

Перед тем как запустить трассировку (trace), мне нужно было сделать еще одну последнюю вещь. Так как функция трассировки выполняет код пошагово (Trace into использует step into, а Trace over использует step over), как описано в разделе "A Brief Look at OllyDbg's User Interface" (Краткий обзор интерфейса OllyDbg) на странице 24,

мне нужно было запустить трассировку в месте, находящемся на уровне или выше любого кода, который мог обновить значение здоровья.

## Определение места для начала трассировки (Figuring Out Where to Start the Trace)

Чтобы найти подходящее место, я открыл основной модуль игры в окне CPU OllyDbg, кликнул правой кнопкой мыши в области дизассемблера (disassembler pane) и выбрал Search for  All intermodular calls.

Открылось окно References, в котором отобразился список внешних API-функций, вызываемых игрой.

Почти все игровые программы запрашивают новые сообщения через функцию Windows USER32.PeekMessage(), поэтому я отсортировал список по колонке Destination и ввёл PEEK (можно искать в списке, просто набирая имя, если окно в фокусе), чтобы найти первый вызов USER32.PeekMessage().

Благодаря сортировке по месту назначения (Destination sorting) все вызовы этой функции были сгруппированы в смежный блок следом за первым вызовом, как показано на Рисунке 2-5 (Figure 2-5).

Я установил точку останова на каждом из них, выбрав их и нажав F2.

Address	Disassembly	Destination
00695E60	CALL EDI	GD132.PatBlt
006D0340	CALL DWORD PTR DS:[&GD132.PatBlt]	GD132.PatBlt
006D0289	CALL DWORD PTR DS:[&GD132.PatBlt]	GD132.PatBlt
005C6129	CALL DWORD PTR DS:[&SHLWAPI.PathFindExtensionA]	SHLWAPI.PathFindExtensionA
005D0805	CALL DWORD PTR DS:[&SHLWAPI.PathFindExtensionA]	SHLWAPI.PathFindExtensionA
005D0845	CALL DWORD PTR DS:[&SHLWAPI.PathFindExtensionA]	SHLWAPI.PathFindExtensionA
005D085F	CALL DWORD PTR DS:[&SHLWAPI.PathFindExtensionA]	SHLWAPI.PathFindExtensionA
005D087E	CALL DWORD PTR DS:[&SHLWAPI.PathFindExtensionA]	SHLWAPI.PathFindExtensionA
005D089C	CALL DWORD PTR DS:[&SHLWAPI.PathFindExtensionA]	SHLWAPI.PathFindExtensionA
005D09CC	CALL DWORD PTR DS:[&SHLWAPI.PathFindExtensionA]	SHLWAPI.PathFindExtensionA
005D09E7	CALL DWORD PTR DS:[&SHLWAPI.PathFindExtensionA]	SHLWAPI.PathFindExtensionA
005D0A03	CALL DWORD PTR DS:[&SHLWAPI.PathFindExtensionA]	SHLWAPI.PathFindExtensionA
005D0A11	CALL DWORD PTR DS:[&SHLWAPI.PathFindExtensionA]	SHLWAPI.PathFindExtensionA
005D0A22	CALL DWORD PTR DS:[&SHLWAPI.PathFindExtensionA]	SHLWAPI.PathFindExtensionA
005D0A53	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
005C1067	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
005C1CEB	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
005C1D00	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
005C4268	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
005C45E8	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
00632FD7	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
00673209	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
00676869	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
00676888	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
006AF871	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
006AF930	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
006AF93D	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
006AF94C	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
006AF946	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
006AFCD1	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
006B58C0	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
006B58E7	CALL DWORD PTR DS:[&USER32.PeekMessageA]	USER32.PeekMessageA
005F8AA4	CALL DWORD PTR DS:[&GD132.Polygon]	GD132.Polygon
005F953D	CALL DWORD PTR DS:[&GD132.Polygon]	GD132.Polygon
005F958D	CALL DWORD PTR DS:[&GD132.Polygon]	GD132.Polygon
006SEBE7	CALL DWORD PTR DS:[&GD132.Polygon]	GD132.Polygon
006SEBE8	CALL DWORD PTR DS:[&GD132.Polygon]	GD132.Polygon
006BD569	CALL DWORD PTR DS:[&GD132.Polygon]	GD132.Polygon
006BD56A	CALL DWORD PTR DS:[&GD132.Polygon]	GD132.Polygon
MAINFNIF4	CALL PTR DS:[&MAINFNIF4]	MAINFNIF4

Рисунок 2-5: Окно найденных межмодульных вызовов OllyDb

Хотя вызовов USER32.PeekMessage() было около десятка, только два из них вызывали мои точки останова.

Еще лучше, активные вызовы находились рядом друг с другом в безусловном цикле.

Внизу этого цикла находилось несколько внутренних вызовов функций.

Это выглядело точно как основной игровой цикл (main game loop).

## Активация трассировки (Activating the Trace)

Чтобы наконец запустить трассировку, я удалил все предыдущие точки останова и установил одну точку останова в начале предполагаемого главного цикла. Как только точка останова сработала, я удалил ее. Затем я нажал CTRL-T в окне CPU (CPU window), что открыло диалоговое окно "Condition to pause run trace", показанное на Рисунке 2-6 (Figure 2-6). В этом новом окне я включил опцию "Condition is TRUE", вставил свое выражение в поле рядом с ней и нажал OK. После этого я вернулся в окно CPU (CPU window) и нажал CTRL-F11, чтобы начать сеанс "Trace Into".

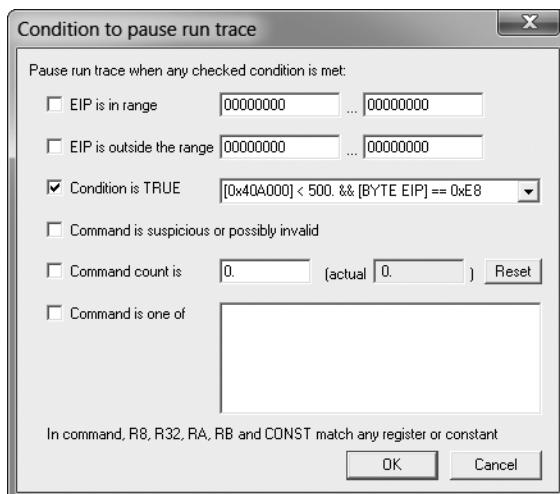


Рисунок 2-6: Условие для приостановки диалога трассировки выполнения

Как только трассировка (trace) началась, игра стала работать так медленно, что была практически неиграбельной. Чтобы уменьшить здоровье моего тестового персонажа, я открыл вторую копию игры, вошел в систему под другим персонажем и атаковал моего тестового персонажа. Когда выполнение трассировки доднalo реальное время, OllyDbg обнаружил изменение моего здоровья и вызвал точку останова на следующем вызове функции — точно как ожидалось. В этой игре основные участки кода, изменяющие здоровье, были вызваны напрямую из сетевого кода.

Используя эту трассировку (trace), я смог найти функцию, которую сетевой модуль вызывал сразу после того, как сетевой пакет указывал игре изменить здоровье игрока.

Вот псевдокод (pseudocode) того, что делала игра:

---

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

---

Я знал, что в игре существовал код, который выполнялся только при изменении здоровья игрока, и мне нужно было добавить код, который также мог реагировать на такие изменения. Не зная общей структуры кода, я предположил, что код, зависящий от здоровья, выполнялся из какой-то функции, которая вызывалась сразу после обновления значения здоровья.

Моя условная точка останова в трассировке (trace conditional breakpoint) подтвердила это предположение, так как она сработала прямо на функции `observe()` ①. Оттуда я смог разместить перехват (hook) на функцию (hooking — метод перехвата вызовов функций, описан в разделе "Hooking to Redirect Game Execution" на странице 153), что позволило выполнять мой собственный код при изменении здоровья игрока.

## Плагины OllyDbg для игровых хакеров (OllyDbg Plug-ins for Game Hackers)

Гибкая система плагинов OllyDbg — это, возможно, одна из его самых мощных функций.

Опытные игровые хакеры часто настраивают свои среды OllyDbg с десятками полезных плагинов, как доступных публично, так и созданных на заказ.

Вы можете скачать популярные плагины с OpenRCE ([http://www.openrce.org/downloads/browse/OllyDbg\\_Plugins](http://www.openrce.org/downloads/browse/OllyDbg_Plugins)) и tut4you (<http://www.tuts4you.com/download.php?list.9/>).

Установка очень проста: просто распакуйте файлы плагина и поместите их в папку установки OllyDbg.

После установки некоторые плагины можно найти в меню OllyDbg's Plugin. Другие могут находиться только в определенных местах интерфейса OllyDbg.

Вы можете найти сотни мощных плагинов в этих онлайн-репозиториях, но нужно быть осторожным при выборе своего набора инструментов.

Перегруженная ненужными плагинами среда может серьезно снизить производительность.

В этом разделе я внимательно отобрал четыре плагина, которые не только являются неотъемлемой частью инструментария игрового хакера, но и минимально вторгаются в среду.

## Копирование ассемблерного кода с помощью Asm2Clipboard

Asm2Clipboard — это минималистичный плагин из репозитория OpenRCE, который позволяет копировать фрагменты ассемблерного кода из окна дизассемблера в буфер обмена. Это может быть полезно для обновления смещений адресов и создания кодовых пещер (code caves), двух ключевых техник взлома игр, которые я подробно рассматриваю в главах 5 и 7. С установленным Asm2Clipboard вы можете выделить блок ассемблерного кода в дизассемблере, кликнуть правой кнопкой мыши по выделенному коду, развернуть подменю Asm2Clipboard и выбрать:

Copy fixed Asm code to clipboard

Copy Asm code to clipboard

Второй вариант добавляет адрес каждой инструкции в качестве комментария, а первый копирует только чистый код.

## Добавление Cheat Engine в OllyDbg с помощью Cheat Utility (Adding Cheat Engine to OllyDbg with Cheat Utility)

Плагин Cheat Utility из tuts4you представляет собой значительно урезанную версию Cheat Engine, встроенную в OllyDbg. Хотя Cheat Utility позволяет выполнять поиск только по точным значениям с очень ограниченным количеством типов данных, он упрощает выполнение простых сканирований, когда вам не нужна полная функциональность Cheat Engine. После установки Cheat Utility для открытия его интерфейса (показано на Рисунке 2-7 (Figure 2-7)) выберите:

Plugins  Cheat utility  Start.

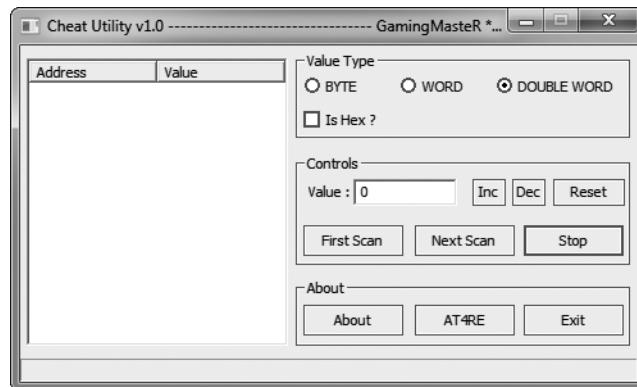


Рисунок 2-7: Интерфейс утилиты Cheat Utility

Интерфейс и работа Cheat Utility практически полностью повторяют Cheat Engine, поэтому ознакомьтесь с Главой 1 (Chapter 1), если вам нужно освежить знания.

**НОТ**

*Games Invader*, обновленная версия Cheat Utility, также от tuts4you, была создана для предоставления большей функциональности. Однако я нашел ее нестабильной, поэтому предпочитаю Cheat Utility, так как всегда могу использовать Cheat Engine для расширенного сканирования.

## Управление OllyDbg через командную строку (Controlling OllyDbg Through the Command Line)

Плагин командной строки (command line plug-in) позволяет управлять OllyDbg через небольшое окно командной строки.

Чтобы открыть этот плагин, нажмите ALT-F1 или выберите Plugins ▶ Command line ▶ Command line.

После этого должно появиться окно, показанное на Рисунке 2-8 (Figure 2-8), которое служит интерфейсом командной строки.

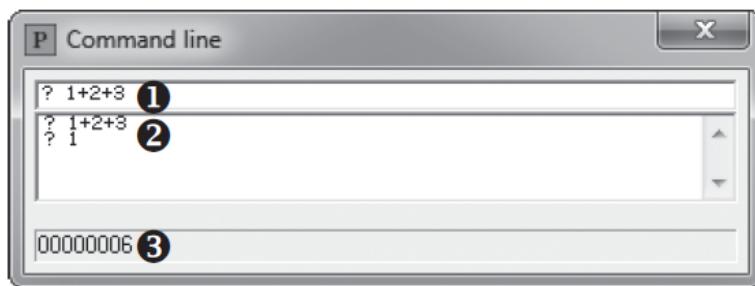


Рисунок 2-8: Интерфейс командной строки

Чтобы выполнить команду, введите ее в поле ввода ① и нажмите ENTER.

Вы увидите историю команд сессии в центральном списке ②, а нижняя метка отображает возвращаемое значение команды ③ (если есть).

Хотя существует множество доступных команд, я считаю, что большинство из них бесполезны.

Я использую этот инструмент в основном для тестирования парсинга выражений в качестве удобного калькулятора, но есть несколько дополнительных вариантов использования, которые также стоит упомянуть.

Я описал их в Таблице 2-6 (Table 2-6).

**Таблица 2-6:** Команды командной строки плагина

Команда	Функция
BC identifier	Удаляет все точки останова, присутствующие на идентификаторе, который может быть адресом кода или именем API-функции.
BP identifier [,condition]	Устанавливает точку останова отладчика на идентификаторе, который может быть адресом кода или именем API-функции. Если <b>identifier</b> — это имя API-функции, точка останова будет установлена на точке входа в функцию. <b>Condition</b> является необязательным выражением, которое, если присутствует, будет установлено в качестве условия останова.
BPX label	Устанавливает точку останова отладчика на каждом экземпляре <b>label</b> внутри текущего дизассемблируемого модуля. Этот <b>label</b> обычно является именем API-функции.
CALC expression	Вычисляет выражение и отображает результат.
? expression	Вычисляет выражение и отображает результат.
HD address	Удаляет все аппаратные точки останова, присутствующие на <b>address</b> .
HE address	Устанавливает аппаратную точку останова на исполнение (hardware on-execute breakpoint) на <b>address</b> .
HR address	Устанавливает аппаратную точку останова на доступ (hardware on-access breakpoint) на <b>address</b> . Одновременно могут существовать не более четырех аппаратных точек останова.
HW address	Устанавливает аппаратную точку останова на запись (hardware on-write breakpoint) на <b>address</b> .
MD	Удаляет любую существующую точку останова в памяти (memory breakpoint), если таковая имеется.
MR address1, address2	Устанавливает точку останова на доступ к памяти (memory on-access breakpoint), начиная с <b>address1</b> и до <b>address2</b> . Заменяет любую существующую точку останова в памяти.
MW address1, address2	Устанавливает точку останова на запись в память (memory on-write breakpoint), начиная с <b>address1</b> и до <b>address2</b> . Заменяет любую существующую точку останова в памяти.
WATCH expression	Открывает окно Watches (Watches window) и добавляет <b>expression</b> в список наблюдения. Выражения в этом списке будут переоцениваться каждый раз, когда процесс получает сообщение, и результаты вычислений будут отображаться рядом с ними.
W expression	Открывает окно Watches (Watches window) и добавляет <b>expression</b> в список наблюдения. Выражения в этом списке будут переоцениваться каждый раз, когда процесс получает сообщение, и результаты вычислений будут отображаться рядом с ними.

Плагин командной строки был разработан автором OllyDbg и должен поставляться предустановленным вместе с OllyDbg.

## Визуализация потока управления с OllyFlow (Visualizing Control Flow with OllyFlow)

OllyFlow, который можно найти в каталоге плагинов OpenRCE, — это чисто визуальный плагин, который может генерировать графы кода, как показано на Рисунке 2-9 (Figure 2-9), и отображать их с помощью Wingraph32.

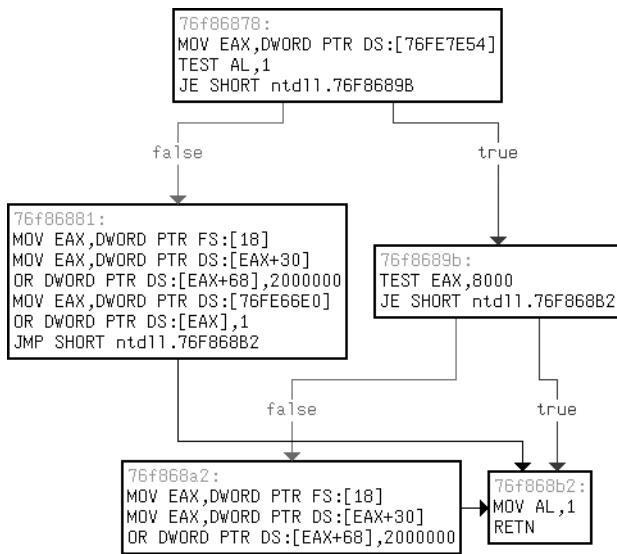


Рисунок 2-9: Блок-схема функции OllyFlow

**NOTE**

Wingraph32 не входит в состав OllyFlow, но доступен в бесплатной версии IDA по ссылке: <https://www.hex-rays.com/products/ida/>. Скачайте .exe и поместите его в папку установки OllyDbg.

Хотя эти графы не являются интерактивными, они помогают легко идентифицировать конструкции, такие как циклы (loops) и вложенные операторы if() в игровом коде, что чрезвычайно важно при анализе потока управления (control flow analysis).

С установленным OllyFlow можно генерировать граф через Plugins ▶ OllyFlow (или кликнув правой кнопкой мыши в дизассемблере, развернув подменю OllyFlow graph) и выбрав один из следующих вариантов:

Generate function flowchart — Генерирует график текущей функции, разбивая различные блоки кода и показывая пути переходов (jump paths). Рисунок 2-9 (Figure 2-9) показывает блок-схему функции. Без сомнения, это самая полезная функция OllyFlow.

Generate xrefs from graph — Генерирует график всех функций, которые вызываются текущей функцией.

Generate xrefs to graph — Генерирует график всех функций, которые вызывают текущую функцию.

Generate call stack graph — Генерирует график предполагаемого пути вызовов, начиная с точки входа в процесс и до текущей функции.

Generate module graph — Теоретически генерирует полный график всех вызовов функций во всем модуле, но на практике редко

работает.

Чтобы понять, насколько полезен OllyFlow, посмотрите на график на Рисунке 2-9 (Figure 2-9) и сравните его с относительно простым ассемблерным кодом, который его генерировал.

---

```
76f86878:  
①    MOV EAX,DWORD PTR DS:[76FE7E54]  
      TEST AL,1  
      JE ntdll.76F8689B  
76f86881:  
②    MOV EAX,DWORD PTR FS:[18]  
      MOV EAX,DWORD PTR DS:[EAX+30]  
      OR DWORD PTR DS:[EAX+68],20000000  
      MOV EAX,DWORD PTR DS:[76FE66E0]  
      OR DWORD PTR DS:[EAX],1  
      JMP ntdll.76F868B2  
76f8689b:  
③    TEST EAX,8000  
      JE ntdll.76F868B2  
76f868a2:  
④    MOV EAX,DWORD PTR FS:[18]  
      MOV EAX,DWORD PTR DS:[EAX+30]  
      OR DWORD PTR DS:[EAX+68],20000000  
76f868b2:  
⑤    MOV AL,1  
      RETN
```

---

Функция начинается с ① и переходит к ②, если ветвление проваливается, или прыгает к ③, если успешно выполняется.

После выполнения ② она переходит непосредственно к ⑤, которая возвращает управление из функции.

После выполнения ③ она либо переходит к ④, либо разветвляется на ⑤ для немедленного возврата.

После выполнения ④ она безусловно переходит к ⑤.

Что именно делает эта функция, не важно для понимания работы OllyFlow; сейчас главное — увидеть, как код соотносится с графиком.

### **Изменение оператора If( )**

Если вы думаете, что готовы погрузиться в работу с OllyDbg, продолжайте чтение.

Перейдите на <https://www.nostarch.com/gamehacking/>, скачайте ресурсные файлы книги, возьмите BasicDebugging.exe и запустите его.

На первый взгляд вы увидите классическую игру Pong. В этой версии Pong мяч становится невидимым, когда он оказывается на экране противника.

Ваша задача — отключить эту функцию, чтобы мяч всегда оставался видимым.

Чтобы упростить задачу, я сделал игру автономной. ИграТЬ не нужно — только взламывать.

Чтобы начать, подключите OllyDbg к игре. Затем сфокусируйте окно CPU на главном модуле (найдите .exe в списке модулей и дважды щелкните по нему) и используйте Referenced text strings, чтобы найти строку, отображаемую в момент скрытия мяча.

Далее дважды щелкните строку, чтобы открыть ее в коде, и проанализируйте окружающий код, пока не найдете if(), который определяет, нужно ли скрыть мяч.

Наконец, используя функцию патчинга кода (code-patching feature), измените оператор if(), чтобы мяч всегда отрисовывался.

Как дополнительный бонус, вы можете попробовать использовать OllyFlow, чтобы создать график этой функции и лучше понять, что именно она делает.

(Подсказка: оператор if() проверяет, меньше ли x-координата мяча 0x140. Если да, он переходит к коду, который рисует мяч. Если нет, он отрисовывает сцену без мяча. Если заменить 0x140 на, скажем, 0xFFFF, мяч никогда не будет скрытым.)

## **Заключительные мысли (Closing Thoughts)**

OllyDbg — гораздо более сложный зверь, чем Cheat Engine, но лучший способ его изучить — просто использовать. Так что смело погружайтесь в процесс и пачкайте руки!

Вы можете начать с комбинирования изученных в этой главе элементов управления с навыками отладки, а затем попробовать их на реальных играх.

Если вы еще не готовы вмешиваться в свою виртуальную судьбу, попробуйте пример из "Patching an if() Statement" в тренировочной среде.

Когда закончите, перейдите к Главе 3 (Chapter 3), где я представлю вам Process Monitor и Process Explorer — два инструмента, которые окажутся бесценными в разведке для взлома игр.

### **3) Разведка с использованием Process Monitor и Process Explorer**



Cheat Engine и OllyDbg могут помочь вам разобрать память и код игры, но вам также нужно понять, как игра

взаимодействует с файлами, значениями реестра, сетевыми подключениями и другими процессами. Чтобы узнать, как происходит это взаимодействие, необходимо использовать два инструмента, которые отлично справляются с этой задачей

мониторинг внешних действий процессов: Process Monitor и Process Explorer. С помощью этих инструментов можно найти полную карту игры, обнаружить файлы сохранений, определить ключи реестра, используемые для хранения настроек, и перечислить IP-адреса удаленных игровых серверов.

В этой главе я расскажу вам, как использовать Process Monitor и Process Explorer для регистрации системных событий и их анализа, чтобы понять, как игра была задействована. Эти инструменты используются в основном для первоначальной разведки потрясающая возможность получить четкое и подробное представление о том, как именно игра взаимодействует с вашей системой. Обе программы можно загрузить с сайта Windows Sysinternals (<https://technet.microsoft.com/en-us/sysinternals/>).

#### **Process Monitor**

Вы можете узнать много нового об игре, просто исследуя, как она взаимодействует с реестром, файловой системой и сетью.

Process Monitor — это мощный инструмент системного мониторинга, который регистрирует подобные события в реальном времени и позволяет бесшовно интегрировать данные в сеанс отладки.

Этот инструмент предоставляет огромный объем полезной информации о взаимодействии игры с внешней средой.

При внимательном анализе (а иногда и спонтанной интуиции) эти данные могут раскрыть сведения о файлах данных, сетевых подключениях и событиях реестра, которые помогут вам увидеть и контролировать поведение игры.

В этом разделе я покажу вам, как использовать Process Monitor для логирования данных, навигации по ним и осмысленного анализа файлов, с которыми взаимодействует игра.

После обзора интерфейса у вас будет возможность опробовать Process Monitor в разделе "Finding a High Score File" на странице 55.

## Логирование внутриигровых событий

Журналы Process Monitor могут содержать всевозможную полезную информацию, но их наиболее практическое применение — помочь вам определить, где хранятся файлы данных, такие как определения внутриигровых предметов. Когда вы запускаете Process Monitor, первым диалоговым окном, которое вы видите, является Process Monitor Filter, показанный на Рисунке 3-1.

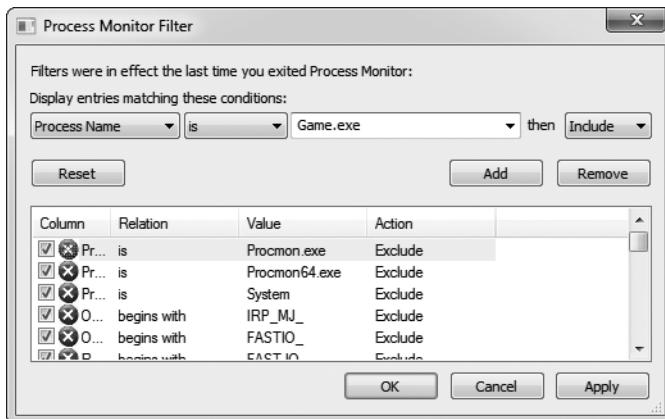


Рисунок 3-1: Диалог фильтра монитора процесса

Это диалоговое окно позволяет вам отображать или скрывать события на основе ряда динамических свойств, которыми они обладают. Чтобы начать мониторинг процессов, выберите Process Name > Is > YourGameFilename.exe > Include, затем нажмите Add.

Apply и OK. Это укажет Process Monitor показывать события, вызванные YourGameFilename.exe. При правильной настройке фильтров вы будете перенаправлены в основное окно, показанное на Рисунке 3-2.

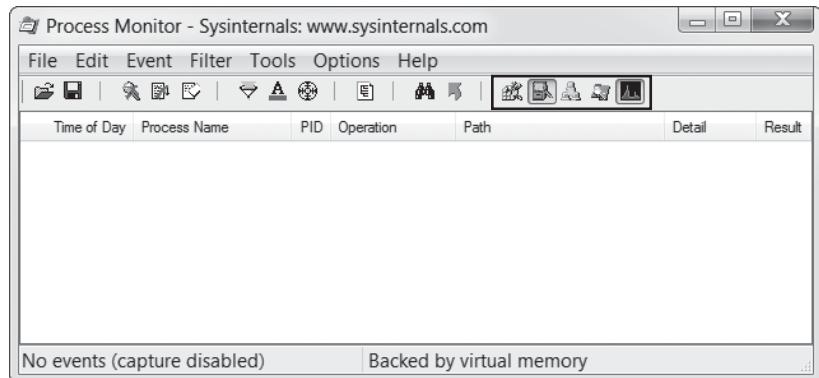


Рисунок 3-2: Главное окно Process Monitor

Чтобы настроить столбцы, отображаемые в области журнала Process Monitor, щелкните правой кнопкой мыши на заголовке и выберите Select Columns. Доступно множество параметров, но я рекомендую выбрать семь.

Time of Day — позволяет увидеть, когда происходят действия.

Process Name — полезно, если вы отслеживаете несколько процессов. Однако при использовании фильтра для одного процесса (обычно применяемого в играх) отключение этой опции может сэкономить место.

Process ID — аналогично Process Name, но отображает ID процесса, а не его имя.

Operation — показывает, какое действие было выполнено; поэтому этот параметр обязательен.

Path — отображает путь к цели выполняемого действия; также обязательен.

Detail — полезен только в некоторых случаях, но его включение не повредит.

Result — показывает, когда действия, например загрузка файлов, завершаются сбоем.

Добавление большого количества столбцов может сделать журнал слишком загроможденным, но использование только этих параметров поможет сохранить лаконичность вывода.

Когда Process Monitor запущен и вы определили нужные столбцы, можно воспользоваться пятью фильтрами классов событий (обведены черным на Рисунке 3-2) для дополнительной очистки логов. Эти фильтры позволяют выбрать, какие события отображать в журнале в зависимости от их типа. Слева направо фильтры расположены следующим образом:

Registry — показывает всю активность реестра. При создании процесса в реестре возникает много "белого шума", так как игры редко используют реестр, тогда как библиотеки Windows всегда с ним работают. Отключение этого фильтра поможет сэкономить место в журнале.

Filesystem — показывает всю активность файловой системы. Это самый важный фильтр класса событий, поскольку знание того, где хранятся файлы данных и как к ним осуществляется доступ, имеет ключевое значение при написании эффективного бота.

Network — показывает всю сетевую активность. Стек вызовов (call stack) сетевых событий может быть полезен для поиска сетевого кода внутри игры.

Process and thread activity — показывает все действия процессов и потоков. Стек вызовов (call stack) этих событий может дать представление о том, как код игры управляет потоками.

Process profiling — периодически показывает информацию об использовании памяти и процессора каждым выполняемым процессом. Взломщики игр редко используют этот фильтр.

Если фильтрация событий на уровне классов оказывается недостаточно точной для устранения нежелательных данных в логах, щелкните правой кнопкой мыши на конкретных событиях, чтобы открыть дополнительные параметры фильтрации на уровне событий. Как только фильтрация будет настроена таким образом, чтобы лог содержал только необходимые вам данные, можно приступать к его анализу. В Таблице 3-1 перечислены полезные сочетания клавиш для управления поведением журнала.

**Таблица 3-1:** Горячие клавиши монитора процессов

Горячая клавиша	Действие
CTRL-E	Включает/отключает логирование.
CTRL-A	Включает/отключает автоматическую прокрутку журнала.
CTRL-X	Очищает журнал.
CTRL-L	Открывает диалоговое окно Filter.
CTRL-H	Открывает диалоговое окно Highlight. Это окно выглядит похоже на Filter, но используется для указания, какие события должны быть выделены.
CTRL-F	Открывает диалоговое окно Search.
CTRL-P	Открывает диалоговое окно Event Properties для выбранного события.

Перемещаясь по журналу, вы можете изучить записанные операции, чтобы увидеть мелкие детали события.

## Анализ событий в журнале Process Monitor

Process Monitor фиксирует все возможные данные о событиях, позволяя узнать о них больше, чем просто о файлах, с которыми они взаимодействуют. Внимательный анализ информативных столбцов, таких как Result и Detail, может дать очень интересные сведения.

Например, я обнаружил, что игры иногда считывают структуры данных поэлементно напрямую из файлов. Это поведение можно заметить, если журнал содержит множество чтений одного и того

же файла, где каждое чтение имеет последовательные смещения, но разную длину. Рассмотрим гипотетический журнал событий, приведённый в Таблице 3-2.

**Таблица 3-2:** Пример журнала событий

Операция	Путь	Детали
Create File	C:\file.dat	Запрошенный доступ: Чтение
Read File	C:\file.dat	Смещение: 0, Размер: 4
Read File	C:\file.dat	Смещение: 4, Размер: 2
Read File	C:\file.dat	Смещение: 6, Размер: 2
Read File	C:\file.dat	Смещение: 8, Размер: 4
Read File	C:\file.dat	Смещение: 12, Размер: 4
...	...	...Продолжает считывать блоки по 4 байта в течение некоторого времени...

Этот журнал показывает, что игра считывает структуру из файла поштучно, раскрывая некоторые намёки на то, как эта структура устроена. Например, предположим, что эти операции чтения соответствуют следующей структуре данных:

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



Сравним журнал в Таблице 3-2 с этой структурой. Сначала игра считывает 4 байта заголовка (header). Затем она считывает два 2-байтовых значения: effectCount и itemCount. После этого создаются два массива целых чисел: effects и items, размеры которых соответствуют значениям effectCount и itemCount. Затем игра заполняет эти массивы данными из файла, считывая по 4 байта (effectCount + itemCount) раз.

**НОТЕ** Разработчики определенно не должны использовать подобный процесс для чтения данных из файла, но вы будете удивлены тем, как часто это происходит. К счастью для вас, подобная наивность лишь облегчает анализ.

В данном случае журнал событий может выявить небольшие фрагменты информации внутри файла. Однако стоит помнить, что если коррелировать прочитанные данные с известной структурой легко, то обратное проектирование неизвестной структуры исключительно по журналу событий — задача гораздо

сложнее.

Обычно хакеры игр используют отладчик (debugger), чтобы получить больше контекста о каждом интересном событии, а данные из Process Monitor могут быть бесшовно интегрированы в сеанс отладки, эффективно связывая две мощные парадигмы реверс-инжиниринга.

## Отладка игры для сбора дополнительных данных

Давайте отойдём от гипотетического чтения файлов и рассмотрим, как Process Monitor позволяет вам перейти от логирования событий к отладке. Process Monitor сохраняет полный стек вызовов (stack trace) для каждого события, показывая цепочку выполнения, которая привела к его срабатыванию.

Вы можете просмотреть эти стековые трассы (stack traces) во вкладке Stack окна Event Properties (двойной щелчок по событию или CTRL-P), как показано на Рисунке 3-3.

The screenshot shows the 'Event Properties' dialog box with the 'Stack' tab selected. The table displays a call stack with 21 frames. The columns are labeled: Frame (1), Module (2), Location (3), Address (4), and Path (5). The stack starts with K\_0 in ntoskrnl.exe at CcMdWriteAbort + 0x3103 and ends with U\_21 in chrome.dll at RelaunchChromeBrowserWithNewComma + 0x15. The 'Path' column shows the full file path for each module.

Frame	Module	Location	Address	Path
K_0	ntoskrnl.exe	CcMdWriteAbort + 0x3103	0xffff800039cf453	C:\Windows\system32\
K_1	tcpip.sys	tcpip.sys + 0xb678	0xffff800018bc678	C:\Windows\System32\
K_2	tcpip.sys	tcpip.sys + 0x3c14e	0xffff8000183d14e	C:\Windows\System32\
K_3	tcpip.sys	tcpip.sys + 0x3c695	0xffff8000183d695	C:\Windows\System32\
K_4	ntoskrnl.exe	KeExpandKernelStackAndCalloutEx + 0xd8	0xffff800038db878	C:\Windows\system32\
K_5	tcpip.sys	tcpip.sys + 0x3c728	0xffff8000183d728	C:\Windows\System32\
K_6	afd.sys	afd.sys + 0x48e9e	0xffff80003ee3e9e	C:\Windows\system32\
K_7	afd.sys	afd.sys + 0x48ad3	0xffff80003ee3ad3	C:\Windows\system32\
K_8	afd.sys	afd.sys + 0x2c04c	0xffff80003ec704c	C:\Windows\system32\
K_9	ntoskrnl.exe	NtMapViewOfSection + 0x1313	0xffff80003bec113	C:\Windows\system32\
K_10	ntoskrnl.exe	NtDeviceIoControlFile + 0x56	0xffff80003becc06	C:\Windows\system32\
K_11	ntoskrnl.exe	KeSynchronizeExecution + 0xa23	0xffff800038cee53	C:\Windows\system32\
U_12	wow64cpu.dll	TurboDispatchJumpAddressEnd + 0x6c0	0x75492e09	C:\Windows\SYSTEM3
U_13	wow64cpu.dll	TurboDispatchJumpAddressEnd + 0x1fb	0x75492944	C:\Windows\SYSTEM3
U_14	wow64.dll	Wow64SystemServiceEx + 0x1ce	0x7550d132	C:\Windows\SYSTEM3
U_15	wow64.dll	Wow64LdrpInitialize + 0x42b	0x7550c54b	C:\Windows\SYSTEM3
U_16	ntdll.dll	RtlIsDosDeviceName_U + 0x23a27	0x77aad447	C:\Windows\SYSTEM3
U_17	ntdll.dll	LdrInitializeThunk + 0xe	0x77a5c34e	C:\Windows\SYSTEM3
U_18	ntdll.dll	NtDeviceIoControlFile + 0x15	0x77c2f921	C:\Windows\SysWOW64
U_19	mswsock.dll	mswsock.dll + 0x2930	0x733e2930	C:\Windows\SysWOW64
U_20	WS2_32.dll	WSASendTo + 0x81	0x75f8b38d	C:\Windows\syswow64
U_21	chrome.dll	RelaunchChromeBrowserWithNewComma + 0x15	0x6086457e	C:\Program Files\Google\Chrome\

Рисунок 3-3: Стек вызовов событий монитора процессов

Стек вызовов отображается в таблице, начиная со столбца Frame (1), который показывает режим выполнения и индекс фрейма стека.

Розовая "K" в этом столбце означает, что вызов произошёл в режиме ядра (kernel mode).

Синяя "U" означает, что вызов произошёл в пользовательском режиме (user mode).

Поскольку хакеры игр обычно работают в пользовательском режиме, операции в режиме ядра обычно не представляют интереса.

Столбец Module (❷) показывает исполняемый модуль, в котором находился вызывающий код. Каждый модуль — это просто имя бинарного файла, который совершил вызов. Это помогает определить, какие вызовы были сделаны внутри игрового бинарного файла.

Столбец Location (❸) показывает имя функции, совершившей вызов, а также смещение вызова (call offset).

Имена функций извлекаются из таблицы экспорта модуля и обычно отсутствуют для функций внутри игровых бинарников.

Если имена функций отсутствуют, вместо них отображается имя модуля и смещение вызова (offset) — сколько байтов после базового адреса модуля находится вызов в памяти.

**Н О Т Е** В контексте кода смещение (offset) означает, сколько байтов ассемблерного кода находится между элементом и его исходным положением.

Столбец Address (❹) показывает адрес кода вызова, что очень полезно, так как можно перейти к этому адресу в дизассемблере OllyDbg.

Столбец Path (❺) показывает путь к модулю, который совершил вызов.

На мой взгляд, стек вызовов — это, безусловно, самая мощная функция в Process Monitor. Он показывает весь контекст, который привёл к событию, что чрезвычайно полезно при отладке игры.

С его помощью можно:

Найти точный код, который спровоцировал событие.

Проследить всю цепочку вызовов (call chain), чтобы понять, как код оказался в этом состоянии.

Определить, какие библиотеки использовались для выполнения каждого действия.

Process Explorer, связанное приложение Process Monitor, не имеет многих дополнительных возможностей по сравнению с Process Monitor или OllyDbg. Однако оно гораздо эффективнее раскрывает некоторые функции, что делает его идеальным выбором в определённых ситуациях.

### **Поиск файла С рекордами**

Если вы готовы протестировать свои навыки работы с Process Monitor, то вы пришли по адресу. Откройте каталог GameHackingExamples/Chapter3\_FindingFiles.

Запустите FindingFiles.exe.

Вы увидите, что это игра Pong, похожая на ту, что описана в разделе "Patching an if() Statement" на странице 46.

В отличие от Главы 2, теперь игра действительно работает способна.

Она также отображает ваш текущий счёт и рекордный счёт за всё время.

Теперь перезапустите игру, запустив Process Monitor перед её выполнением во второй раз.

Отфильтруйте события, связанные с активностью файловой системы (filesystem activity).

Создайте любые другие фильтры, которые сочтёте нужными.

Попробуйте определить, где игра хранит файл с рекордным счётом.

Дополнительное задание: Попробуйте изменить этот файл, чтобы заставить игру отображать максимально возможный счёт.

## **Process Explorer**

Process Explorer — это расширенный диспетчер задач (в нём даже есть кнопка, с помощью которой можно сделать его диспетчером задач по умолчанию). Он очень удобен, когда вы начинаете разбираться в том, как работает игра.

Он предоставляет подробные данные о запущенных процессах, включая:

Родительские и дочерние процессы

Использование процессора (CPU usage)

Использование памяти (memory usage)

Загруженные модули (loaded modules)

Открытые дескрипторы (open handles)

Аргументы командной строки (command line arguments)

Кроме того, Process Explorer может управлять этими процессами.

Он превосходно справляется с отображением высокоуровневой информации, такой как:

Деревья процессов (process trees)

Потребление памяти (memory consumption)

Доступ к файлам (file access)

Идентификаторы процессов (process IDs)

Все эти данные могут быть очень полезны.

Конечно, ни одна из этих данных не имеет особой ценности сама по себе. Но если внимательно проанализировать их, можно выявить корреляции и сделать полезные выводы о том, какие глобальные объекты (например, файлы, мьютексы, сегменты разделяемой памяти) доступны игре. Дополнительно, данные из Process Explorer становятся ещё ценнее, если сопоставлять их с данными, собранными во время сеанса отладки (debugging session).

В этом разделе представлено:

Описание интерфейса Process Explorer

Разбор свойств, которые он показывает

Пояснение, как можно использовать его для управления дескрипторами (handles), которые представляют системные ресурсы. После ознакомления с этим разделом используйте "Finding and Closing a Mutex" на странице 60, чтобы улучшить свои навыки.

## Пользовательский интерфейс и элементы управления Process Explorer

Когда вы открываете Process Explorer, вы видите окно, разделенное на три разных раздела, как показано на рисунке 3-4.

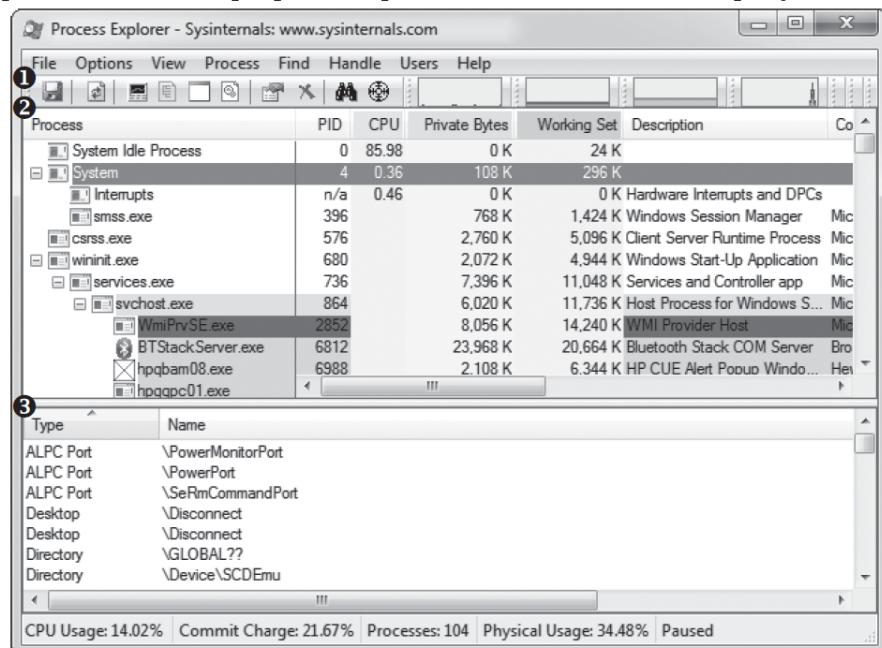


Рисунок 3-4: Главное окно Process Explorer

Эти три секции включают: панель инструментов (1), верхнюю панель (2) и нижнюю панель (3).

Верхняя панель отображает список процессов, используя древовидную структуру для отображения их родительско-дочерних связей. Разные процессы выделяются разными цветами; если вам не нравятся текущие цвета, нажмите Options > Configure Colors, чтобы открыть диалоговое окно, которое позволяет просматривать и изменять их.

Так же, как и в Process Monitor, отображение этой таблицы очень гибкое, и вы можете настроить его, щёлкнув правой кнопкой мыши на заголовке таблицы и выбрав Select Columns. Существует, вероятно, более 100 вариантов настройки, но я считаю, что настройки по умолчанию с добавлением столбца ASLR Enabled работают просто отлично.

**НОТЕ** *Address Space Layout Randomization (ASLR) — это функция безопасности Windows, которая размещает исполняемые образы в непредсказуемых местах. Знание, включена ли она, чрезвычайно важно, если вы пытаетесь изменить игровые значения в памяти.*

Нижняя панель имеет три возможных состояния:

Hidden (Скрыто) — скрывает панель с экрана.

DLLs — отображает список динамических библиотек (Dynamic Link Libraries, DLLs), загруженных в текущий процесс.

Handles — показывает список дескрипторов (handles), удерживаемых процессом (видно на Рисунке 3-4).

Вы можете скрыть или показать всю нижнюю панель, переключив View ▶ Show Lower Pane.

Когда панель открыта, можно изменить отображаемую информацию, выбрав: View ▶ Lower Pane View ▶ DLLs, View ▶ Lower Pane View ▶ Handles

Вы также можете использовать горячие клавиши (hotkeys), чтобы быстро переключаться между режимами нижней панели, не затрагивая процессы в верхней панели. Эти горячие клавиши приведены в Таблице 3-3.

**Таблица 3-3:** Горячие клавиши **Process Explorer**

Горячая клавиша	Действие
CTRL-F	Искать значение в данных нижней панели.
CTRL-L	Переключать нижнюю панель между скрытым и видимым состоянием.
CTRL-D	Переключать нижнюю панель для отображения DLLs.
CTRL-H	Переключать нижнюю панель для отображения дескрипторов (Handles).
Пробел (Spacebar)	Включать/выключать автообновление списка процессов.
ENTER	Открывать диалоговое окно Properties для выбранного процесса.
DEL	Завершать выбранный процесс.
SHIFT-DEL	Завершать выбранный процесс и все его дочерние процессы.

Используйте графический интерфейс (GUI) или горячие клавиши, чтобы практиковаться в смене режимов. Когда вы ознакомитесь с главным окном, мы рассмотрим ещё одно важное диалоговое окно Process Explorer, называемое Properties.

## Изучение свойств процесса

Так же, как Process Monitor, Process Explorer использует динамический подход к сбору данных. Результатом является широкий и подробный спектр информации. Если вы откроете диалоговое окно Properties (показано на Рисунке 3-5) для процесса, вы увидите масштабную панель вкладок, содержащую 10 вкладок.

Вкладка Image, выбранная по умолчанию (показана на Рисунке

3-5), отображает:

Имя исполняемого файла (Executable name)

Версию (Version)

Дату сборки (Build date)

Полный путь (Complete path)

Также отображается:

Текущий рабочий каталог (Current working directory)

Статус Address Space Layout Randomization (ASLR) для исполняемого файла

Статус ASLR — это наиболее важная информация в этом разделе, так как он напрямую влияет на то, как бот может считывать память из игры. Я расскажу об этом подробнее в Главе 6.

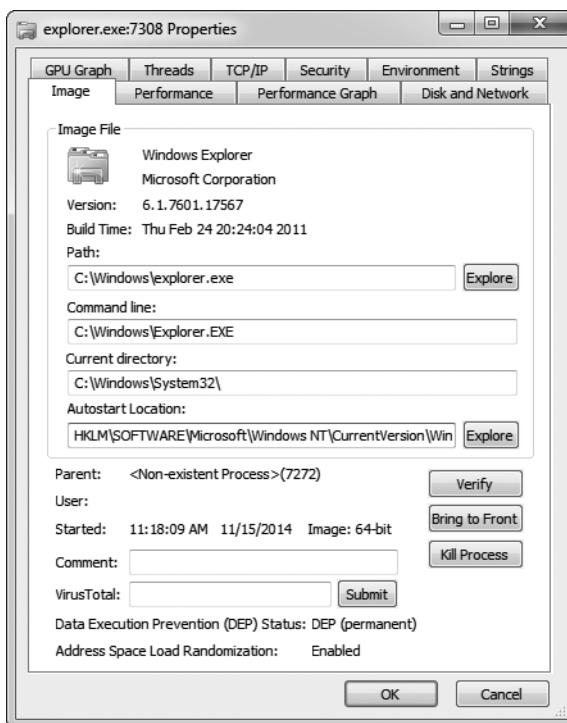


Рисунок 3-5: Диалоговое окно свойств Process Explorer

Вкладки "Производительность", "График производительности", "Диск и сеть" и "График GPU" отображают огромное количество показателей использования процессора, памяти, диска, сети и GPU процессором. Если вы создаете бота, который внедряется в игру, эта информация может быть очень полезной, чтобы определить, насколько сильно ваш бот влияет на игру.

На вкладке TCP/IP отображается список активных TCP-соединений, с помощью которого вы можете найти IP-адреса игровых серверов, к которым подключается игра. Если вы

пытаетесь проверить скорость соединения, прервать подключение или изучить сетевой протокол игры, эта информация очень важна.

На вкладке Strings отображается список строк, найденных в двоичном файле или в памяти процесса. В отличие от списка строк в OllyDbg, который показывает только строки, на которые ссылается ассемблерный код, этот список включает любые вхождения трех или более последовательных читаемых символов, за которыми следует нулевой терминатор. Когда бинарные файлы игры обновляются, вы можете использовать инструмент сравнения этого списка из каждой версии игры, чтобы определить, есть ли в нем новые строки, которые вы хотите исследовать.

Вкладка Threads показывает список потоков, запущенных в процессе, и позволяет приостановить, возобновить или завершить каждый из них; вкладка Security отображает привилегии безопасности процесса; а вкладка Environment отображает все переменные среды, известные процессу или установленные им.

**НО ТЕ** *Если вы откроете диалог свойств процесса .NET, вы заметите две дополнительные вкладки: сборки .NET и производительность .NET. Данные на этих вкладках вполне объяснимы. Имейте в виду, что большинство приемов, описанных в этой книге, не будут работать с играми, написанными на .NET.*

## Параметры управления дескрипторами (Handle Manipulation Options)

Как вы уже видели, Process Explorer может предоставить обширную информацию о процессе. Однако на этом его возможности не заканчиваются: он также позволяет управлять определёнными частями процесса.

Например, можно просматривать и управлять открытыми дескрипторами (handles) непосредственно через нижнюю панель (lower pane) Process Explorer (см. Рисунок 3-4).

Одна только эта функция делает Process Explorer мощным инструментом, достойным того, чтобы включить его в свой арсенал.

Закрытие дескриптора осуществляется просто:

Щёлкните правой кнопкой мыши по нужному дескриптору.  
Выберите Close Handle.

Это может быть очень полезно, например, если нужно закрыть мьютексы (mutexes), что является необходимым условием для определённых типов взлома (hacks).

**НО ТЕ** *Вы можете щёлкнуть правой кнопкой мыши по заголовку нижней панели и выбрать Select Columns, чтобы настроить отображение столбцов. Один из столбцов, который может оказаться особенно полезным, — Handle Value. Он помогает, если вы видите дескриптор, передаваемый в OllyDbg, и хотите понять, что он делает.*

## Закрытие мьютексов (Closing Mutexes)

Игры часто позволяют работать только одному клиенту одновременно; это называется ограничением на одну копию (single-instance limitation). Существует несколько способов реализации ограничения одной копии, но использование системного мьютекса (system mutex) является распространённым решением, так как мьютексы работают в пределах всей сессии (sessionwide) и доступны по простому имени. Использование мьютексов для ограничения количества запущенных копий очень просто, и благодаря Process Explorer убрать это ограничение так же легко. Это позволяет запускать несколько копий игры одновременно.

Сначала давайте разберёмся, как игра может реализовать ограничение одной копии через мьютекс:

```
int main(int argc, char *argv[]) {
    // создать мьютекс
    HANDLE mutex = CreateMutex(NULL, FALSE, "onlyoneplease");

    if (GetLastError() == ERROR_ALREADY_EXISTS) {
        // мьютекс уже существует, выход
        ErrorBox("An instance is already running.");
        return 0;
    }

    // мьютекс не существовал; он только что был создан,
    // поэтому запускаем игру
    RunGame();

    // игра завершена; закрываем мьютекс, чтобы освободить его
    // для будущих экземпляров
    if (mutex)
        CloseHandle(mutex);

    return 0;
}
```

Этот пример кода создаёт мьютекс с именем "onlyoneplease".

Затем функция GetLastError() проверяет, был ли мьютекс уже создан. Если мьютекс существует, программа закрывает игру. Если мьютекс не существует, игра создаёт первый экземпляр, блокируя запуск любых новых клиентов игры. В этом примере игра выполняется normally, и после завершения вызывается CloseHandle(), чтобы освободить мьютекс и позволить запускать будущие экземпляры игры. Вы можете использовать Process Explorer, чтобы закрыть мьютексы, ограничивающие запуск нескольких экземпляров игры, и запускать сразу несколько копий игры одновременно.

Для этого:

Выберите режим отображения Handles в нижней панели.

Найдите все дескрипторы с типом Mutant.

Определите, какой из них ограничивает запуск нескольких копий игры.

Закройте этот мьютекс.

#### NOTE

*Мьютессы также используются для синхронизации данных между потоками и процессами. Закрывайте мьютекс только в том случае, если уверены, что его единственное предназначение — именно то, которое вы пытаетесь обойти!*

Хаки для многоクlientских режимов очень востребованы, поэтому возможность быстро разрабатывать их для новых игркритически важна для успеха разработчика ботов.

Так как мьютессы — один из самых распространённых способов ограничения одной копии (single-instance limitation), Process Explorer является неотъемлемым инструментом для прототипирования таких взломов (hacks).

## Анализ доступа к файлам (Inspecting File Accesses)

В отличие от Process Monitor, Process Explorer не может отображать список вызовов файловой системы (filesystem calls).

Однако в режиме Handles (нижняя панель Process Explorer) можно увидеть все файловые дескрипторы, которые в данный момент открыты игрой.

Это позволяет точно определить, какие файлы находятся в постоянном использовании, без необходимости настраивать сложные фильтры в Process Monitor.

Чтобы увидеть все файлы, которые игра использует в данный момент, найдите дескрипторы с типом "File".

Эта функциональность может быть полезна, если вы:

Пытаетесь найти лог-файлы (logfiles) или файлы сохранений (save files).

Хотите обнаружить именованные каналы (named pipes), используемые для межпроцессорного взаимодействия (IPC).

Файлы именованных каналов имеют префикс: \Device\NamedPipe\

Обнаружение такого канала может указывать на то, что игра взаимодействует с другим процессом.

### **Поиск и закрытие мьютекса**

Чтобы применить **навыки работы с Process Explorer**, выполните следующие шаги:

- Перейдите в каталог: GameHackingExamples/Chapter3\_CloseMutex
- Запустите **CloseMutex.exe**.

Эта игра **ведёт себя так же**, как та, что упоминалась в разделе "**Finding a High Score File**" (стр. 55),

но не позволяет запустить несколько экземпляров одновременно.

Как вы могли догадаться, ограничение реализовано через мьютекс (**single-instance-limitation mutex**).

Чтобы обойти это ограничение:

Откройте **Process Explorer**.

В нижней панели выберите **режим Handles**.

Найдите **мьютекс**, который отвечает за это ограничение.

Закройте его.

Если всё сделано правильно, вы сможете **открыть второй экземпляр игры**.

## **Заключительные мысли (Closing Thoughts)**

Чтобы эффективно использовать Process Monitor и Process Explorer, нужно глубоко разбираться в данных, которые эти приложения отображают, а также в интерфейсах, через которые они их показывают.

Хотя этот обзор главы даёт хорошую базу, тонкости этих приложений можно изучить только через опыт, поэтому я рекомендую вам экспериментировать с ними на своей системе.

Вы не будете использовать эти инструменты регулярно, но в какой-то момент они спасут вас, когда вы будете пытаться разобраться, как работает определённый код.

Вспоминая какой-то неочевидный фрагмент информации, который зацепил ваш взгляд во время предыдущего сеанса работы с Process Explorer или Process Monitor, вы сможете разгадать сложные моменты.

Бот почему я считаю их полезными инструментами разведки (reconnaissance tools).

## **Часть 2: Анализ игры**

## 4) От кода в памяти: общий вводный курс



На самом низком уровне код игры, данные, ввод и вывод — это сложные абстракции беспорядочно изменяющихся байтов.

Многие из этих байтов представляют переменные или машинный код, сгенерированный компилятором на основе исходного кода игры. Некоторые байты представляют изображения, модели и звуки.

Другие существуют лишь мгновение, появляясь как ввод аппаратного обеспечения компьютера и уничтожаясь, когда игра завершает их обработку.

Оставшиеся байты информируют игрока о текущем состоянии игры. Но люди не мыслят байтами, поэтому компьютер должен перевести их в понятный для нас формат. Существует огромный разрыв и в обратном направлении. Компьютер не понимает высокоуровневый код и контент игры, поэтому они должны быть переведены из абстрактных представлений в байты.

Некоторые виды контента — такие как изображения, звуки и текст — сохраняются без потерь (*losslessly*) и готовы к отображению. Игрок получает отклик от системы за микросекунды. Код игры, логика и переменные, с другой стороны, лишены какой-либо читаемости и скомпилированы в машинные данные.

Манипулируя данными игры, хакеры получают невероятные преимущества в игровом процессе. Однако для этого им нужно понимать, как код разработчика преобразуется после компиляции и выполнения. По сути, они должны мыслить, как компьютеры.

Чтобы помочь вам думать, как компьютер, эта глава начнётся с объяснения представления чисел, текста, структур и объединений (*unions*) в памяти на уровне байтов. Затем мы углубимся в хранение экземпляров классов (*class instances*) в памяти и как

абстрактные экземпляры знают, какие виртуальные функции (*virtual functions*) вызывать во время выполнения (*runtime*). В последней части главы вы пройдёте краткий курс по ассемблеру x86, который охватывает: синтаксис (*syntax*), регистры (*registers*), операнды (*operands*), стек вызовов (*call stack*), арифметические операции (*arithmetic operations*), операции ветвления (*branching operations*), вызовы функций (*function calls*) и конвенции вызовов (*calling conventions*).

Эта глава очень сильно сфокусирована на технических деталях. В ней нет информации, непосредственно связанной с взломом игр, но знания, которые вы здесь получите, станут центральными в следующих главах, где мы будем разбирать темы чтения и записи памяти программным способом (*programmatically reading and writing memory*), инъекцию кода (*injecting code*) и манипулирование потоком управления (*manipulating control flow*).

Так как C++ является фактическим стандартом как для разработки игр, так и для разработки ботов, эта глава объясняет взаимосвязь между кодом C++ и памятью, которая его представляет. Большинство нативных языков имеют очень схожую (иногда идентичную) низкоуровневую структуру и поведение, поэтому вы сможете применить полученные здесь знания почти к любому программному обеспечению.

Весь пример кода из этой главы находится в каталоге `GameHackingExamples/Chapter4_CodeToMemory` с исходными файлами книги. Проекты могут быть скомпилированы в Visual Studio 2010, но также должны работать с любым другим компилятором C++. Вы можете загрузить их по ссылке <https://www.nostarch.com/gamehacking/> и скомпилировать, если хотите следовать вместе с материалом.

## Как переменные и другие данные отображаются в памяти (*How Variables and Other Data Manifest in Memory*)

Правильное манипулирование состоянием игры может быть очень сложным, а поиск данных, которые управляют им, не всегда так прост, как нажатие Next Scan и надежда на то, что Cheat Engine не подведёт. Фактически, многие хаки должны манипулировать десятками связанных значений одновременно. Поиск этих значений и их взаимосвязей часто требует аналитического подхода и идентификации структур и шаблонов.

Более того, разработка игровых хаков обычно означает воссоздание оригинальных структур внутри кода бота. Чтобы делать такие вещи, необходимо глубокое понимание того, как переменные и данные располагаются в памяти игры. С помощью примеров кода, дампов памяти в OllyDbg и нескольких таблиц, связывающих всю информацию, этот раздел научит вас всему, что нужно знать о том, как различные типы данных отображаются в памяти.

## Числовые данные (Numeric Data)

Большинство значений, которые нужны хакерам игр (такие как здоровье игрока, мана, местоположение и уровень), представлены числовыми типами данных. Так как числовые типы данных являются основой для всех других типов, их понимание чрезвычайно важно.

К счастью, они имеют относительно простые представления в памяти:

Они предсказуемо выровнены.

Они имеют фиксированную разрядность (bit width).

Таблица 4-1 показывает пять основных числовых типов данных, которые можно встретить в играх для Windows, вместе с их размерами и диапазонами значений.

Таблица 4-1: Числовые типы данных

Имя типа (Type name(s))	Размер (Size)	Диапазон знаковых чисел (Signed range)	Диапазон беззнаковых чисел (Unsigned range)
char, BYTE	8 бит	-128 до 127	0 до 255
short, WORD, wchar_t	16 бит	-32,768 до -32,767	0 до 65,535
int, long, DWORD	32 бита	-2,147,483,648 до 2,147,483,647	0 до 4,294,967,295
long long	64 бита	-9,223,372,036,854,775,808 до 9,223,372,036,854,775,807	0 до 18,446,744,073,709,551,615
float	32 бита	$+/-1.17549 \times 10^{-38}$ до $+/-3.40282 \times 10^{38}$	N/A

Размеры числовых типов данных могут различаться в зависимости от архитектуры и даже компиляторов. Поскольку эта книга сосредоточена на взломе игр x86 на Windows, используются стандартные для Microsoft имена и размеры типов. За исключением float, типы данных в Таблице 4-1 хранятся с малым порядком байтов (little-endian ordering), что означает, что наименее значимые байты целого числа хранятся в младших адресах памяти.

Например, Рисунок 4-1 показывает, что DWORD 0x0A0B0C0D представлен байтами 0x0D 0x0C 0x0B 0x0A.

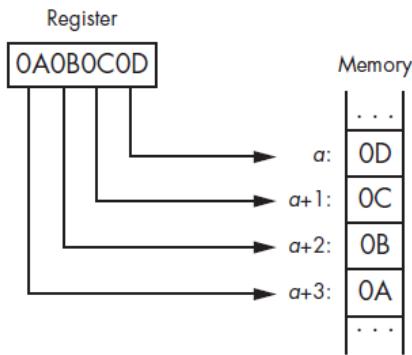


Рисунок 4-1: Диаграмма упорядочивания младших порядков

Тип данных `float` может содержать смешанные числа, поэтому его представление в памяти не так просто, как у других типов данных. Например, если вы видите `0x0D 0x0C 0x0B 0x0A` в памяти, и это значение является `float`, вы не можете просто преобразовать его в `0x0A0B0C0D`.

Вместо этого значения `float` состоят из трёх компонентов: знака (`sign`) (бит 0), экспоненты (`exponent`) (биты 1–8) и мантиссы (`mantissa`) (биты 9–31).

Знак определяет, является ли число отрицательным или положительным, экспонента определяет, на сколько позиций нужно переместить десятичную точку (начиная перед мантиссой), а мантисса хранит приближённое значение числа.

Вы можете восстановить сохранённое значение, вычислив выражение  $\text{mantissa} \times 10^n$  (где  $n$  — это экспонента) и умножив результат на -1, если установлен знак.

Теперь давайте посмотрим на некоторые числовые типы данных в памяти. Листинг 4-1 инициализирует девять переменных.

---

```

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;

```

---

Листинг 4-1: Создание переменных числовых типов данных в C++

Начиная с самого верха, этот пример включает переменные типов `char`, `short`, `int`, `long long` и `float`. Четыре из них являются беззнаковыми (`unsigned`), а пять — знаковыми (`signed`). (В C++ тип `float` не может быть беззнаковым.) Учитывая всё, что вы изучили

до сих пор, внимательно изучите взаимосвязь между кодом в Листинге 4-1 и дампом памяти в Рисунке 4-2. Предположим, что переменные объявлены в глобальной области видимости.

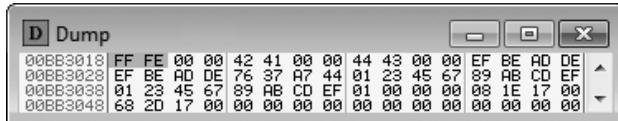


Рисунок 4-2: OllyDbg дамп памяти наших числовых данных

Вы можете заметить, что некоторые значения расположены с произвольными промежутками. Это связано с тем, что процессорам значительно быстрее обращаться к значениям, которые находятся по адресам, кратным размеру адреса (address size) (в x86 это 32 бита).

Компиляторы добавляют нули (padding), чтобы выравнивать (align) значения по таким адресам. Поэтому padding также называют выравниванием (alignment).

Однобайтовые значения не выравниваются, так как операции, которые их обрабатывают, работают одинаково независимо от выравнивания.

Учитывая это, обратите внимание на Таблицу 4-2, которая предоставляет перекрёстную ссылку между дампом памяти в Рисунке 4-2 и переменными, объявленными в Листинге 4-1. **Таблица 4-2:** Переход от памяти к коду для листинга 4-1 и рисунка 4-2

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

Столбец Адрес (Address) показывает расположение данных в памяти, а столбец Данные (Data) указывает, что именно там хранится. Столбец Объект (Object) указывает, какой переменной из Листинга 4-1 соответствует данный фрагмент памяти.

Обратите внимание, что floatValue размещён перед ulonglongValue в памяти, несмотря на то, что он был объявлен

последним в Листинге 4-1.

Так как эти переменные объявлены в глобальной области видимости (global scope), компилятор может размещать их где угодно.

Этот конкретный порядок, скорее всего, является результатом либо выравнивания (alignment), либо оптимизации (optimization).

## Строковые данные (String Data)

Большинство разработчиков используют термин строка (string) так, как будто он является синонимом текста (text), но текст — это лишь самое распространённое применение строк. На низком уровне строки — это просто массивы произвольных числовых объектов, которые выглядят линейными и невыравненными (unaligned) в памяти.

Листинг 4-2 показывает четыре объявления строковых переменных.

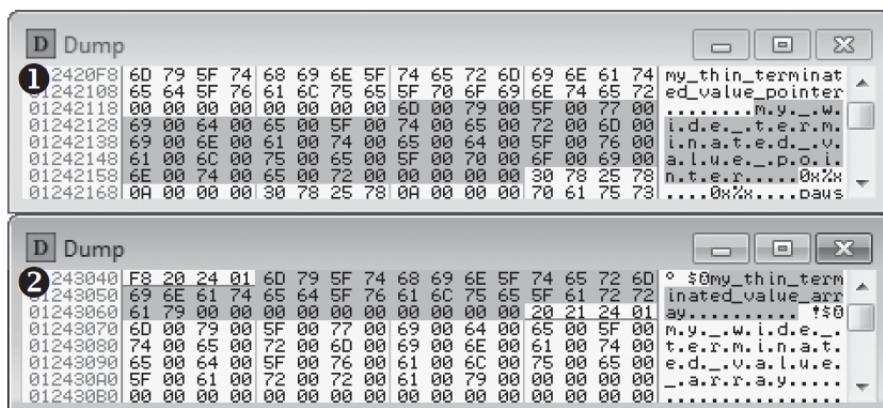
```
// char занимает 1 байт на символ
char* thinStringP = "my_thin_terminated_value_pointer";
char thinStringA[40] = "my_thin_terminated_value_array";

// wchar_t занимает 2 байта на символ
wchar_t* wideStringP = L"my_wide_terminated_value_pointer";
wchar_t wideStringA[40] = L"my_wide_terminated_value_array";
```

Листинг 4-2: Объявление нескольких строк в C++

В контексте текста строки содержат символьные объекты (char для 8-битной кодировки (8-bit encoding) или wchar\_t для 16-битной кодировки (16-bit encoding)), а конец каждой строки указывается нулевым терминатором (null terminator), то есть символом 0x0.

Теперь давайте посмотрим на память, где хранятся эти переменные, как показано в двух дампах памяти в Рисунке 4-3 (Figure 4-3).



*Рисунок 4-3: В этом дампе памяти OllyDbg строковых данных человекочитаемый текст в колонке ASCII - это текст, который мы сохранили в листинге 4-2.*

Если вы не привыкли читать дампы памяти, дамп OllyDbg может показаться сложным для понимания. Таблица 4-3 позволяет глубже рассмотреть связь между кодом в Листинге 4-2 и памятью в Рисунке 4-3.

**Таблица 4-3:** Переход от памяти к коду для листинга **4-2** и рисунка **4-3**

**Pane 1**

Адрес (Address)	Размер (Size)	Данные (Data)	Объект (Object)
0x012420F8	32 байта	0x6D 0x79 0x5F (...) 0x74 0x65 0x72	Символы thinStringP
0x01242118	4 байта	0x00 0x00 0x00 0x00	Терминатор thinStringP и заполнение (padding)
0x0124211C	4 байта	0x00 0x00 0x00 0x00	Несвязанные данные (Unrelated data)
0x01242120	64 байта	0x6D 0x00 0x79 (...) 0x00 0x72 0x00	Символы wideStringP
0x01242160	4 байта	0x00 0x00 0x00 0x00	Терминатор wideStringP и заполнение (padding)
{...}	-	-	Несвязанные данные (Unrelated data)

**Pane 2**

Адрес (Address)	Размер (Size)	Данные (Data)	Объект (Object)
0x01243040	4 байта	0xF8 0x20 0x24 0x01	Указатель на thinStringP (Pointer to thinStringP) в 0x012420F8
0x01243044	30 байт	0x6D 0x79 0x5F (...) 0x72 0x61 0x79	Символы thinStringA
0x01243062	10 байт	0x00 (повторяется 10 раз)	Терминатор thinStringA и заполнение массива
0x0124306C	4 байта	0x20 0x21 0x24 0x01	Указатель на wideStringP (Pointer to wideStringP) в 0x01242120
0x01243070	60 байт	0x6D 0x00 0x79 (...) 0x00 0x79 0x00	Символы wideStringA
0x012430AC	20 байт	0x00 (повторяется 10 раз)	Терминатор wideStringA и заполнение массива

В Рисунке 4-3 панель 1 показывает, что значения, хранящиеся по адресам thinStringP (0x01243040) и wideStringP (0x0124306C), занимают в памяти всего 4 байта и не содержат строковых

данных. Это потому, что эти переменные на самом деле являются указателями (pointers) на первые символы их соответствующих массивов.

Например, `thinStringP` содержит `0x012420F8`, и в панели 2 Рисунка 4-3 можно увидеть "my\_thin\_terminated\_value\_pointer" по адресу `0x012420F8`.

Посмотрите на данные между этими указателями в панели 1, и вы увидите текст, хранящийся в `thinStringA` и `wideStringA`. Кроме того, обратите внимание, что `thinStringA` и `wideStringA` дополнены (padded) за пределами их нулевых терминаторов.

Это потому, что эти переменные были объявлены как массивы длиной 40, поэтому они заполнены до 40 символов.

## Структуры данных (Data Structures)

В отличие от рассмотренных ранее типов данных, структуры (structures) — это контейнеры, которые хранят несколько связанных простых фрагментов данных.

Игровые хакеры, которые знают, как распознавать структуры в памяти, могут имитировать эти структуры в своем коде.

Это может существенно уменьшить количество адресов, которые необходимо найти, так как им нужно найти только адрес начала структуры, а не адрес каждого отдельного элемента.

**Н О Т Е** В этом разделе рассматриваются структуры как простые контейнеры, не содержащие функций-членов (*member functions*) и содержащие только простые данные. Объекты, которые выходят за эти рамки, будут рассмотрены в разделе "Классы и VF-таблицы" на странице 74 (*Classes and VF Tables on page 74*).

## Порядок элементов в структуре и выравнивание (Structure Element Order and Alignment)

Поскольку структуры просто представляют собой набор объектов, они не видны явно в дампах памяти. Вместо этого дамп памяти структуры показывает объекты, содержащиеся в этой структуре. Такой дамп будет выглядеть похоже на другие, показанные в этой главе, но с важными различиями в порядке (order) и выравнивании (alignment). Чтобы увидеть эти различия, посмотрите Листинг 4-3 (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\n",
       &m->ubyteValue, &m->byteValue,
       &m->uwordValue, &m->wordValue,
       &m->udwordValue, &m->dwordValue,
       &m->ulongLongValue, &m->longLongValue,
       &m->floatValue);
```

---

Листинг 4-3: Структура C++ и код, который ее использует

Этот код объявляет структуру с именем MyStruct и создаёт переменную m, которая якобы указывает на экземпляр структуры по адресу 0. На самом деле экземпляра структуры по адресу 0 не существует, но этот трюк позволяет использовать оператор амперсанда (&) в вызове printf() для получения адреса каждого члена структуры. Поскольку структура находится по адресу 0, адрес, напечатанный для каждого члена, эквивалентен его смещению от начала структуры.

Конечная цель этого примера — точно увидеть, как каждый член располагается в памяти относительно начала структуры. Если бы вы запустили код, то увидели бы следующий вывод:

---

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

Как видно, переменные в MyStruct упорядочены точно так же, как они были определены в коде. Такое последовательное размещение членов является обязательным свойством структур. Сравните это с примером из Листинга 4-1, где объявлен идентичный набор переменных: в дампе памяти из Рисунка 4-2 компилятор явно разместил некоторые значения не по порядку.

Кроме того, вы могли заметить, что члены структуры не выровнены так же, как глобальные переменные в Листинге 4-1. Если бы они были выровнены аналогично, перед uwordValue находилось бы 2 байта заполнения (padding bytes). Это связано с тем, что члены структуры выравниваются по адресам, делящимся либо на структурное выравнивание членов (struct member alignment) (опция компилятора, принимающая значения 1, 2, 4, 8 или 16 байт; в данном случае установлено 4), либо на размер самого члена, в зависимости от того, что меньше. Я расположил члены MyStruct так, чтобы компилятору не пришлось добавлять дополнительные байты заполнения.

Однако, если бы мы добавили char сразу после ulongLongValue, вызов printf() дал бы следующий результат:

---

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

Теперь взгляните на оригинальный и изменённый вывод вместе:

---

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

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

---

В изменённой версии последние два значения, которые

являются смещениями longLongValue и floatValue от начала структуры, изменились. Благодаря выравниванию членов структуры (struct member alignment) переменная longLongValue сдвигается на 4 байта (1 байт для char и 3 байта заполнения после него), чтобы гарантировать её размещение по адресу, делящемуся на 4.

## Как работают структуры

Понимание структур — как они выравниваются и как их можно имитировать — может быть очень полезным. Например, если вы воспроизведёте структуры игры в своём коде, вы сможете читать или записывать целые структуры из памяти за одну операцию. Рассмотрим объявление структуры, которая хранит текущее и максимальное здоровье игрока:

---

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

---

Если неопытный взломщик игры хочет прочитать эту информацию из памяти, он может написать что-то вроде этого, чтобы получить значения здоровья:

---

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

---

Этот взломщик игры не осознаёт, что то, что эти значения расположены прямо рядом друг с другом в памяти, может быть не просто случайным совпадением. Поэтому он использует две отдельные переменные.

Но если бы вы использовали свои знания о структурах, вы могли бы прийти к выводу, что поскольку эти значения тесно связаны и находятся рядом в памяти, взломщик мог бы вместо этого использовать структуру:

---

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

---

Поскольку этот код предполагает, что используется структура и правильно её имитирует, он может получить текущее и максимальное здоровье всего за одну строку ❶.

Мы углубимся в то, как писать свой код для чтения памяти в Главе 6.

## Unions (Объединения)

В отличие от структур, которые инкапсулируют несколько связанных частей данных, объединения (unions) содержат только

один элемент данных, который доступен через несколько переменных. Объединения (unions) подчиняются трём правилам:

- Размер объединения (union) в памяти равен размеру его самого большого члена.
- Все члены объединения (union) ссылаются на одну и ту же область памяти.
- Объединение (union) наследует выравнивание своего самого большого члена.

Вызов printf() в следующем коде помогает проиллюстрировать первые два правила:

---

```
union {
    BYTE byteValue;
    struct {
        WORD first;
        WORD second;
    } words;
    DWORD value;
} dwValue;
dwValue.value = 0xDEADBEEF;
printf("Size %d\nAddresses 0x%08x,0x%08x\nValues 0x%08x,0x%08x\n",
    sizeof(dwValue), &dwValue.value, &dwValue.words,
    dwValue.words.first, dwValue.words.second);
```

---

Этот вызов printf() выводит следующий результат:

---

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

---

Первое правило иллюстрируется значением Size, которое напечатано первым. Хотя dwValue содержит три члена, которые занимают в общей сложности 9 байт, его размер составляет всего 4 байта. Этот размер подтверждает второе правило, так как dwValue.value и dwValue.words оба указывают на адрес 0x2efda8, что видно из значений, напечатанных после слова Addresses.

Второе правило также подтверждается тем, что dwValue.words.first и dwValue.words.second содержат 0xbeef и 0xdead, напечатанные после Values, что логично, поскольку dwValue.value равен 0xdeadbeef.

Третье правило в этом примере не демонстрируется, так как у нас недостаточно памяти для анализа выравнивания. Однако, если бы этот union был помещён внутрь структуры и окружён другими типами, компилятор всегда выровнял бы его, как DWORD.

## Простой класс (A Simple Class)

Классы с обычными функциями, такие как bar в Листинге 4-4, соответствуют той же компоновке в памяти, что и структуры.

---

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

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

---

Листинг 4-4: Класс C++

Вызов printf() в Листинге 4-4 выдаст следующий результат:

---

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

---

Хотя bar содержит две функции-члена, этот вывод показывает, что он занимает всего 8 байт, необходимых для хранения bar1 и bar2. Это связано с тем, что класс bar не включает абстракцию для этих функций-членов, и программа может обращаться к ним напрямую.

**Н О Т Е** Уровни доступа, такие как публичный, частный защищенный не проявляются в памяти. Независимо от этих модификаторов, члены классов все равно упорядочиваются так, как они определены.

## Класс с виртуальными функциями (A Class with Virtual Functions)

В классах, которые включают абстрактные функции (часто называемые виртуальными функциями), программа должна знать, какую функцию вызывать. Рассмотрим определения классов в Листинге 4-5:

---

```
class foo {
public:
    foo() : myValue1(0xDEADBEEF), myValue2(0xBABABA) {}
    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"); }
};
```

---

*Листинг 4-5: Сайт foo, fooaa и foob классы*

Класс foo имеет три виртуальные функции: bar, baz и barbaz. Классы fooaa и foob наследуются от класса foo и перегружают как bar, так и baz.

Поскольку fooaa и foob имеют публичный базовый класс foo, указатель foo может указывать на них, но программа все равно должна вызывать правильные версии bar и baz. Вы можете увидеть это, выполнив следующий код:

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

А вот и вывод:

```
call fooaa::bar()
```

Вывод показывает, что \_testfoo->bar() вызвал fooaa::bar(), даже несмотря на то, что \_testfoo является указателем foo. Программа знала, какую версию функции вызывать, потому что компилятор включил таблицу VF (виртуальных функций) (VF (virtual function) table) в память \_testfoo.

Таблицы VF — это массивы адресов функций, которые экземпляры абстрактных классов используют для передачи программе информации о местонахождении их перегруженных функций.

## Экземпляры классов и таблицы виртуальных функций (Class Instances and Virtual Function Tables)

Чтобы понять взаимосвязь между экземплярами классов и таблицами VF, давайте рассмотрим дамп памяти трех объектов, объявленных в этом листинге:

```
foo _foo = foo();
fooaa _fooaa = fooaa();
foob _foob = foob();
```

Эти объекты относятся к типам, определенным в Листинге 4-5. Вы можете увидеть их в памяти на Рисунке 4-4.

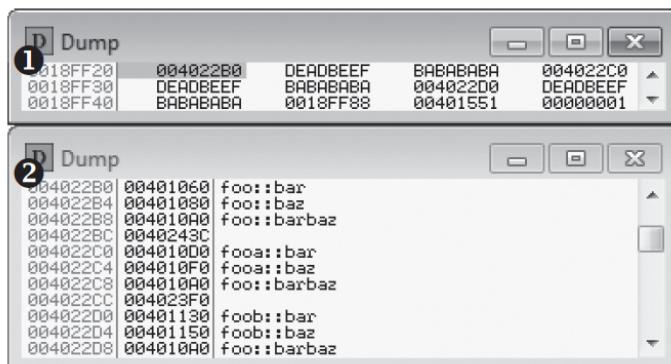


Рисунок 4-4: OllyDbg дамп памяти данных класса

❶ Панель показывает, что каждый экземпляр класса хранит свои члены так же, как структура, но перед ними добавляется значение DWORD, которое указывает на таблицу VF данного экземпляра класса.

❷ Панель показывает таблицы VF для каждого из трех экземпляров класса. Связь между кодом и памятью представлена в Таблице 4-4, которая демонстрирует, как эти панели и код объединяются.

**Таблица 4-4:** Переход от памяти к коду для листинга 4-5 и рисунка 4-4

Адрес	Размер	Данные	Объект
<b>Pane 1</b>			
0x0018FF20	4 байта	0x004022B0	Начало _foo и указатель на таблицу VF
0x0018FF24	8 байтов	0xDEADBEEF 0xBABABABA	_foo.myValue1 и _foo.myValue2
0x0018FF2C	4 байта	0x004022C0	Начало _fooa и указатель на таблицу VF
0x0018FF30	8 байтов	0xDEADBEEF 0xBABABABA	_fooa.myValue1 и _fooa.myValue2
0x0018FF38	4 байта	0x004022D0	Начало _foob и указатель на таблицу VF
0x0018FF3C	8 байтов	0xDEADBEEF 0xBABABABA	_foob.myValue1 и _foob.myValue2
...	...	...	Несвязанные данные
<b>Pane 2</b>			
0x004022B0	4 байта	0x00401060	Начало таблицы VF foo; адрес foo::bar
0x004022B4	4 байта	0x00401080	Адрес foo::baz
0x004022B8	4 байта	0x004010A0	Адрес foo::barbaz
0x004022BC	4 байта	0x0040243C	Несвязанные данные
0x004022C0	4 байта	0x004010E0	Начало таблицы VF fooa; адрес fooa::bar
0x004022C4	4 байта	0x004010F0	Адрес fooa::baz
0x004022C8	4 байта	0x004010A0	Адрес foo::barbaz
0x004022CC	4 байта	0x004023F0	Несвязанные данные
0x004022D0	4 байта	0x00401150	Начало таблицы VF foob; адрес foob::bar
0x004022D4	4 байта	0x00401160	Адрес foob::baz
0x004022D8	4 байта	0x004010A0	↓ Адрес foo::barbaz

Эта таблица показывает, как таблицы VF для кода в Листинге 4-5 размещаются в памяти. Каждая таблица VF создается компилятором при сборке бинарного файла и остается неизменной. Чтобы сэкономить место, экземпляры одного и того же класса ссылаются на одну и ту же таблицу VF, из-за чего таблицы VF не включаются в сам класс.

Поскольку у нас есть три таблицы VF, вы можете задаться вопросом, как экземпляр класса знает, какую таблицу VF использовать. Компилятор вставляет код, похожий на следующий фрагмент ассемблера в каждом виртуальном конструкторе класса:

---

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

---

Этот пример берет статический адрес таблицы VF (VFADDR) и размещает его в памяти в качестве первого члена класса.

Теперь посмотрите на адреса 0x004022B0, 0x004022C0 и

0x004022D0 в Таблице 4-4. Эти адреса содержат начало таблиц VF классов foo, fooa и foob. Обратите внимание, что foo::barbaz присутствует во всех трёх таблицах VF; это потому, что функция не перегружена ни одним из подклассов, что означает, что экземпляры каждого подкласса будут вызывать исходную реализацию напрямую.

Также обратите внимание, что foo::myStaticValue не появляется в этой таблице соответствия. Поскольку это значение статическое, оно фактически не должно существовать как часть класса foo; оно включено в этот класс только для лучшей организации кода. В реальности оно обрабатывается как глобальная переменная и размещается в другом месте.

#### ● ТАБЛИЦЫ VF И CHEAT ENGINE

Помните, что первый элемент структуры указателей Cheat Engine должен указывать на параметр модуля для сканирования указателей (Figure 1-4, страница 142). Теперь, когда вы прочитали немного о таблицах VF, эти знания помогут вам понять, как работает этот параметр: он заставляет Cheat Engine игнорировать все области кучи, где первый член не является указателем на действительную таблицу VF. Это ускоряет сканирование, но работает только в том случае, если каждый шаг в пути указателя является частью экземпляра абстрактного класса.

Тур по памяти заканчивается здесь, но если у вас возникнут трудности с идентификацией блока данных в будущем, вернитесь к этому разделу для справки. Далее мы рассмотрим, как компьютер может понимать написанный на высокогоревневом языке исходный код игры.

## Интенсивный курс по ассемблеру x86 (x86 Assembly Crash Course)

Когда исходный код программы компилируется в бинарный файл, из него удаляются все ненужные артефакты, и он переводится в машинный код. Этот машинный код, состоящий только из байтов (байты команд называются опкодами (opcodes), но есть и байты, представляющие операнды), поступает непосредственно в процессор и указывает ему, как именно следует работать. Эти 0 и 1 переключают транзисторы для выполнения вычислений, и их может быть крайне сложно понять. Чтобы сделать работу с таким кодом немного проще, инженеры используют ассемблер (assembly language) — упрощенный язык, который представляет машинные опкоды в виде сокращенных имен (так называемых мнемоник (mnemonics)) и простой синтаксической структуры.

Ассемблер важен для тех, кто занимается хакингом игр, поскольку мощные хаки можно осуществить только с помощью прямой модификации ассемблерного кода игры, например, с помощью методов NOPing или hooking. В этом разделе вы изучите основы ассемблера x86 (x86 assembly language) — варианта ассемблера, предназначенного для работы с 32-битными процессорами. Ассемблерный язык очень обширен, поэтому в целях краткости здесь рассматривается только небольшой набор

концепций, наиболее полезных для хакинга игр.

**Н О Т Е** В этом разделе многие небольшие фрагменты ассемблерного кода включают комментарии, начинающиеся с точки с запятой (;), чтобы более подробно объяснить каждую инструкцию.

## Синтаксис команды

Ассемблерный язык используется для описания машинного кода, поэтому его синтаксис довольно прост. Этот синтаксис облегчает понимание отдельных команд (также называемых операциями (operations)), но в то же время делает сложные блоки кода трудными для восприятия. Даже алгоритмы, которые легко читаются на языках высокого уровня, могут выглядеть запутанными, если записаны на ассемблере. Например, рассмотрим следующий фрагмент псевдокода:

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

в ассемблере x86 будет выглядеть как Листинг 4-6.

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

Листинг 4-6: Некоторые команды ассемблера x86

Поэтому требуется обширная практика, чтобы понять даже самые тривиальные функции в ассемблере. Однако понимание отдельных команд очень простое, и к концу этого раздела вы будете знать, как разбирать команды, которые я только что вам показал.

## Инструкции (Instructions)

Первая часть команды ассемблера называется инструкцией. Если приравнять команду ассемблера к команде терминала, то инструкция — это программа для выполнения. На уровне машинного кода инструкции, как правило, являются первым байтом команды<sup>1</sup>; также существуют 2-байтовые инструкции, где первым байтом является 0x0F. В любом случае инструкция точно указывает процессору, что делать. В Листинге 4-6 (Listing 4-6), CMP, JQ, MOV и JMP — это все инструкции.

<sup>1</sup> Рэндалл Хайд Искусство языка ассемблера, 2-е издание (No Starch Press, 2010) - это замечательная книга, которая научит вас всему, что нужно знать об ассемблере.

<sup>2</sup> Каждая команда должна укладываться в 15 байт. Большинство команд состоит из 6 или менее человек.

## Синтаксис operandов (Operand Syntax)

Хотя некоторые инструкции являются полными командами, подавляющее большинство остаётся незавершёнными, если за ними не следуют operandы (или параметры). Каждая команда в Листинге 4-6 (Listing 4-6) содержит как минимум один operand, например EBX, EAX и label1.

Operandы в ассемблере бывают трёх типов:

Непосредственное значение (Immediate value) — Целочисленное значение, которое объявляется внутри команды (шестнадцатеричные значения оканчиваются на h).

Регистр (Register) — Имя, которое относится к регистру процессора.

Смещение памяти (Memory offset) — Выражение, заключённое в квадратные скобки, которое представляет адрес памяти значения. Выражение может быть непосредственным значением или регистром. Кроме того, оно может представлять сумму или разность регистра и непосредственного значения (например, [REG+Ah] или [REG-10h]).

Каждая инструкция в x86-ассемблере может содержать от 0 до 3 operandов, и запятые используются для разделения нескольких operandов. В большинстве случаев инструкции, требующие двух operandов, содержат исходный operand (source operand) и целевой operand (destination operand). Порядок operandов зависит от синтаксиса ассемблера.

Например, Листинг 4-7 (Listing 4-7) содержит группу псевдокоманд, написанных в Intel-синтаксисе, который используется Windows (а значит, и хакерами игр под Windows).

```
MOV R1, 1          ; установить R1 (регистр) в 1 (непосредственное значение)
❶ MOV R1, [BADFOODh] ; установить R1 в значение по адресу [BADFOODh] (смещение памяти)
MOV R1, [R2+10h]   ; установить R1 в значение по адресу [R2+10h] (смещение памяти)
MOV R1, [R2-20h]   ; установить R1 в значение по адресу [R2-20h] (смещение памяти)
```

Листинг 4-7: Демонстрация синтаксиса Intel

В Intel-синтаксисе целевой operand указывается первым, за ним следует исходный operand, поэтому в ❶ R1 является целевым operandом, а [BADFOODh] — исходным.

С другой стороны, компиляторы, такие как GCC (которые могут использоваться для написания ботов на Windows), используют синтаксис, известный как AT&T или UNIX-синтаксис. Этот синтаксис работает немного иначе, как показано в следующем примере:

```
MOV $1, %R1        ; установить R1 (регистр) в 1 (непосредственное значение)
MOV 0xBADFOOD, %R1 ; установить R1 в значение по адресу 0xBADFOOD (смещение памяти)
MOV 0x10(%R2), %R1 ; установить R1 в значение по адресу 0x10(%R2) (смещение памяти)
MOV -0x20(%R2), %R1 ; установить R1 в значение по адресу -0x20(%R2) (смещение памяти)
```

Этот код представляет собой AT&T-версию Листинга 4-7 (Listing

4-7). AT&T-синтаксис не только меняет порядок операндов, но также требует префиксов операндов и использует другой формат для операндов смещения памяти.

## Команды ассемблера (Assembly Commands)

Как только вы разберётесь с инструкциями ассемблера и с тем, как форматировать их операнды, вы сможете начать писать команды. В следующем коде показана ассемблерная функция, состоящая из очень простых команд, которая фактически ничего не делает.

```
PUSH EBP      ; поместить EBP (регистр) в стек
MOV EBP, ESP  ; присвоить EBP значение ESP (регистр, вершина стека)
PUSH -1       ; поместить -1 (непосредственное значение) в стек
ADD ESP, 4    ; отменить "PUSH -1", чтобы вернуть ESP обратно (PUSH
              ; уменьшает ESP на 4, так как стек растёт вниз)
MOV ESP, EBP  ; присвоить ESP значение EBP (они останутся одинаковыми,
              ; так как мы не изменили ESP)
POP EBP       ; присвоить EBP значение с вершины стека (оно будет таким же,
              ; с каким EBP начинался, так как его сначала сохранил PUSH EBP)
XOR EAX, EAX  ; выполнить "исключающее ИЛИ" (XOR) для регистра EAX с самим собой
              ; (тот же эффект, что и 'MOV EAX, 0', но намного быстрее)
RETN          ; выйти из функции, вернув значение 0 (EAX обычно содержит возвращаемое значение)
```

Первые две строки (PUSH и MOV) создают кадр стека. Затем в стек помещается -1, но это действие отменяется командой ADD ESP, 4, которая возвращает стек в исходное состояние. Затем кадр стека удаляется, возвращаемое значение (хранящееся в EAX) обнуляется с помощью инструкции XOR, и функция завершает выполнение.

Вы узнаете больше о кадрах стека и функциях в разделе "Стек вызовов" (The Call Stack, стр. 86) и "Вызовы функций" (Function Calls, стр. 94). Пока что обратите внимание на константы в коде, а именно EBP, ESP и EAX, которые часто используются в коде в качестве operandов. Эти значения, наряду с другими, называются регистрами процессора (processor registers), и их понимание имеет решающее значение для работы со стеком, вызовами функций и другими низкоуровневыми аспектами ассемблерного кода.

## Регистры процессора (Processor Registers)

В отличие от языков высокого уровня, язык ассемблера не использует именованные переменные. Вместо этого он обращается к данным через их адреса в памяти. Однако при интенсивных вычислениях процессору может быть очень затратно постоянно работать с чтением и записью данных в RAM.

Чтобы снизить эту нагрузку, x86-процессоры предоставляют небольшой набор временных переменных, называемых регистрами процессора. Эти регистры — это небольшие области хранения внутри самого процессора. Доступ к ним требует гораздо меньших затрат, чем доступ к RAM, поэтому ассемблер использует их для описания своего внутреннего состояния, передачи

временных данных и хранения контекстно-зависимых переменных.

## Общие регистры (General Registers)

Когда ассемблерный код должен хранить или обрабатывать произвольные данные, он использует подмножество процессорных регистров, называемых общими регистрами (general registers). Эти регистры используются исключительно для хранения данных процесса, таких как локальные переменные функции.

Каждый общий регистр имеет размер 32 бита, поэтому его можно рассматривать как переменную DWORD.

Также общие регистры оптимизированы для выполнения определённых задач:

EAX, аккумулятор (EAX, the accumulator). Этот регистр оптимизирован для математических вычислений. Некоторые операции, такие как умножение и деление, могут выполняться только в EAX.

EBX, базовый регистр (EBX, the base register) Этот регистр используется для дополнительного хранения данных. В 16-битных процессорах единственным регистром, который можно было использовать для работы с адресами памяти, был BX, поэтому EBX применялся как ссылка на RAM. Однако в x86-ассемблере теперь любой регистр может использоваться в качестве ссылки на адрес, поэтому EBX утратил своё изначальное назначение.

ECX, счётчик (ECX, the counter) Этот регистр оптимизирован для работы в качестве переменной-счётчика (в языках высокого уровня часто используется как i в циклах).

EDX, регистр данных (EDX, the data register) Этот регистр оптимизирован для работы в паре с EAX. Например, в 64-битных вычислениях:

EAX представляет биты 0–31,

EDX представляет биты 32–63.

Эти регистры также содержат набор 8- и 16-битных подрегистров, которые позволяют обращаться к частям данных.

Можно представить каждый общий регистр как union, где:

основное имя регистра обозначает 32-битное хранилище,

подрегистры предоставляют доступ к меньшим частям этого регистра.

Следующий код показывает, как может выглядеть union для регистра EAX:

---

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

---

В этом примере AX предоставляет доступ к младшему WORD регистра EAX, а AL — к младшему BYTE AX, а AH — к старшему

BYTE.

Все общие регистры имеют такую структуру, и ниже представлены подрегистры других регистров в Рисунке 4-5 (Figure 4-5).

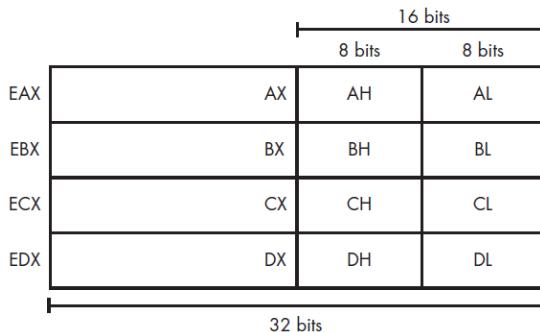


Рисунок 4-5: Регистры и подрегистры x86

EAX, EBX, ECX и EDX также содержат старшие слова, но компилятор почти никогда не использует их напрямую, так как он может просто работать с младшим словом, если требуется только хранение WORD.

## Индексные регистры (Index Registers)

Ассемблер x86 также имеет четыре индексных регистра (index registers), которые используются для:

доступа к потокам данных,  
обращения к стеку вызовов,  
отслеживания локальной информации.

Как и общие регистры, индексные регистры имеют размер 32 бита, но их предназначение более строго определено:

EDI, индекс назначения (EDI, the destination index) Этот регистр используется для индексирования памяти при записи данных. Если в коде нет операций записи, компилятор может использовать EDI как произвольное хранилище.

ESI, индекс источника (ESI, the source index) Этот регистр используется для индексирования памяти при чтении данных. Также может использоваться произвольно.

ESP, указатель стека (ESP, the stack pointer) Этот регистр указывает на вершину стека вызовов. Все операции со стеком обращаются к нему напрямую. ESP можно использовать только при работе со стеком, и он всегда должен указывать на вершину стека.

EBP, базовый указатель стека (EBP, the stack base pointer) Этот регистр отмечает дно кадра стека. Функции используют его для ссылки на свои параметры и локальные переменные. Однако в некоторых случаях компилятор может исключить это поведение, и тогда EBP можно использовать произвольно.

Как и общие регистры, каждый индексный регистр имеет 16-битный аналог: DI, SI, SP и BP.

Однако у индексных регистров нет 8-битных подрегистров.

#### Почему у некоторых x86-регистров есть подрегистры?

Это связано с историческими причинами. Архитектура x86 изначально была 16-битной, и в 32-битных процессорах регистры AX, BX, CX, DX, DI, SI, SP и BP были расширены. Расширенные версии получили тот же суффикс, но с префиксом "E" (что означает "extended", расширенный). 16-битные версии остались для обратной совместимости. Это также объясняет, почему у индексных регистров нет 8-битных подрегистров: они предназначены для работы со смещениями памяти, и нет практической необходимости обращаться к их отдельным байтам.

## Регистр индекса выполнения (The Execution Index Register)

Регистр индекса выполнения, обозначаемый как EIP, имеет очень конкретное назначение: он указывает на адрес кода, который в данный момент выполняется процессором.

Поскольку этот регистр управляет потоком выполнения, он непосредственно инкрементируется процессором и недоступен для прямого изменения в ассемблерном коде.

Чтобы изменить EIP, ассемблерный код должен обращаться к нему косвенно, используя такие инструкции, как CALL, JMP и RETN.

## Регистр EFLAGS (The EFLAGS Register)

В отличие от языков высокого уровня, ассемблер не имеет бинарных операторов сравнения вроде ==, >, <.

Вместо этого используется команда CMP, которая сравнивает два значения и сохраняет результат в регистре EFLAGS. После этого код может изменять свой поток выполнения, используя специальные операции, зависящие от значений, хранящихся в EFLAGS.

Хотя команды сравнения являются единственными пользовательскими операциями, которые могут обращаться к EFLAGS, они используют только "бит состояния" (status bits): 0, 2, 4, 6, 7 и 11.

Биты 8–10 используются как флаги управления (control flags).

Биты 12–14 и 16–21 работают как системные флаги (system flags).

Оставшиеся биты зарезервированы для процессора.

Таблица 4-5 (Table 4-5) показывает тип, имя и описание каждого бита регистра EFLAGS.

**Таблица 4-5: Биты EFLAGS**

Бит(ы)	Тип	Имя	Описание
0	Статус (Status)	Перенос (Carry)	Устанавливается, если при предыдущей инструкции произошло заимствование или перенос из старшего бита.
2	Статус (Status)	Чётность (Parity)	Устанавливается, если младший байт результата предыдущей инструкции содержит чётное число установленных битов.
4	Статус (Status)	Коррекция (Adjust)	Аналогичен флагу переноса, но учитывает 4 младших значащих бита.
6	Статус (Status)	Ноль (Zero)	Устанавливается, если результат предыдущей инструкции равен 0.
7	Статус (Status)	Знак (Sign)	Устанавливается, если результат предыдущей инструкции имеет установленный знаковый бит (старший бит).
8	Управление (Control)	Ловушка (Trap)	Если установлен, процессор отправляет прерывание ядру ОС после выполнения следующей операции.
9	Управление (Control)	Прерывание (Interrupt)	Если не установлен, система игнорирует маскируемые прерывания.
10	Управление (Control)	Направление (Direction)	Если установлен, ESI и EDI уменьшаются в операциях, которые их изменяют автоматически. Если не установлен, они увеличиваются.
11	Статус (Status)	Переполнение (Overflow)	Устанавливается, если в предыдущей инструкции произошло переполнение значения (например, если ADD выполняется над положительным значением, но результат становится отрицательным).

Регистр EFLAGS также содержит системные биты и зарезервированные биты, но они не используются в пользовательском режиме и при создании игровых читов, поэтому не включены в таблицу.

При отладке игрового кода важно помнить о EFLAGS. Например, если вы установите точку останова на JE (jump if equal), вы можете проверить бит 0 регистра EFLAGS, чтобы определить, будет ли выполнен переход.

## Регистры сегментов (Segment Registers)

В языке ассемблера также существует набор 16-битных регистров, называемых регистрами сегментов (segment registers).

В отличие от других регистров, они не используются для хранения данных, а только для их адресации.

Теоретически, они указывают на изолированные сегменты памяти.

Различные типы данных могут храниться в полностью отдельных сегментах памяти. Реализация такой сегментации остается на усмотрение операционной системы. Эти x86-сегментные регистры и их назначение: CS, сегмент кода (CS, the code segment) Этот регистр указывает на память, содержащую код приложения. DS, сегмент данных (DS, the data segment) Этот регистр указывает на память, содержащую данные приложения. ES, FS и GS, дополнительные сегменты (ES, FS, and GS, the extra

segments) Эти регистры указывают на проприетарные сегменты памяти, используемые операционной системой. SS, сегмент стека (SS, the stack segment) Этот регистр указывает на память, которая используется в качестве выделенного стека вызовов.

В ассемблерном коде сегментные регистры используются как префиксы к операндам с адресацией по смещению. Если сегментный регистр не указан явно, по умолчанию используется DS. Это означает, что команда PUSH [EBP] фактически эквивалентна PUSH DS:[EBP]. Однако команда PUSH FS:[EBP] работает иначе: она считывает данные из сегмента FS, а не из сегмента DS.

Если внимательно изучить реализацию сегментации памяти Windows x86, можно заметить, что эти сегментные регистры на самом деле не использовались так, как это предполагалось изначально. Чтобы увидеть это на практике, можно выполнить следующие команды в командной строке плагина OllyDbg, когда OllyDbg подключен к приостановленному процессу.

```
? CALC (DS==SS && SS==GS && GS==ES)
? 1
? CALC DS-CS
? 8
? CALC FS-DS
; возвращет ненулевое значение (и изменяется между потоками)
```

Этот вывод даёт нам три важных факта. Во-первых, он показывает, что Windows использует только три сегмента: FS, CS и всё остальное. Это подтверждается тем, что DS, SS, GS и ES равны. По той же причине этот вывод показывает, что DS, SS, GS и ES можно использовать взаимозаменяя, так как они указывают на одни и те же сегменты памяти.

Во-вторых, поскольку FS изменяется в зависимости от потока, это означает, что он зависит от конкретного потока. FS — это интересный сегментный регистр, так как он указывает на данные, специфичные для потока. В разделе "Обход ASLR в продакшене" (Bypassing ASLR in Production, стр. 128) будет рассмотрено, как данные в FS можно использовать для обхода ASLR, что является важным для большинства ботов.

На практике, если ассемблерный код генерирован компилятором для Windows, в нём будет использоваться всего три сегмента: DS, FS и SS. Интересно, что CS хоть и показывает постоянное смещение от DS, но не имеет реальной роли в коде пользовательского режима.

Зная всё это, можно сделать вывод, что в Windows фактически используются только два сегмента: FS и всё остальное. Эти два сегмента фактически указывают на разные области в одной и той же памяти (нет простого способа это проверить, но это правда), что показывает, что Windows на самом деле вообще не использует сегменты памяти. Вместо этого он использует плоскую модель памяти, в которой сегментные регистры почти не имеют значения.

Хотя все сегментные регистры указывают на одну и ту же память, только FS и CS указывают на разные области, и CS не используется. В заключение, есть три вещи, которые нужно знать о сегментных регистрах при работе с x86-ассемблером в Windows:

DS, SS, GS и ES взаимозаменяемы, но для ясности DS должен использоваться для доступа к данным, а SS — для доступа к стеку вызовов.

CS можно спокойно забыть.

FS — единственный сегментный регистр со специальным назначением, его лучше пока не трогать.

## Стек вызовов

Регистры являются мощным инструментом, но, к сожалению, их количество очень ограничено. Чтобы ассемблерный код мог эффективно хранить все свои локальные данные, он также должен использовать стек вызовов (call stack). Стек используется для хранения множества различных значений, включая параметры функций, адреса возврата и некоторые локальные переменные.

Понимание принципов работы стека вызовов окажется полезным при реверс-инжиниринге игр. Более того, эти знания будут необходимы, когда мы перейдём к манипуляции потоком выполнения в Главе 8.

## Структура

Стек вызовов можно представить как FILO (first-in-last-out, "первым пришёл — последним вышел") — список значений DWORD, которыми можно непосредственно управлять в ассемблерном коде. Термин "стек" используется, потому что его структура напоминает стопку бумаги: объекты добавляются и удаляются только сверху.

Данные добавляются в стек с помощью команды операнда PUSH. Данные извлекаются из стека (и загружаются в регистр) с помощью команды POP.

Рисунок 4-6 показывает, как выглядит этот процесс.

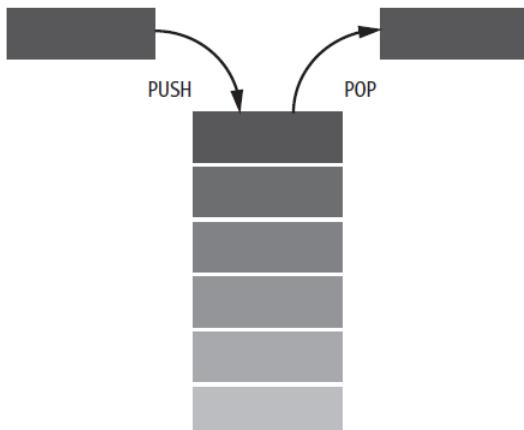


Рисунок 4-6: Структура стека

В Windows стек растёт от больших адресов памяти к меньшим. Он занимает ограниченный блок памяти, заполняясь до адреса 0x00000000 (абсолютный верх) от адреса n (абсолютный низ). Это означает, что ESP (указатель вершины стека) уменьшается по мере добавления элементов и увеличивается при их удалении.

## Кадр стека (The Stack Frame)

Когда ассемблерная функция использует стек для хранения данных, она обращается к этим данным, создавая кадр стека (stack frame).

Это делается сохранением ESP в EBP и последующим вычитанием n байтов из ESP, создавая промежуток размером n байтов между регистрами EBP и ESP.

Чтобы лучше понять это, представьте, что стек на рисунке 4-7 передаётся в функцию, которая требует 0x0C байтов локальной памяти. В этом примере адрес 0x0000 — абсолютная вершина стека. У нас есть неиспользуемая память в диапазоне от 0x0000 до 0xFF00 - 4, и в момент вызова функции адрес 0xFF00 является вершиной стека.

ESP указывает на этот адрес. Память после 0xFF00 используется предыдущими функциями в цепочке вызовов (от 0xFF04 до 0xFFFF). Когда функция вызывается, первое, что она делает, — выполняет следующий ассемблерный код, который создаёт кадр стека размером 0x0C (12 в десятичной системе) байтов.

```
PUSH EBP      ; сохраняет нижнюю границу нижнего кадра стека
MOV EBP, ESP  ; сохраняет нижнюю границу текущего кадра стека в EBP
              ; (также на 4 байта выше нижнего кадра стека)
SUB ESP, 0x0C  ; вычитает 0x0C байтов из ESP, перемещая его вверх по стеку
              ; чтобы отметить вершину кадра стека
```

После выполнения этого кода стек приобретает вид, представленный на рисунке 4-8. После создания этого стека функция может работать с 0x0C байтами, выделенными в стеке. 0x0000 по-прежнему остается абсолютной вершиной стека. У нас есть неиспользуемая память стека в диапазоне от 0x0000 до 0xFF00 – 20, а память по адресу 0xFF00 – 16 содержит последние 4 байта локального хранилища (обращение происходит через [EBP-0Ch]). Этот адрес также является верхней границей текущего кадра стека, поэтому ESP указывает сюда. 0xFF00 – 12 содержит средние 4 байта локального хранилища (обращение через [EBP-8h]), и 0xFF00 – 8 содержит первые 4 байта локального хранилища (обращение через [EBP-4h]). EBP указывает на 0xFF00 – 4, что является нижней границей текущего кадра стека; этот адрес хранит исходное значение регистра EBP. 0xFF00 — это верхняя граница нижнего кадра стека, и оригинальный ESP на рисунке 4-



Figure 4-7: Initial example stack  
(read from bottom to top)

7 указывал именно сюда. Наконец, можно увидеть область памяти стека, используемую предыдущими функциями в цепочке вызовов, в диапазоне от 0xFF04 до 0xFFFF.

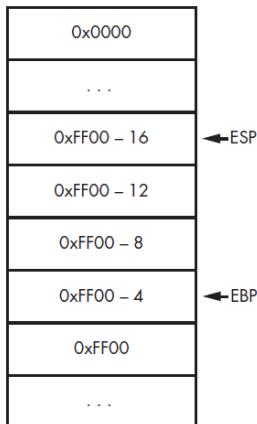


Рисунок 4-8: Пример стека с установленной рамкой стека (читать снизу вверх)

С текущим состоянием стека функция может свободно использовать свои локальные данные. Если эта функция вызывает другую функцию, новая функция создаст свой собственный кадр стека, используя тот же метод (кадры стека действительно накапливаются).

Когда функция завершает работу с кадром стека, она должна восстановить стек в его предыдущее состояние. В нашем случае это означает, что стек должен выглядеть как на рисунке 4-7.

Когда вторая функция завершает выполнение, первая функция очищает стек, используя следующие две команды:

```
MOV ESP, EBP ; разрушает кадр стека, перемещая ESP на 4 байта выше  
; его исходного значения (0xFF00-4)  
POP EBP ; восстанавливает нижнюю границу старого кадра стека, сохранённого  
; командой "PUSH EBP". Также добавляет 4 байта к ESP,  
; возвращая его в исходное положение
```

Если вы хотите изменить параметры, переданные функции в игре, не ищите их в кадре стека самой функции.

Параметры хранятся в кадре стека вызывающей функции и доступны по адресам [EBP+8h], [EBP+C<sub>h</sub>] и далее. Они начинаются с [EBP+8h], потому что [EBP+4h] содержит адрес возврата функции.

(Тема подробнее рассматривается в разделе "Function Calls" на странице 94.)

**Н О Т Е** Код может быть скомпилирован без кадров стека. Если это так, вы заметите, что функции не начинают выполнение с PUSH EBP, а вместо этого обращаются ко всему относительно ESP. Однако, чаще всего кадры стека включены в скомпилированном игровом коде.

Теперь, когда вы знаете основы ассемблерного кода, давайте рассмотрим некоторые особенности, которые пригодятся вам при взломе игр.

## Важные инструкции x86 для взлома игр

Хотя в языке ассемблера сотни инструкций, многие хорошо подготовленные игровые хакеры понимают лишь небольшое их подмножество, о котором я подробно рассказываю здесь. Это подмножество обычно включает в себя все инструкции, которые используются для модификации данных, вызова функций, сравнения значений или перемещения по коду.

### Изменение данных (Data Modification)

Изменение данных часто выполняется в несколько этапов ассемблерных операций, но в конечном итоге результат должен быть сохранён либо в памяти, либо в регистре. Обычно для этого используется инструкция MOV.

MOV принимает два операнда: целевой (destination) и источник (source). Таблица 4-6 показывает все возможные комбинации operandов MOV и ожидаемые результаты.

Таблица 4-6: Операторы для команды MOV Инструкция

Синтаксис инструкции	Результат
MOV R1, R2	Копирует значение R2 в R1.
MOV R1, [R2]	Копирует значение из памяти по адресу R2 в R1.
MOV R1, [R2+Ah]	Копирует значение из памяти по адресу R2+0xA в R1.
MOV R1, [DEADBEEFh]	Копирует значение из памяти по адресу 0xDEADBEEF в R1.
MOV R1, BADFOODh	Копирует значение 0xBADFOOD в R1.
MOV [R1], R2	Копирует значение R2 в память по адресу R1.
MOV [R1], BADFOODh	Копирует значение 0xBADFOOD в память по адресу R1.
MOV [R1+4h], R2	Копирует значение R2 в память по адресу R1+0x4.
MOV [R1+4h], BADFOODh	Копирует значение 0xBADFOOD в память по адресу R1+0x4.
MOV [DEADBEEFh], R1	Копирует значение R1 в память по адресу 0xDEADBEEF.
MOV [DEADBEEFh], BADFOODh	Копирует значение 0xBADFOOD в память по адресу 0xDEADBEEF.

Инструкция MOV может использовать множество комбинаций operandов, но некоторые недопустимы.

Во-первых, целевой operand не может быть непосредственным значением; он должен быть регистром или адресом памяти, поскольку непосредственные значения нельзя изменить.

Во-вторых, значения нельзя копировать напрямую из одной

ячейки памяти в другую.

Копирование значения требует двух отдельных операций, например:

```
MOV EAX, [EBP+10h] ; копируем данные из памяти по адресу EBP+0x10 в EAX  
MOV [DEADBEEFh], EAX ; сохраняем скопированные данные в памяти по адресу 0xDEADBEEF
```

Эти инструкции копируют данные, хранящиеся по адресу EBP+0x10, в память по адресу 0xDEADBEEF.

## Арифметика (Arithmetic)

Как и во многих языках высокого уровня, ассемблерный язык имеет два типа арифметических операций:

- унарные (unary),
- бинарные (binary).

Унарные инструкции принимают единственный operand, который выступает одновременно как источник и цель операции. Этот operand может быть регистром или адресом памяти.

Таблица 4-7 показывает распространённые унарные арифметические инструкции в x86.

**Таблица 4-7:** Унарные арифметические инструкции

Синтаксис инструкции	Результат
INC operand	Прибавляет 1 к значению операнда.
DEC operand	Вычитает 1 из значения операнда.
NOT operand	Логически инвертирует значение операнда (переворачивает все биты).
NEG operand	Выполняет двухкомплементарное отрицание (переворачивает все биты и прибавляет 1; фактически умножает на -1 ).

## Бинарные инструкции (Binary Instructions)

Бинарные инструкции (которые составляют большинство арифметических операций в x86) по синтаксису похожи на инструкцию MOV. Они требуют два операнда и имеют схожие ограничения. Однако, в отличие от MOV, их целевой operand выполняет вторую роль: он также является левым operandом в вычислении. Например, операция ADD EAX, EBX в ассемблере эквивалентна  $EAX = EAX + EBX$  или  $EAX += EBX$  в C++. Таблица 4-8 показывает распространённые бинарные арифметические инструкции в x86.

**Таблица 4-8:** Инструкции двоичной арифметики

Синтаксис инструкции	Функция	Примечания к операндам
<code>ADD destination, source</code>	<code>destination += source</code>	—
<code>SUB destination, source</code>	<code>destination -= source</code>	—
<code>AND destination, source</code>	<code>destination &amp;= source</code>	—
<code>OR destination, source</code>	<code>destination  = source</code>	<code>= source</code>
<code>XOR destination, source</code>	<code>destination ^= source</code>	—
<code>SHL destination, source</code>	<code>destination = destination &lt;&lt; source</code>	<code>source</code> должен быть CL или 8-битным непосредственным значением.
<code>SHR destination, source</code>	<code>destination = destination &gt;&gt; source</code>	<code>source</code> должен быть CL или 8-битным непосредственным значением.
<code>IMUL destination, source</code>	<code>destination *= source</code>	<code>destination</code> должен быть регистром; <code>source</code> не может быть непосредственным значением.

Из этих арифметических инструкций IMUL является особенной, потому что ей можно передать третий operand в виде непосредственного значения. В этом случае целевой operand больше не участвует в вычислении, которое теперь выполняется между оставшимися operandами. Например, команда IMUL EAX, EBX, 4h эквивалентна  $EAX = EBX * 0x4$  в C++. Можно также передать один operand в IMUL (Примечание: Также существует инструкция беззнакового умножения MUL, которая работает только с одним operandом.). В этом случае operand выступает в качестве источника и может быть либо адресом памяти, либо регистром. В зависимости от размера исходного operandа инструкция использует разные части регистра EAX для ввода и вывода, как показано в Таблице 4-9.

**Таблица 4-9:** Возможные IMUL Операторы регистра

Размер источника	Вход	Выход
8 бит	AL	16 бит, хранится в AH:AL (что равно AX).
16 бит	AX	32 бит, хранится в DX:AX (биты 0–15 в AX, 16–31 в DX).
32 бит	EAX	64 бит, хранится в EDX:EAX (биты 0–31 в EAX, 32–64 в EDX).

Обратите внимание, что даже если входным значением является один регистр, выход занимает два регистра. Это связано с тем, что при умножении результат обычно больше входных

значений.

Рассмотрим пример вычисления с IMUL, использующим один 32-битный операнд:

```
IMUL [BADFOODh] ; 32-битный операнд находится по адресу 0xBADFOOD
```

Эта команда эквивалентна следующему псевдокоду:

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

Точно так же рассмотрим операцию IMUL с 16-битным операндом:

```
IMUL CX ; 16-битный операнд хранится в CX
```

Эквивалент в псевдокоде:

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

Наконец, инструкция IMUL с 8-битным операндом:

```
IMUL CL ; 8-битный операнд хранится в CL
```

Эквивалент в псевдокоде:

```
AX = AL * CL
```

В x86-ассемблере также есть операция деления, выполняемая с помощью инструкции IDIV. Инструкция IDIV принимает один операнд и следует тем же правилам работы с регистрами, что и IMUL. Как показано в Таблице 4-10, операции IDIV требуют два входных операнда и два выходных.

**Таблица 4-10:** Возможно IDIV Операторы регистра

Размер источника	Вход	Выход
8 бит	16 бит, хранится в AH:AL (то есть AX)	Остаток в AH; частное в AL.
16 бит	32 бит, хранится в DX:AX	Остаток в DX; частное в AX.
32 бит	64 бит, хранится в EDX:EAX	Остаток в EDX; частное в EAX.

При делении размер входных данных обычно больше выходных, поэтому входные данные занимают два регистра. Кроме того, операции деления должны сохранять остаток, который записывается в первый входной регистр. Пример 32-битного вычисления с IDIV:

```
MOV EDX, 0      ; старший DWORD в EDX отсутствует, поэтому он устанавливается в 0
MOV EAX, inputValue ; 32-битное входное значение
IDIV ECX        ; делим EDX:EAX на ECX
```

Эквивалент в псевдокоде:

```
EAX = EDX:EAX / ECX ; частное  
EDX = EDX:EAX % ECX ; остаток
```

Эти особенности IDIV и IMUL важно помнить, так как их поведение может показаться запутанным при поверхностном рассмотрении команд.

## Разветвление (Branching)

После вычисления выражения программы могут определять, что выполнять дальше, основываясь на результате, обычно используя конструкции if() или switch(). Однако на уровне ассемблера такие конструкции управления потоком отсутствуют.

Вместо этого ассемблерный код использует регистр EFLAGS для принятия решений и выполнения команд прыжков (jmp), которые позволяют исполнять различные блоки кода. Этот процесс называется разветвлением (branching).

Чтобы получить нужные значения в EFLAGS, ассемблерный код использует две инструкции:

TEST — выполняет логическое AND между операндами.

CMP — выполняет вычитание с учетом знака, сравнивая операнды.

Примечание: Так же, как MUL является беззнаковым аналогом IMUL, инструкция DIV является беззнаковым аналогом IDIV.

Чтобы выполнить разветвление правильно, в коде должна присутствовать команда перехода (jmp), которая идёт сразу после операции сравнения. Каждый тип прыжковой инструкции (jmp instruction) принимает один operand, который указывает адрес кода, на который следует перейти. То, как поведёт себя конкретная инструкция перехода, зависит от статуса битов в EFLAGS.

**Таблица 4-11:** Общие инструкции перехода x86

Инструкция	Название	Поведение
JMP dest	Безусловный переход	Переходит на dest (устанавливает EIP в dest).
JE dest	Переход, если равно (Jump if equal)	Переходит, если ZF (флаг нуля) установлен в 1.
JNE dest	Переход, если не равно (Jump if not equal)	Переходит, если ZF равно 0.
JG dest	Переход, если больше (Jump if greater)	Переходит, если ZF = 0 и SF (флаг знака) равен OF (флагу переполнения).
JGE dest	Переход, если больше или равно (Jump if greater or equal)	Переходит, если SF равен OF.
JA dest	Беззнаковый JG (Unsigned JG)	Переходит, если CF (флаг переноса) равен 0 и ZF равен 0.
JAE dest	Беззнаковый JGE (Unsigned JGE)	Переходит, если CF равен 0.
JL dest	Переход, если меньше (Jump if less)	Переходит, если SF не равен OF.
JLE dest	Переход, если меньше или равно (Jump if less or equal)	Переходит, если ZF равен 1 или SF не равен OF.
JB dest	Беззнаковый JL (Unsigned JL)	Переходит, если CF равен 1.
JBE dest	Беззнаковый JLE (Unsigned JLE)	Переходит, если CF равен 1 или ZF равен 1.
JO dest	Переход, если произошло переполнение (Jump if overflow)	Переходит, если OF равен 1.
JNO dest	Переход, если переполнения нет (Jump if not overflow)	Переходит, если OF равен 0.
JZ dest	Переход, если ноль (Jump if zero)	Переходит, если ZF равен 1 (идентично JE).
JNZ dest	Переход, если не ноль (Jump if not zero)	↓ Переходит, если ZF равен 0 (идентично JNE).

Запомнить, какие флаги управляют инструкциями переходов, может быть сложно, но их назначение четко выражено в названии. Хорошее правило: если переход (JMP) следует за CMP, он эквивалентен соответствующему оператору сравнения. Например, в Таблице 4-11 JE означает "прыжок, если равно" (==). Следовательно, если JE следует за CMP, это эквивалентно оператору ==.

Точно так же:

JGE соответствует >,

JLE соответствует <,

и так далее.

---

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

---

**Листинг 4-8:** Простой условный оператор

Этот оператор if() просто проверяет, больше ли EBX, чем EAX, и устанавливает ECX в зависимости от результата. В ассемблере этот же оператор может выглядеть следующим образом:

```
--snip--  
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:  
① MOV ECX, EDX ; ECX = EDX (if block)  
label2:  
--snip--
```

Ассемблер для оператора if() в Листинге 4-8 начинается с инструкции CMP, затем выполняется условный переход, если EBX больше, чем EAX. Если переход выполняется, EIP устанавливается на блок if в ① с помощью инструкции JG. Если переход не выполняется, код продолжает выполняться последовательно и достигает блока else сразу после инструкции JG. Когда выполнение блока else завершается, безусловная команда JMP устанавливает EIP на 0x7, пропуская блок if.

## Вызовы функций

В ассемблерном коде функции — это изолированные блоки команд, выполняемые с помощью инструкции CALL. Инструкция CALL, принимающая адрес функции в качестве единственного операнда, помещает адрес возврата в стек и устанавливает EIP в значение операнда. Следующий псевдокод показывает, как работает CALL, с адресами памяти слева в шестнадцатеричном формате:

```
0x1: CALL EAX  
0x2: ...
```

Когда выполняется CALL EAX, следующий адрес помещается в стек, а EIP устанавливается в EAX, что показывает, что CALL по сути представляет собой PUSH и JMP. Следующий псевдокод иллюстрирует этот момент:

```
0x1: PUSH 3h  
0x2: JMP EAX  
0x3: ...
```

Хотя между инструкцией PUSH и выполняемым кодом имеется дополнительный адрес, результат остается тем же: перед выполнением кода по адресу EAX адрес кода, следующего за переходом, сохраняется в стеке.

Это происходит для того, чтобы callee (функция, которая вызывается) знала, куда вернуться в caller (функцию, выполняющую вызов) после завершения. Если вызывается функция без параметров, достаточно одной команды CALL. Если

callee принимает параметры, они сначала должны быть помещены в стек в обратном порядке. Следующий псевдокод показывает, как может выглядеть вызов функции с тремя параметрами:

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

Когда выполняется callee, вершина стека содержит адрес возврата, который указывает на код после вызова CALL. Первый параметр (0x100) находится под адресом возврата в стеке. Второй параметр (0x200) расположен ниже первого, затем идет третий параметр (0x300). Callee настраивает фрейм стека, используя смещения памяти от EBP, чтобы ссылаться на каждый параметр.

После завершения выполнения callee восстанавливает стековый кадр caller и выполняет команду RET, которая извлекает адрес возврата из стека и переходит к нему. Поскольку параметры не являются частью стекового кадра callee, они остаются в стеке даже после выполнения RET. Если caller отвечает за очистку стека, он добавляет 12 (3 параметра по 4 байта) к ESP сразу после завершения CALL ECX. Если callee отвечает за очистку, он делает это с помощью RET 12 вместо RET. Эта ответственность определяется конвенцией вызова (calling convention) callee.

Конвенция вызова (calling convention) сообщает компилятору, как ассемблерный код должен передавать параметры, хранить указатели на экземпляры, передавать возвращаемое значение и очищать стек. Разные компиляторы используют различные соглашения о вызовах, но в Таблице 4-12 перечислены четыре, которые наиболее вероятно встречаются хакеру игр.

**Таблица 4-12:** Вызывающие конвенции, которые необходимо знать для взлома игр

Директива	Кто очищает стек	Примечания
_cdecl	caller	Стандартная конвенция в Visual Studio.
_stdcall	callee	Используется в Win32 API-функциях.
_fastcall	callee	Первые два параметра DWORD (или меньше) передаются в ECX и EDX.
_thiscall	callee	Используется для методов классов. Указатель на экземпляр класса передается в ECX.

Столбец Directive в Таблице 4-12 указывает название конвенции вызова, а столбец Cleaner показывает, кто очищает стек — caller (вызывающая функция) или callee (вызванная функция). Во всех четырех конвенциях вызова:

параметры всегда передаются справа налево;  
возвращаемые значения всегда хранятся в EAX.

Это стандарт, но не жесткое правило — он может отличаться в других соглашениях о вызовах.

## **Заключительные мысли (Closing Thoughts)**

Моя цель при написании этой главы заключалась в том, чтобы помочь вам понять память и ассемблер в общем смысле, прежде чем мы углубимся в конкретные аспекты взлома игр. Теперь, когда вы научились мыслить как компьютер, вы достаточно подготовлены, чтобы начать решать более сложные задачи в области судебного анализа памяти. Если вам не терпится увидеть, как все это можно применить на практике, перейдите к главе "Применение перехватов вызовов в Adobe AIR" на странице 169 или "Применение перехватов переходов и VF-хуков в Direct3D" на странице 175.

Если вы хотите на практике поработать с памятью, скомпилируйте пример кода из этой главы и используйте Cheat Engine или OllyDbg, чтобы анализировать, изменять и экспериментировать с памятью, пока не освоите этот процесс. Это важно, потому что в следующей главе мы будем развивать эти навыки, изучая продвинутые методы судебного анализа памяти.

## 5) Продвинутый анализ памяти



Независимо от того, занимаетесь ли вы разработкой игр в качестве хобби или бизнеса, в конце концов вы окажетесь между молотом и наковальней. . . нечленораздельный

дамп памяти. Будь то соревнование с конкурирующим разработчиком ботов за выпуск очень востребованной функции, борьба с постоянным шквалом обновлений от игровой компании или поиск сложной структуры данных в памяти, вам понадобятся первоклассные навыки экспертизы памяти, чтобы одержать победу.

Успешная разработка бота — это шаткое равновесие между скоростью и мастерством, и упорные хакеры должны справиться с этой задачей, быстро выпуская гениальные функции, оперативно реагируя на обновления игры и с готовностью разыскивая даже самые неуловимые части данных. Для этого, однако, требуется полное понимание общих схем памяти, продвинутых структур данных и назначения различных фрагментов данных.

Эти три аспекта экспертизы памяти - пожалуй, самое эффективное оружие в вашем арсенале, и в этой главе вы узнаете, как его использовать. Сначала я расскажу о продвинутых методах сканирования памяти, которые направлены на поиск данных путем понимания их назначения и использования. Далее я расскажу вам, как использовать шаблоны памяти для работы с обновлениями игры и настройки ботов без необходимости переносить все адреса с нуля. В заключение я рассмотрю четыре наиболее распространенные сложные структуры данных в стандартной библиотеке C++ (`std::string`, `std::vector`, `std::list` и `std::map`), чтобы вы могли распознать их в памяти и перечислить их содержимое. Я надеюсь, что к концу этой главы вы будете глубоко разбираться в вопросах форензики памяти и сможете справиться с любой задачей, связанной со сканированием памяти.

## Продвинутое сканирование памяти (Advanced Memory Scanning)

Внутри исходного кода игры каждая часть данных имеет холодное, расчетное определение. Однако, когда игра запущена, все эти данные объединяются, создавая нечто новое. Игроки воспринимают только красивый пейзаж, яркие звуки и напряженные приключения; сами данные, которые создают эти ощущения, не имеют для них значения.

Имея это в виду, представьте себе, что Хакер А только начал разбираться в своей любимой игре, желая автоматизировать некоторые скучные моменты с помощью бота. Он еще не обладает полным пониманием устройства памяти, и для него данные — это всего лишь предположения. Он думает: «У меня 500 единиц здоровья, значит, я могу найти адрес здоровья, попросив Cheat Engine искать 4-байтовое целое число со значением 500.» Хакер А понимает данные точно: это просто информация (значения), хранящаяся в определенных местах (адресах) с заданными структурами (типами).

Теперь представьте Хакера В, который уже досконально изучил игру изнутри и снаружи; он понимает, как игровой процесс изменяет состояние игры в памяти, и данные больше не представляют для него загадку. Он знает, что каждое определенное свойство данных можно выявить, исходя из их назначения. В отличие от Хакера А, Хакер В понимает данные на более глубоком уровне, выходящем за рамки простых объявлений переменных: он рассматривает назначение и использование данных. В этом разделе мы рассмотрим оба подхода.

Каждая единица данных в игре имеет свое назначение, и ассемблерный код игры в какой-то момент должен ссылаться на эти данные для выполнения их функций. Найти уникальный код, который использует данные, значит найти маркер, который остается неизменным даже после обновлений игры, пока данные не будут удалены или их назначение не изменится. Позвольте мне объяснить, почему это важно.

## Определение назначения (Deducing Purpose)

До сих пор я показывал вам, как слепо сканировать память на предмет конкретных данных, не принимая во внимание, как эти данные используются. Этот метод может быть эффективным, но он не всегда является наиболее удобным. Во многих случаях гораздо проще определить назначение данных, выяснить, какой код может использовать эти данные, а затем через этот код найти в памяти адрес самих данных.

Это может не показаться простым, но таким же непростым является и метод «сканировать память игры на наличие определенного значения определенного типа данных, а затем непрерывно фильтровать список результатов на основе изменяющихся критериев», который вы изучили до сих пор. Итак, давайте рассмотрим, как мы можем найти адрес здоровья по его назначению.

Рассмотрим код в Листинге 5-1.

---

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

*Листинг 5-1: Структура, содержащая параметры игрока, и функция, которая их отображает*

Если представить, что `printString()` — это встроенная игровая функция для вывода текста, то этот код очень похож на то, что можно найти в игре. Структура `PlayerVital` имеет два свойства: текущее значение и максимальное значение. Значение `health` является структурой `PlayerVital`, поэтому оно тоже содержит эти свойства. Основываясь только на названии, можно сделать вывод, что `health` хранит данные о здоровье игрока. Кроме того, можно понять, что `printString()` использует эти данные, чтобы отображать информацию о здоровье.

Даже без кода можно логически прийти к такому же выводу, просто глядя на текст здоровья в интерфейсе игры — ведь сам компьютер не может делать что-либо без кода. Помимо самой переменной здоровья, существует несколько элементов кода, которые необходимы для отображения текста игроку. Во-первых, должна быть функция, которая отображает текст. Во-вторых, строки `Health` и другие связанные данные должны находиться рядом.

**NOTE** *Почему я предполагаю, что текст разделен на две отдельные строки, а не одну? Игровой интерфейс показывает, что текущее значение здоровья отображается между этими двумя строками, но существуют и другие способы, как это могло быть реализовано, включая форматированные строки, `strcat()`, или даже несколько вызовов функций отображения текста. При анализе данных лучше держать свои предположения широкими, чтобы учитывать все возможные варианты.*

Чтобы найти `Health` без использования сканера памяти, мы можем использовать эти две разные строки. Однако мы, скорее всего, не будем иметь представления о том, как выглядит функция отображения текста, где она находится и сколько раз вызывается. Реально, струны — это все, что мы знаем, чтобы искать, и этого достаточно. Давайте пройдемся по нему.

Найти здоровье без использования сканера памяти можно, используя эти две отдельные строки. Мы, вероятно, не имеем представления о том, как выглядит функция, которая отображает текст, где она находится или сколько раз вызывается. На практике единственное, что нам известно — это строки, которые мы должны искать, и этого достаточно. Давайте разберем этот процесс.

## Поиск здоровья игрока с помощью OllyDbg

Я проведу вас через процесс отслеживания структуры здоровья в этом разделе, но также включил бинарный файл, который анализирую, в ресурсы книги. Чтобы следовать за мной и получить практический опыт, используйте файл Chapter5\_AdvancedMemoryForensics\_Scanning.exe.

Сначала откройте OllyDbg и прикрепите его к исполняемому файлу. Затем откройте окно Executable modules в OllyDbg и дважды щелкните по основному модулю; в моем примере главный модуль — единственный .exe в окне модулей.

Окно CPU должно появиться. Теперь щелкните правой кнопкой мыши в панели Disassembler и выберите Search for □ All referenced text strings. Это должно открыть окно References, показанное на рисунке 5-1.

Address	Disassembly	Text string
00401000	PUSH EBP	(Initial CPU selection)
00401010	PUSH Chapter5.0044	ASCII "Health: %d of %d"
0040101E	PUSH Chapter5.0040	ASCII "pause"
00401065	PUSH OFFSET Chapter5._	ASCII ";#"
0040170A	PUSH Chapter5._se	ASCII ";"

Рисунок 5-1: Окно ссылок OllyDbg, показывающее только список строк. В реальной игре их было бы гораздо больше, чем четыре.

Из этого окна щелкните правой кнопкой мыши и выберите Search for text. Появится диалог поиска. Введите строку, которую вы ищете, как показано на рисунке 5-2, и сделайте поиск как можно более широким, отключив Case sensitive и включив Entire scope.

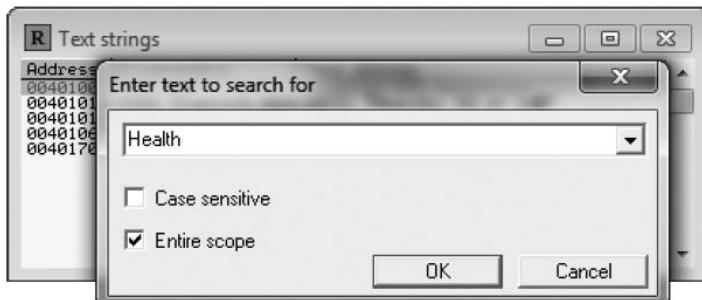


Рисунок 5-2: Поиск строк в OllyDbg

Нажмите OK, чтобы выполнить поиск. Окно ссылок (References) снова станет активным, а первое совпадение будет выделено. Дважды щелкните по совпадению, чтобы увидеть ассемблерный код, использующий строку, в окне CPU. Панель Disassembler сфокусируется на строке кода по адресу 0x401030, которая передает строковый параметр в printString(). Вы можете увидеть эту строку на рисунке 5-3, где я выделил весь блок вызова функции.

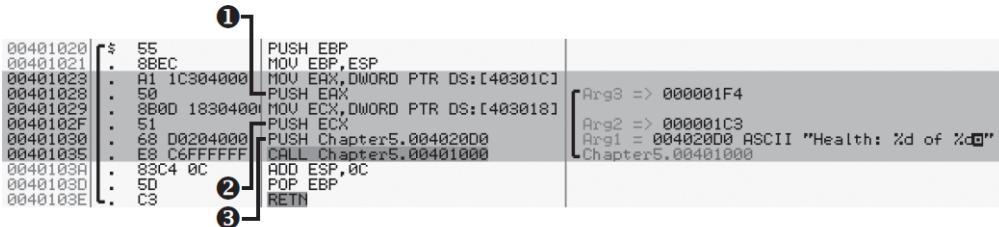


Рисунок 5-3: Просмотр printString() вызов в панели дизассемблера окна CPU

Читая ассемблерный код, можно получить очень точное представление о том, что именно делает игра. Чёрная скобка слева показывает, что строка Health находится внутри вызова функции. Обратите внимание на аргументы этой функции. В порядке следования это EAX ①, ECX ② и строка формата по адресу 0x4020D0 ③.

EAX — это значение по адресу 0x40301C, ECX — это значение по адресу 0x403018, а строка формата содержит Health. Поскольку строка содержит два плейсхолдера формата, можно предположить, что оставшиеся два параметра являются аргументами для этих плейсхолдеров.

Зная, какие аргументы передаются и что они помещаются в стек в обратном порядке, можно проследить процесс обратно и заключить, что исходный код выглядел примерно так, как в Листинге 5-2.

---

```

int currentHealth; // value at 0x403018
int maxHealth;    // value at 0x40301C
--snip--
someFunction("Health: %d of %d\n",
              currentHealth, maxHealth);

```

---

Листинг 5-2: Как игровой хакер может интерпретировать сборку, скомпилированную на рисунке 5-3

Значения, хранящиеся в EAX и ECX, находятся рядом в памяти, что может означать, что они являются частью структуры. Однако, чтобы упростить пример, здесь они просто представлены как определения переменных. В любом случае, это два числа, используемых для отображения здоровья игрока.

Поскольку оба этих важных значения отображаются в

пользовательском интерфейсе игры, было легко сделать предположения о коде, который отвечает за их вывод. Если вы знаете назначение данных, можно быстро найти код, который их использует; в данном случае это знание позволило нам быстро найти оба адреса.

Во многих случаях поиск адресов может быть таким же простым, но некоторые данные имеют настолько сложное назначение, что сложно догадаться, что именно искать. Например, поиск данных карты или местоположений персонажей в OllyDbg может быть довольно сложной задачей.

Строки далеко не единственные маркеры, с помощью которых можно найти данные, которые нужно изменить в игре, но они, безусловно, самые простые для объяснения без сложных примеров. Более того, некоторые игры встраивают в свой код строки логов или сообщений об ошибках, и изучение окна Referenced text strings в OllyDbg может быть быстрым способом определить, присутствуют ли такие строки. Если вы освоите логику работы логов в игре, то сможете находить значения еще быстрее.

## Определение новых адресов после обновлений игры

Когда код приложения изменяется и компилируется заново, создается новый бинарный файл, отражающий эти изменения. Этот бинарный файл может быть очень похож на предыдущий, или же бинарники могут не иметь ничего общего; разница между двумя версиями напрямую связана со сложностью внесенных изменений.

Небольшие изменения, такие как модификация строк или обновленные константы, могут оставить бинарники почти идентичными и часто не оказывают никакого влияния на адреса кода или данных. Однако более сложные изменения — такие как добавление нового пользовательского интерфейса, переработка внутренней структуры или новый игровой контент — часто вызывают сдвиги в расположении важной памяти.

### АВТОМАТИЧЕСКИЙ ПОИСК CURRENTHEALTH И MAXHEALTH

В разделах "Поиск шаблонов в ассемблере" на странице 19 и "Поиск строк" на странице 21 я показал несколько Lua-скриптов для Cheat Engine и объяснил, как они работают. Используя функцию `findString()` в этих примерах, можно заставить Cheat Engine автоматически находить адрес строки формата, который мы только что нашли вручную в OllyDbg.

Затем можно написать небольшую функцию, которая будет сканировать этот адрес, находя байт **0x68** (байт инструкции PUSH), как видно рядом с **0x401030** (на рисунке 5-3), чтобы определить адрес кода, который загружает значение в стек. После этого можно считать **4 байта** из `pushAddress` — 5 и `pushAddress` — 12, чтобы автоматически находить `currentHealth` и `maxHealth`.

Это может не показаться полезным, так как мы уже нашли адреса вручную, но если бы это была реальная игра, эти адреса изменились бы после обновлений. Использование этого знания для автоматического поиска новых адресов может быть очень полезным. Если готовы к вызову, попробуйте!

Из-за постоянных исправлений ошибок, улучшений контента и добавления новых функций, онлайн-игры являются одним из наиболее быстро развивающихся видов программного обеспечения. Некоторые игры выпускают обновления раз в неделю, и хакеры, создающие ботов, проводят большую часть своего времени, анализируя новые бинарные файлы, чтобы соответствующим образом обновлять своих ботов.

Если вы создадите продвинутых ботов, они со временем будут все больше зависеть от базы данных с адресами памяти. Когда выходит обновление, определение новых адресов для множества значений и функций становится самой затратной по времени задачей, с которой вам неизбежно придется столкнуться. Полагаться на советы из "Советы по победе в гонке обновлений" может быть полезно, но сами советы не помогут вам найти обновленные адреса. Можно автоматически находить некоторые адреса с помощью скриптов Cheat Engine, но этот метод тоже не всегда работает. Иногда вам придется делать всю грязную работу вручную.

Если вы попытаетесь снова искать адреса тем же методом, что и вначале, то просто потратите время впустую. Однако у вас есть большое преимущество: старый бинарный файл и найденные ранее адреса. Используя эти два элемента, можно быстро найти все адреса, которые нужно обновить, в разы быстрее.

Рисунок 5-4 показывает два разных дизассемблированных файла: новый бинарный файл игры слева и предыдущую версию справа. Этот снимок взят из реальной игры (название которой останется неназванным), чтобы предоставить вам реалистичный пример.

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]
0047B656	. 64:A3 000000	MOV DWORD PTR FS:[0],EAX	0047B526	. 64:A3 000000	MOV DWORD PTR FS:[0],EAX
0047B657	. 8965 F0	MOV DWORD PTR SS:[EBP-10],ESP	0047B527	. 8965 F0	MOV DWORD PTR SS:[EBP-10],ESP
0047B658	. C785 E0F0FFF	MOV DWORD PTR SS:[EBP-220],-1	0047B528	. C785 E0F0FFF	MOV DWORD PTR SS:[EBP-220],-1
0047B659	. C745 FC 000000	MOV DWORD PTR SS:[EBP-4],0	0047B529	. C745 FC 000000	MOV DWORD PTR SS:[EBP-4],0
0047B672	> E8 29800000	CALL QWORD PTR SS:[EBP-4040]	0047B542	> E8 09800000	CALL QWORD PTR SS:[EBP-40533F20]
0047B677	. 8BF8	MOV EDI,EAX	0047B547	. 8BF8	MOV EDI,EAX
0047B679	. 89BD E0F0FFF	MOV DWORD PTR SS:[EBP-220],EDI	0047B549	. 89BD E0F0FFF	MOV DWORD PTR SS:[EBP-220],EDI
0047B67F	. 83FF FF	CMP EDI,-1	0047B54F	. 83FF FF	CMP EDI,-1
0047B682	.~ 75 1F	JNZ SHORT 0047B6A3	0047B552	.~ 75 1F	JNZ SHORT 0047B573

Рисунок 5-4: Разборки двух версий одной игры бок о бок

Мой бот изменил код по адресу 0x047B542 (справа), и мне нужно было найти соответствующий код в новой версии, который я обнаружил по адресу 0x047B672 (слева). Эта функция вызывает функцию разбора пакетов при получении пакета. Чтобы изначально найти этот адрес (а под «изначально» я имею в виду около 100 обновлений назад), я выяснил, как работает сетевой протокол игры, установил точки останова на множестве API-вызовов, связанных с сетью, пошагово выполнил код и проанализировал данные в стеке, пока не нашел что-то похожее на ожидаемый результат, основываясь на знании протокола.

## СОВЕТЫ ПО ПОБЕДЕ В ГОНКЕ ОБНОВЛЕНИЙ

На насыщенных рынках первым выпустить стабильное обновление — критически важно. Гонка начинается сразу после выхода обновления игры, и хакеры, претендующие на звание самых быстрых, тратят сотни часов на подготовку. Вот основные способы оставаться в топе:

- **Создавайте тревоги обновлений** Написав программное оповещение, которое предупредит вас о выходе обновления игры, можно начать работу над обновлениями как можно раньше.
- **Автоматизируйте установку ботов** Игры часто запланированно выпускают обновления в часы, когда онлайн наименьший. Боты не навидят просыпаться и загружать новое ПО перед работой, но им нравится просыпаться и видеть, что все установлено автоматически, пока игра работает.
- **Используйте меньше адресов** Чем меньше адресов нужно обновлять, тем лучше. Объединение связанных структур данных и удаление ненужного обращения к памяти экономит массу времени.
- **Имейте резервные копии данных** Данные меняются, а хакеры совершают ошибки. Возможность быстро проверить ключевые функции может стать разницей между ботом, который работает после обновления, и ботом, который перестает работать. Некоторые пользователи оставляют отзывы, которые помогают заметить изменения в игре.

Использование этих методов даст вам значительное преимущество, но не всегда приведет к победе. Прежде всего, полезно понимать, как работает **reverse engineering**, и максимально использовать это знание в свою пользу.

Я мог бы повторять тот же процесс для всех 100+ обновлений с тех пор, но это было бы ненужным. Код оставался практически неизменным в течение многих лет, что позволило мне использовать шаблоны из старого кода, чтобы найти адрес вызова функции в новой версии.

Теперь рассмотрим этот фрагмент кода на ассемблере:

```
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
```

Выглядит знакомо? Сравните с Рисунком 5-4, и вы увидите, что этот точный код существует внутри выделенной функциональной части в обеих версиях игры. Независимо от того, что он делает, комбинация операций выглядит довольно уникально; из-за количества различных смещений, которые использует код относительно ЕВР, маловероятно, что идентичный код существует где-либо еще в бинарном файле.

Каждый раз, когда мне нужно обновить этот адрес, я открываю старый бинарный файл в OllyDbg, выделяю этот фрагмент кода, кликаю правой кнопкой мыши и выбираю Asm2Clipboard > Copy fixed asm to clipboard. Затем я открываю новый бинарный файл в OllyDbg, перехожу в окно CPU Window, нажимаю Ctrl+S, вставляю ассемблерный код и запускаю поиск. В 9.5 случаях из 10 это помешает меня прямо над той функцией вызова, которую мне нужно найти в новой версии. Когда выходит обновление, можно использовать этот же метод, чтобы найти все новые адреса функций. Это должно работать для любого кода, который можно найти в ассемблерных файлах. Однако есть несколько нюансов:

OllyDbg ограничивает поиск восьми операциями, поэтому вам

нужно находить маркеры, состоящие из пяти и более. Операции, которые вы используете, не должны содержать ссылки на другие адреса, так как они могли измениться. Если части игры были изменены так, что код, который вы ищете, больше не существует, поиск ничего не найдет. Если разработчики игры изменили оптимизацию или настройки компилятора, расположение кода может значительно отличаться. Как объяснялось в разделе “Automatically Find currentHealth and maxHealth” на странице 102, можно автоматизировать поиск с помощью Cheat Engine Lua-скриптов, которые выполняют эти задачи за вас. Опытные хакеры делают всё возможное, чтобы автоматически находить как можно больше измененных адресов, а лучшие из них способны определить новые адреса почти сразу после выхода обновления. Это может быть много работы, но если делать это правильно, результат того стоит!

## Выявление сложных структур в игровых данных

Глава 4 описала, как игра может хранить данные в статических структурах. Это знание будет достаточным, когда вы пытаетесь найти простые данные, но оно не подходит для поиска данных, которые хранятся с помощью динамических структур. Это связано с тем, что динамические структуры могут быть разбросаны по разным областям памяти, содержать длинные указательные цепочки или требовать сложных алгоритмов для извлечения данных.

Этот раздел исследует общие динамические структуры, которые можно встретить в игровом коде, и объяснит, как читать данные из памяти после их обнаружения. В начале я расскажу об основном составе каждой динамической структуры. Затем я объясню алгоритмы, необходимые для чтения данных из таких структур. (Для простоты в обсуждении каждого алгоритма предполагается, что у вас есть указатель на экземпляр структуры, а также способ чтения памяти.)

В конце я поделюсь полезными советами и хитростями, которые помогут вам определить, заключены ли нужные вам данные в одну из этих структур. Это позволит вам понимать, когда применять полученные знания. Я сосредоточусь на C++, поскольку его объектно-ориентированная природа и стандартная библиотека широко используются для работы с такими структурами.

### НОТЕ

*Некоторые из этих структур могут немного отличаться от машины к машине из-за компиляторов, настроек оптимизации или различных реализаций стандартной библиотеки, но основные принципы останутся неизменными. Кроме того, для краткости я буду опускать несущественные части этих структур, такие как кастомные аллокаторы или функции сравнения. Примеры кода можно найти здесь: <https://www.nostarch.com/gamemhacking/> в ресурсах к Главе 5.*

## The std::string Class

Экземпляры std::string являются одними из самых частых источников динамического хранения данных. Этот класс из C++ Standard Template Library (STL) абстрагирует операции со строками от разработчика, при этом сохраняя эффективность. Благодаря этому std::string широко используется во всех типах программного обеспечения. Например, видеоигра может использовать std::string для хранения строковых данных, таких как имена существ или персонажей.

### Изучение структуры строки std::string

Если убрать все методы-члены и другие некритичные компоненты std::string, останется следующая структура:

```
class string {
    union {
        char* dataP; // Указатель на данные
        char dataA[16]; // Массив данных размером 16
    };
    int length; // Длина строки
};

// Указать на строку в памяти
string* _str = (string*)stringAddress;
```

Класс резервирует 16 символов, которые, предположительно, используются для хранения строки на месте. Он также объявляет, что первые 4 байта могут быть указателем на символ. Это может показаться странным, но на самом деле это результат оптимизации. В какой-то момент разработчики этого класса решили, что 15 символов (плюс нуль-терминатор) — это подходящая длина для многих строк, и таким образом можно сэкономить на выделении памяти и её освобождении, сохраняя 16 байтов непосредственно в объекте. Однако, чтобы поддерживать более длинные строки, они позволили первым 4 байтам этого зарезервированного массива использоваться в качестве указателя на символы более длинных строк.

**Н О Т Е** Если код скомпилирован для 64-битной архитектуры, то первыми 8 байтами (а не 4) будет указатель на символ. Однако в этом примере можно считать, что используются 32-битные адреса и что int занимает 4 байта.

Доступ к строковым данным таким способом приводит к некоторым накладным расходам. Функция для определения правильного буфера данных может выглядеть так:

---

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

---

Фактически, объект `std::string` может быть либо полностью строкой, либо указателем на более длинную строку, что делает эту структуру довольно сложной с точки зрения взлома игр. Некоторые игры могут использовать `std::string` только для коротких строк, не превышающих 15 символов. В этом случае вы можете внедрять боты, которые полагаются на эти строки, даже не зная, что в реальности эта структура гораздо сложнее, чем простая строка.

## Непонимание структуры `std::string` может испортить вам удовольствие

Непонимание истинной природы структуры, содержащей нужные вам данные, может привести к созданию бота, который работает лишь время от времени и подводит в критические моменты. Представьте, например, что вы пытаетесь выяснить, как игра хранит данные о существах. В ходе гипотетического поиска вы обнаруживаете, что все существа в игре хранятся в массиве структур, которые выглядят примерно так, как показано в Листинге 5-3.

---

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

    int creatureType;
};
```

---

*Листинг 5-3: Как вы можете интерпретировать данные о существах, найденные в памяти*

После сканирования данных о существах в памяти, допустим, вы замечаете, что первые 4 байта каждой структуры уникальны для каждого существа, поэтому называете эти байты `uniqueID` и предполагаете, что они представляют собой свойство идентификации. Дальше по памяти вы обнаруживаете, что имя существа хранится сразу после `uniqueID`, и после некоторого анализа делаете вывод, что его длина составляет 16 байтов. Следующее значение, которое вы видите в памяти, оказывается `nameLength`; это немного странное значение, так как строка с

завершающим нулём уже подразумевает длину, но вы игнорируете это несоответствие и продолжаете анализировать данные в памяти. После дальнейшего анализа вы определяете, для чего предназначены оставшиеся значения, определяете структуру, показанную в Листинге 5-3, и пишете бота, который автоматически атакует существ с определёнными именами.

После недель тестирования вашего бота в охоте на существ с именами, такими как Dragon, Cyclops, Giant и Hound, вы решаете впервые продемонстрировать его друзьям. Для первого сражения вы собираете всех, чтобы убить босса по имени Super Bossman Supreme. Вся команда настраивает бота так, чтобы он сначала атаковал босса, а затем переключался на более слабых существ, таких как Demon или Grim Reaper, когда босс выходит за пределы зоны поражения.

Когда ваша команда достигает подземелья босса... вас всех медленно истребляют.

Что пошло не так в этом сценарии? Ваша игра, скорее всего, хранит имена существ с помощью `std::string`, а не просто массива символов. Поля `name` и `nameLength` в `creatureInfo` на самом деле представляют собой поле `std::string`, в котором хранятся указатель на данные и сами данные. Имя Super Bossman Supreme длиннее 15 символов, а поскольку бот не был осведомлён об особенностях реализации `std::string`, он не смог его распознать. Вместо этого бот автоматически нацеливался на ближайших существ с именем Demon, эффективно мешая вам атаковать босса, который медленно истощал ваше здоровье и запасы.

## Определение, хранятся ли данные в `std::string`

Не зная, как устроен класс `std::string`, вам было бы трудно отслеживать ошибки, подобные той, что была описана в гипотетическом примере. Но, сочетая полученные здесь знания с опытом, можно полностью избежать таких багов. Когда вы находите в памяти строку, например `name`, не стоит сразу предполагать, что она хранится в простом массиве. Чтобы определить, является ли строка объектом `std::string`, задайте себе следующие вопросы:

Почему у строки есть поле длины, если она уже содержит завершающий нуль? Если вы не можете придумать вескую причину, скорее всего, перед вами `std::string`.

Есть ли у некоторых существ (или других игровых элементов, в зависимости от того, что вы ищете) имена длиннее 16 символов, но в памяти выделено место только на 16 символов? Если да, то данные почти наверняка хранятся в `std::string`.

Хранится ли имя непосредственно в памяти, требуя от разработчика использования `strcpy()` для его изменения? Скорее всего, это `std::string`, так как работа с "сырыми" С-строками таким образом считается плохой практикой.

Наконец, не забывайте, что также существует класс `std::wstring`, используемый для хранения широких строк. Его реализация очень

похожа, но вместо `char` используется `wchar_t`.

## Класс `std::vector`

Игры должны отслеживать множество динамических массивов данных, но управление массивами переменного размера может быть весьма сложной задачей. Для обеспечения скорости и гибкости разработчики игр часто хранят такие данные, используя шаблонный класс STL под названием `std::vector` вместо простого массива.

Исследование структуры `std::vector`

Объявление этого класса выглядит примерно так, как показано в Листинге 5-4.

---

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

---

Листинг 5-4: Абстрагированный `std::vector` объект

Этот шаблон добавляет дополнительный уровень абстракции, поэтому я продолжу описание, используя `std::vector`, объявленный с типом `DWORD`. Вот как игра может объявить такой вектор:

---

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

---

Теперь давайте разберем, как `std::vector` объектов `DWORD` может выглядеть в памяти. Если бы у вас был адрес `_vec` и он находился в той же области памяти, можно было бы воссоздать базовую структуру класса и получить доступ к `_vec`, как показано в Листинге 5-5.

```
class vector {
    DWORD* begin;
    DWORD* end;
    DWORD* tail;
};

// Указатель на вектор в памяти
vector* _vec = (vector*)vectorAddress;
```

Листинг 5-5: A `DWORD std::vector` object

Можно рассматривать член `begin` как обычный массив, так как он указывает на первый элемент в объекте `std::vector`. Однако в

нем нет члена, содержащего длину массива.

Поэтому длину вектора необходимо вычислять на основе `begin` и `end`, где `end` указывает на пустой объект, следующий за последним элементом в массиве. Код для вычисления длины выглядит так:

---

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

Эта функция просто вычитает адрес, хранящийся в `begin`, из адреса, хранящегося в `end`, чтобы определить количество байтов между ними. Затем, чтобы вычислить количество объектов, оно делит число байтов на размер одного объекта.

Используя `begin` и эту функцию `length()`, можно безопасно получать доступ к элементам вектора `_vec`. Код будет выглядеть следующим образом:

---

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

Данный код получает элемент из вектора по индексу. Однако, если индекс больше длины вектора, будет выброшено исключение `std::out_of_range`. Добавление элементов в `std::vector` было бы очень дорогой операцией, если бы класс не мог резервировать или повторно использовать память. Чтобы решить эту проблему, в классе реализуется функция `reserve()`, которая сообщает вектору, сколько объектов следует зарезервировать.

Абсолютный размер `std::vector` (его `capacity`) определяется через дополнительный указатель, называемый `tail`, в воссозданном нами классе вектора. Вычисление емкости (`capacity`) похоже на вычисление длины:

---

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

Чтобы определить емкость `std::vector`, вместо вычитания `begin` из `end`, как при вычислении длины, эта функция вычитает `begin` из `tail`. Дополнительно это вычисление можно использовать в третьей функции, чтобы определить количество свободных элементов в векторе, используя `tail` и `end`.

---

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

Учитывая правильные функции чтения и записи памяти, можно использовать объявление в Листинге 5-4 и последующие вычисления для доступа и манипулирования векторами в памяти игры. Глава 6 подробно рассматривает чтение памяти, но пока

давайте разберёмся, как определить, хранится ли интересующие вас данные в std::vector.

## Определение хранения данных в std::vector

Как только вы нашли массив данных в памяти игры, есть несколько шагов, которые можно выполнить, чтобы определить, хранится ли он в std::vector. Во-первых, можно быть уверенным, что массив не является std::vector, если у него есть статический адрес, так как объекты std::vector требуют указателей для доступа к базовому массиву. Если массив требует пути указателей, завершающее смещение, равное 0, укажет на std::vector. Чтобы подтвердить это, можно изменить завершающее смещение на 4 и проверить, указывает ли оно на последний объект в массиве, а не на первый. Если так, то это почти наверняка std::vector, поскольку вы только что подтвердили указатели начала и конца.

## Класс std::list

Аналогично std::vector, std::list — это класс, который можно использовать для хранения коллекции элементов в связанным списке. Основные отличия заключаются в том, что std::list не требует непрерывного пространства памяти для элементов, не может напрямую получать доступ к элементам по индексу и может увеличиваться в размерах без воздействия на предыдущие элементы. Из-за накладных расходов, необходимых для доступа к элементам, этот класс редко встречается в играх, но он используется в некоторых особых случаях, которые будут рассмотрены в этом разделе.

## Исследование структуры std::list

Класс std::list выглядит примерно как в Листинге 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;
};
```

---

Листинг 5-6: Абстрагированный std::list объект

Здесь два класса: listItem и list. Чтобы избежать излишней абстракции при объяснении работы std::list, этот объект будет

описан так, как он выглядел бы при использовании DWORD. Вот как игра могла бы объявить std::list с DWORD:

---

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

---

Учитывая такое объявление, std::list будет иметь структуру, как в Листинге 5-7.

---

```
class listItem {
    listItem* next;
    listItem* prev;
    DWORD value;
};

class list {
    listItem* root;
    int size;
};

// point to a list
list* _lst = (list*)listAddress;
```

---

Листинг 5-7: A DWORD std::list object

Класс list представляет заголовок списка, а listItem представляет значения, хранящиеся в списке. Вместо хранения элементов последовательно, элементы списка хранятся независимо. Каждый элемент содержит указатель на следующий элемент (next) и на предыдущий элемент (prev). Эти указатели используются для поиска элементов в списке. Корневой элемент (root) действует как маркер конца списка; указатель next последнего элемента указывает на root, как и указатель prev первого элемента. Указатели next и prev элемента root также указывают на первый и последний элементы соответственно. На схеме на Рисунке 5-5 показано, как это выглядит.

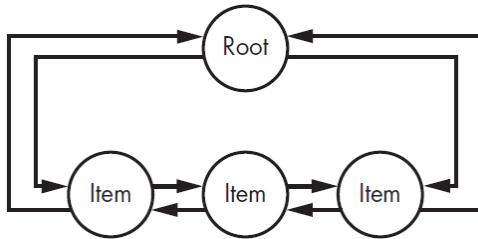


Рисунок 5-5: Схема std::list

(Графическое представление двусвязного списка с узлом Root, связанным с элементами Item.)

Учитывая такую структуру, следующий код можно использовать для итерации по объекту std::list:

```

// Итерация вперёд
listItem* it = _lst->root->next;
while (it != _lst->root) {
    std::cout << "Значение: " << it->value << std::endl;
    it = it->next;
}

// Итерация назад
it = _lst->root->prev;
while (it != _lst->root) {
    std::cout << "Значение: " << it->value << std::endl;
    it = it->prev;
}

```

Первый цикл начинается с первого элемента (`root->next`) и выполняет итерации вперёд (`it = it->next`), пока не дойдёт до конечного маркера (`root`). Второй цикл начинается с последнего элемента (`root->prev`) и выполняет итерации назад (`it = it->prev`), пока не дойдёт до конечного маркера (`root`). Эта итерация зависит от `next` и `prev`, потому что, в отличие от объектов в массиве, объекты в `std::list` не являются непрерывными в памяти. Поскольку память каждого объекта в `std::list` не является непрерывной, нет простого и быстрого способа вычислить размер. Вместо этого класс просто определяет член `size`.

Кроме того, концепция резервирования памяти для новых объектов не имеет значения для списков, поэтому нет переменной или вычисления, определяющего вместимость (`capacity`) списка.

## Определение, хранится ли игровая информация в `std::list`

Определить, находятся ли объекты в классе `std::list`, может быть непросто, но есть несколько подсказок, на которые можно обратить внимание. Во-первых, элементы в `std::list` не могут иметь статические адреса, поэтому, если искомые данные имеют статический адрес, то можно быть уверенным, что они не в `std::list`. Однако элементы, которые явно являются частью коллекции, могут быть частью `std::list`, если они не являются смежными в памяти.

Также стоит учитывать, что объекты в `std::list` могут образовывать бесконечно длинные цепочки указателей (например, `it->prev->next->prev->next->prev...`), а сканирование указателей в Cheat Engine покажет гораздо больше результатов, если параметр `No Looping Pointers` выключен.

Вы также можете использовать скрипт, чтобы определить, хранится ли значение в связном списке. В Листинге 5-8 показан скрипт для Cheat Engine, который делает именно это.

---

```
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
```

---

Листинг 5-8: Определение того, находятся ли данные в `std::list` с помощью Lua-скрипта Cheat Engine

Здесь довольно много кода, но на самом деле его работа довольно проста. Функция `isValueInLinkedList()` принимает адрес некоторого значения, затем просматривает назад до 40 байтов (10 целочисленных объектов, на случай, если значение находится в какой-то более крупной структуре), начиная с 8 байтов выше указанного адреса (так как должны присутствовать два указателя, каждый размером 4 байта). Из-за выравнивания памяти этот цикл выполняется с шагом 4 байта.

На каждой итерации адрес передаётся в функцию `_verifyLinkedList()`, где и происходит «магия». Если рассматривать это в терминах структуры связного списка, как она определена в этой главе, функция просто выполняет следующее:

---

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

---

Функция в основном предполагает, что переданный ей адрес памяти указывает на узел связного списка, и проверяет, имеют ли предполагаемый узел корректные указатели на предыдущий и следующий узлы. Если узлы действительны, значит, предположение было верным, и адрес действительно принадлежит узлу связного списка. Если же узлы отсутствуют или не указывают на правильные места, предположение было ошибочным, и адрес не является частью связного списка.

Следует помнить, что этот скрипт не предоставит вам адрес корневого узла списка, а лишь адрес узла, содержащего указанное вами значение. Чтобы правильно обходить связный список, вам нужно будет сканировать память в поисках корректного указателя на корневой узел, так что вам потребуется его адрес.

Нахождение этого адреса может потребовать анализа дампов памяти, множества проб и ошибок и немалой головной боли, но это вполне возможно. Лучший способ начать — следовать цепочке предыдущих и следующих узлов, пока не встретится узел с пустыми, бессмысленными данными или значением 0xBAADFOOD (некоторые, но не все, стандартные реализации библиотеки используют это значение для маркировки корневых узлов).

Это исследование также может быть упрощено, если точно знать, сколько узлов находится в списке. Даже без заголовка списка можно определить количество узлов, непрерывно следя по указателям на следующий элемент, пока не окажетесь обратно в начальном узле, как показано в Листинге 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
```

Листинг 5-9: Определение размера произвольного std::list с помощью Lua-скрипта Cheat Engine

Сначала эта функция создаёт счётчик для хранения количества узлов и переменную для хранения адреса следующего узла. Затем цикл while перебирает узлы, пока не вернётся к начальному узлу. В конце функция возвращает значение счётчика, который увеличивался при каждой итерации цикла.

#### НАЙТИ КОРНЕВОЙ УЗЕЛ С ПОМОЩЬЮ СКРИПТА

На самом деле можно написать скрипт, который найдёт корневой узел, но я оставлю это в качестве дополнительного упражнения для вас. Как это работает? Корневой узел должен находиться в цепочке узлов, заголовок списка указывает на корень, а размер списка в памяти сразу следует за корнем. Зная это, можно написать скрипт, который будет искать любую область памяти, содержащую указатель на один из узлов списка, а затем проверять, есть ли за ним размер списка. Чаще всего эта область памяти является заголовком списка, а узел, на который она указывает, — корневым узлом.

## Класс std::map

Как и std::list, std::map использует связи между элементами для формирования своей структуры. Уникальной особенностью std::map является то, что каждый элемент хранит два значения данных (ключ и значение), а сортировка элементов — это встроенное свойство базовой структуры данных: красно-чёрного дерева. Следующий код показывает структуры, составляющие 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;
}
```

---

Красно-чёрное дерево — это самобалансирующееся бинарное дерево поиска, и std::map тоже является таким деревом. В стандартной реализации std::map каждый элемент (или узел) в дереве имеет три указателя: left, parent и right.

Кроме указателей, каждый узел также хранит ключ (key) и значение (value). Узлы упорядочены в дереве на основе сравнения их ключей:

Левый указатель указывает на узел с меньшим ключом.

Правый указатель указывает на узел с большим ключом.

Родительский указатель (parent) указывает на верхний узел.

Первый узел в дереве называется rootNode.

Узлы, у которых нет потомков (left и right равны nullptr), указывают на него.

## Визуализация карты std::map

Рисунок 5-6 показывает std::map, который содержит ключи 1, 6, 8, 11, 13, 15, 17, 22, 25 и 27.

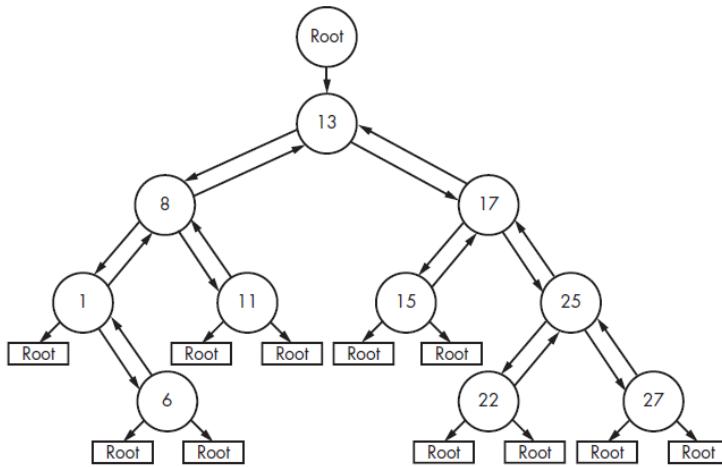


Рисунок 5-6: Красно-черное дерево

Верхний узел (содержащий значение 13) указывает на родителя rootNode. Все, что находится слева от него, имеет меньший ключ, а все, что справа, имеет больший ключ. Это справедливо для любого узла в дереве, и этот принцип обеспечивает эффективный поиск по ключам. Хотя это не представлено на изображении, левый указатель корневого узла указывает на самый левый узел (1), а правый указатель указывает на самый правый узел (27).

## Доступ к данным в std::map

Еще раз, я буду использовать статическое определение std::map при обсуждении того, как извлекать данные из структуры. Поскольку шаблон принимает два типа, я также буду использовать некоторые псевдотипы, чтобы сохранить понятность. Вот объявление объекта std::map, к которому я буду обращаться в оставшейся части раздела:

---

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

---

С этим объявлением структура myMap становится:

---

```
struct mapItem {
    mapItem* left;
    mapItem* parent;
    mapItem* right;
    keyInt key;
    valInt value;
};

struct map {
    DWORD irrelevant;
    mapItem* rootNode;
    int size;
}

map* _map = (map*)mapAddress;
```

---

Существует несколько важных алгоритмов, которые могут понадобиться для доступа к данным в структуре std::map в игре. Во-первых, простая итерация по каждому элементу карты может быть полезной, если вам просто нужно увидеть все данные. Для выполнения этого процесса последовательно можно написать такую функцию итерации:

---

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

---

Функция для обхода всей карты сначала читает текущий узел и проверяет, является ли он rootNode. Если нет, она рекурсивно вызывает себя для левого узла, затем печатает узел и рекурсивно вызывает себя для правого узла.

Чтобы вызвать эту функцию, необходимо передать указатель на rootNode, например:

---

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

---

Цель std::map — хранить данные с ключами, обеспечивая быстрый доступ. Когда необходимо найти узел с определенным ключом, лучше использовать внутренний алгоритм поиска, а не сканировать всё дерево. Код для поиска в std::map выглядит следующим образом:

---

```
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;
}
```

---

Начав с вершины дерева, функция рекурсивно уходит влево, если текущий ключ больше искомого, и вправо, если он меньше. Если ключи совпадают, возвращается текущий узел. Если поиск доходит до нижнего уровня дерева и не находит ключ, возвращается NULL, так как ключа нет в карте.

Пример использования findItem():

---

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

---

Пока findItem() не возвращает NULL, этот код должен выводить узел из \_map.

## Определение хранения игровых данных в std::map

Обычно я даже не рассматриваю возможность того, что данные могут находиться в std::map, пока не убедюсь, что коллекция — это не массив, std::vector или std::list. Если все три варианта исключены, то, как и в std::list, можно проверить три целочисленных значения перед значением и посмотреть, указывают ли они на память, которая может содержать другие узлы карты.

Это также можно сделать с помощью Lua-скрипта в Cheat Engine. Скрипт похож на тот, что я показывал для списков: он проходит назад по памяти, чтобы проверить, найдена ли допустимая структура узла перед значением. Однако в отличие от кода для списка, функция, проверяющая узел, гораздо сложнее. Посмотрите код в Листинге 5-10, а затем я его разберу.

---

```
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
```

---

*Листинг 5-10: Определение того, находятся ли данные в std::map с помощью Lua-скрипта Cheat Engine*

Данная функция, получив address, проверяет, находится ли address в структуре map. Сначала она проверяет, есть ли у него допустимый родительский узел, и, если да, проверяет, указывает ли родительский узел на address с любой из сторон ❶. Но этой проверки недостаточно. Если проверка пройдена, функция также поднимается по линии родительских узлов, пока не достигнет узла, который является родителем своего собственного родителя ❷, выполняя до 200 итераций перед завершением. Если в ходе подъёма обнаруживается узел, который сам является своим собственным дедушкой, то address определённо указывает на узел map. Это работает, потому что, как я описал в разделе «Визуализация std::map» на странице 114, на вершине каждой map находится корневой узел, чей родитель указывает на первый узел в дереве, а родитель этого узла указывает обратно на корень.

**НО ТЕ** Спорим, ты не ожидал столкнуться с парадоксом дедушки из путешествий во времени, читая книгу по взлому игр!

Используя эту функцию и слегка модифицированный алгоритм обратного отслеживания из Листа 5-8, ты можешь автоматически определить, находится ли значение внутри 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 0x%x", node))
end
```

---

Помимо названий функций, единственное изменение в этом коде по сравнению с Листингом 5-8 заключается в том, что он начинает итерацию на 12 байтов перед значением, а не на 8, поскольку тар содержит три указателя вместо двух, как в list. Одним из положительных последствий структуры тар является то, что легко получить корневой узел. Когда функция \_verifyMap возвращает true, переменная parentsParent будет содержать адрес корневого узла. С небольшими модификациями можно передавать этот адрес в главный вызов и получать всё необходимое для чтения данных из std::map в одном месте.

## Заключительные мысли

Фorenзика памяти — самая трудоёмкая часть взлома игр, и её сложности могут проявляться в самых разных формах и масштабах. Однако используя принципы, шаблоны и глубокое понимание сложных структур данных, ты можешь быстро преодолеть эти трудности. Если что-то по-прежнему кажется непонятным, обязательно скачай и попробуй пример кода, предоставленный в этой главе, так как он содержит наглядные примеры всех рассмотренных алгоритмов.

В Главе 6 мы углубимся в код, который позволяет читать и записывать память игры прямо из твоих собственных программ, чтобы ты мог сделать первый шаг к практическому использованию всех этих знаний о структурах памяти, адресах и данных.

# 6) Чтение из и запись в память. Запись в память игры



В предыдущих главах обсуждалось, как устроена память, а также как сканировать и изменять память с помощью Cheat Engine и OllyDbg. Работа с памятью будет крайне важна, когда вы начнете писать ботов, и ваш код должен будет знать, как это делать.

Эта глава посвящена деталям работы с памятью на уровне кода. Во-первых, вы узнаете, как с помощью кода находить и получать дескрипторы игровых процессов. Далее вы узнаете, как использовать эти дескрипторы для чтения из памяти и записи в нее как из удаленных процессов, так и из внедренного кода. В завершение вы узнаете, как обойти определенную технику защиты памяти, а также небольшой пример инъекции кода. Код примера для этой главы вы найдете в папке GameHackingExamples/Chapter6\_AccessingMemory в каталоге исходных файлов этой книги.

**НОТЕ**

Когда я говорю о функциях API в этой главе (и в последующих), я имею в виду API Windows, если не указано иное. Если я не упоминаю заголовочный файл для библиотеки, вы можете предположить, что это Windows.h.

## Получение идентификатора процесса игры

Чтобы читать или записывать память игры, тебе нужен её идентификатор процесса (PID) — число, которое уникально идентифицирует активный процесс. Если у игры есть видимое окно, ты можешь получить PID процесса, создавшего это окно, вызвав `GetWindowThreadProcessId()`. Эта функция принимает дескриптор окна в качестве первого параметра и записывает PID во второй параметр. Найти дескриптор окна можно, передав его заголовок (текст на панели задач) в качестве второго параметра функции `FindWindow()`, как показано в Листинге 6-1.

---

```
HWND myWindow =
    FindWindow(NULL, "Title of the game window here");
DWORD PID;
GetWindowThreadProcessId(myWindow, &PID);
```

---

Листинг 6-1: Получение хэндла окна для получения PID

Получив дескриптор окна, всё, что тебе остаётся сделать — создать переменную для хранения PID и вызвать GetWindowThreadProcessId(), как показано в этом примере.

Если игра не использует окно или его имя непредсказуемо, можно найти PID, перечисляя все процессы и ища имя исполняемого файла игры. Листинг 6-2 делает это с помощью API-функций CreateToolhelp32Snapshot(), Process32First(), и Process32Next() из tlhelp32.h.

---

```
#include <tlhelp32.h>

PROCESSENTRY32 entry;
entry.dwSize = sizeof(PROCESSENTRY32);
HANDLE snapshot =
    CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
if (Process32First(snapshot, &entry) == TRUE) {
    while (Process32Next(snapshot, &entry) == TRUE) {
        wstring binPath = entry.szExeFile;
        if (binPath.find(L"game.exe") != wstring::npos) {
            printf("game pid is %d\n", entry.th32ProcessID);
            break;
        }
    }
}
CloseHandle(snapshot);
```

---

Листинг 6-2: Получение PID игры без имени окна

Листинг 6-2 может выглядеть немного сложнее, чем Листинг 6-1, но на самом деле, под всем этим кодом скрывается канонический цикл (итератор; компаратор; инкремент). Функция CreateToolhelp32Snapshot() получает список процессов, называемый snapshot, а entry — это итератор по этому списку. Возвращаемое Process32First() значение инициализирует итератор, в то время как Process32Next() увеличивает его.

Наконец, булево возвращаемое значение Process32Next() — это компаратор. Этот код просто перебирает снимок всех запущенных процессов, ищет тот, у которого путь к бинарному файлу содержит текст game.exe, и выводит его PID.

## Получение дескрипторов процессов

Как только ты узнаешь PID игры, можно получить дескриптор самого процесса, используя API-функцию OpenProcess(). Эта функция позволяет получить дескрипторы с нужными уровнями доступа для чтения и записи памяти. Это критически важно для хакинга игр, так как любая функция, работающая с процессом, требует дескриптора с соответствующими правами.

Давай посмотрим на прототип OpenProcess():

---

```
HANDLE OpenProcess(DWORD DesiredAccess, BOOL InheritHandle, DWORD ProcessId);
```

---

Первый параметр, DesiredAccess, принимает один или несколько флагов доступа, которые будут установлены для дескриптора, возвращаемого OpenProcess(). Существует много флагов, но вот самые распространённые при хакинге игр:

**PROCESS\_VM\_OPERATION** – Позволяет использовать VirtualAllocEx(), VirtualFreeEx() и VirtualProtectEx() для выделения, освобождения и защиты участков памяти.

**PROCESS\_VM\_READ** – Позволяет использовать ReadProcessMemory().

**PROCESS\_VM\_WRITE** – Позволяет использовать WriteProcessMemory(), но также требует флага PROCESS\_VM\_OPERATION. Можно установить оба флага через PROCESS\_VM\_OPERATION | PROCESS\_VM\_WRITE в DesiredAccess.

**PROCESS\_CREATE\_THREAD** – Позволяет использовать CreateRemoteThread().

**PROCESS\_ALL\_ACCESS** – Позволяет делать вообще всё. Однако этот флаг не рекомендуется, так как его могут использовать только процессы с включёнными отладочными привилегиями. Кроме того, могут возникнуть проблемы совместимости со старыми версиями Windows.

При получении дескриптора процесса игры обычно второй параметр InheritHandle устанавливается в false. Третий параметр, ProcessId, принимает PID процесса, который нужно открыть.

## Работа с OpenProcess()

Давай разберём пример вызова OpenProcess(), который использует дескриптор с правами доступа для чтения и записи памяти:

---

```
DWORD PID = getGamePID();
HANDLE process = OpenProcess(
    PROCESS_VM_OPERATION |
    PROCESS_VM_READ |
    PROCESS_VM_WRITE,
```

```
    FALSE,  
    PID  
);  
❶ if (process == INVALID_HANDLE_VALUE) {  
    printf("Failed to open PID %d, error code %d",  
        PID, GetLastError());  
}
```

---

Сначала вызов `getGamePID()` получает PID нужного процесса. (Эту функцию придётся написать самостоятельно, но можно использовать код из Листингов 6-1 и 6-2, дополнив его.) Затем код вызывает `OpenProcess()` с тремя флагами:

`PROCESS_VM_OPERATION` даёт дескриптору права на работу с памятью,

`PROCESS_VM_READ` и `PROCESS_VM_WRITE` позволяют читать и записывать данные.

Этот пример также содержит обработку ошибок ❶, но если у тебя правильный PID, установлены нужные флаги доступа и код работает с теми же правами или выше, чем игра (например, если запустить бота через **Run As Admin**), вызов должен выполняться без ошибок.

Когда работа с дескриптором завершена, его нужно закрыть вызовом `CloseHandle()`:

---

```
CloseHandle(process);
```

---

Ты можешь повторно использовать дескрипторы столько, сколько нужно, так что их можно оставлять открытыми до завершения работы или выхода бота.

Теперь, когда мы разобрались с получением дескриптора процесса для работы с памятью, давай перейдём к тому, как именно можно обращаться к памяти этого процесса.

## Доступ к памяти

API Windows предоставляет две функции, которые критически важны для доступа к памяти: `ReadProcessMemory()` и `WriteProcessMemory()`. Вы можете использовать эти функции для внешнего манипулирования памятью игры.

## Работа с `ReadProcessMemory()` и `WriteProcessMemory()`

Прототипы этих двух функций (показаны в Листинге 6-3) очень похожи друг на друга, и вы будете следовать почти точно таким же шагам для их использования.

---

```
BOOL ReadProcessMemory(
    HANDLE Process, LPVOID Address,
    LPVOID Buffer, DWORD Size,
    DWORD *NumberOfBytesRead
);
BOOL WriteProcessMemory(
    HANDLE Process, LPVOID Address,
    LPCVOID Buffer, DWORD Size,
    DWORD *NumberOfBytesWritten
);
```

---

Листинг 6-3: *ReadProcessMemory()* и *WriteProcessMemory()* прототипы

Обе функции ожидают, что Process будет дескриптором процесса, а Address будет целевым адресом памяти. Когда функция выполняет чтение из памяти, ожидается, что Buffer указывает на объект, который будет содержать считанные данные. Когда функция выполняет запись в память, ожидается, что Buffer указывает на данные, которые будут записаны. В обоих случаях Size определяет размер Buffer в байтах. Последний параметр обеих функций используется для необязательного возврата количества байтов, которые были затронуты; можно безопасно установить его в NULL. Если функция не завершится ошибкой, значение, возвращенное в последнем параметре, должно быть равно Size.

## Доступ к значению в памяти с использованием **ReadProcessMemory()** и **WriteProcessMemory()**

Код в Листинге 6-4 показывает, как можно использовать эти функции для доступа к значению в памяти.

---

```
DWORD val;
ReadProcessMemory(proc, adr, &val, sizeof(DWORD), 0);
printf("Current mem value is %d\n", val);

val++;

WriteProcessMemory(proc, adr, &val, sizeof(DWORD), 0);
ReadProcessMemory(proc, adr, &val, sizeof(DWORD), 0);
printf("New mem value is confirmed as %d\n", val);
```

---

Листинг 6-4: Чтение из памяти процесса и запись в нее с помощью Windows API

Перед тем как подобный код появится в программе, вам нужно найти PID (proc), как описано в разделе «Получение идентификатора процесса игры» на странице 120, а также адрес памяти (adr), из которого вы хотите читать или в который хотите записывать. Имея эти значения, функция *ReadProcessMemory()* сохраняет извлеченное значение из памяти в переменную val.

Затем код увеличивает val и заменяет оригинальное значение, вызывая WriteProcessMemory(). После выполнения записи ReadProcessMemory() вызывается снова на тот же адрес, чтобы подтвердить новое значение в памяти. Заметьте, что val на самом деле не является буфером. Передача &val в параметре Buffer работает, потому что это может быть указатель на любую статическую структуру памяти, если Size совпадает.

## Запись шаблонных функций доступа к памяти

Конечно, пример в Листинге 6-4 предполагает, что вы уже знаете, с каким типом памяти работаете, и жестко задает тип как DWORD. Чтобы быть универсальным игровым хакером, лучше иметь в своем арсенале некий универсальный код, чтобы избежать дублирования кода для разных типов. Универсальные функции чтения и записи памяти, которые поддерживают разные типы, могут выглядеть как в Листинге 6-5.

---

```
template<typename T>
T readMemory(HANDLE proc, LPVOID adr) {
    T val;
    ReadProcessMemory(proc, adr, &val, sizeof(T), NULL);
    return val;
}

template<typename T>
void writeMemory(HANDLE proc, LPVOID adr, T val) {
    WriteProcessMemory(proc, adr, &val, sizeof(T), NULL);
}
```

---

Листинг 6-5: Общие функции памяти

Эти функции используют шаблоны C++, чтобы принимать произвольные типы в качестве аргументов. Они позволяют вам получать доступ к памяти с любыми типами данных, которые вам нравятся, в очень чистом виде. Например, используя шаблоны функций readMemory() и writeMemory(), которые только что были показаны, можно сделать вызовы, представленные в Листинге 6-6.

```
DWORD value = readMemory<DWORD>(proc, adr); // чтение
writeMemory<DWORD>(proc, adr, value++); // увеличение и запись
```

Листинг 6-6: Вызов шаблонизированных функций доступа к памяти функции доступа

## Защита памяти

Когда игра (или любая другая программа) выделяет память, она размещается в **странице**. В x86 Windows страницы представляют собой блоки по 4096 байт, которые хранят данные. Поскольку вся память должна находиться внутри страницы, минимальная единица выделения — **4096 байт**. Операционная система может размещать блоки памяти, размер которых меньше 4096 байт, в рамках уже существующей страницы, если там есть достаточно неиспользованного пространства. Также возможно размещение в **новой** выделенной странице или через две соседние страницы с одинаковыми атрибутами.

Блоки памяти размером **4096 байт или больше** занимают **n** страниц, где **n** вычисляется по формуле:

$$n = \text{размер памяти} / 4096$$

Операционная система обычно пытается найти свободное место в существующих страницах при выделении памяти, но при необходимости выделяет новые страницы.

### NOTE

Возможно, что большие блоки займут  $n + 1$  страниц, так как нет гарантии, что блок начнётся с начала страницы.

## Различие атрибутов защиты памяти в x86 Windows

Важно понимать, что каждая страница памяти имеет набор определённых атрибутов. Большинство из них **прозрачны** в пользовательском режиме, но один из них особенно важен при работе с памятью — **защита**.

## Различие атрибутов защиты памяти в x86 Windows

Техники чтения памяти, которые вы изучили до этого момента, **очень простые**. Они предполагают, что память, к которой вы обращаетесь, защищена атрибутом **PAGE\_READWRITE**. Это предположение **корректно** для переменных данных, но другие типы данных могут находиться на страницах с **различными типами защиты**.

**Таблица 6-1:** Типы защиты памяти

Тип защиты	Значение	Чтение?	Запись?	Выполнение?	Особые разрешения?
PAGE_NOACCESS	0x01	Нет	Нет	Нет	-
PAGE_READONLY	0x02	Да	Нет	Нет	-
PAGE_READWRITE	0x04	Да	Да	Нет	-
PAGE_WRITECOPY	0x08	Да	Да	Нет	Да, копирование при записи
PAGE_EXECUTE	0x10	Нет	Нет	Да	-
PAGE_EXECUTE_READ	0x20	Да	Нет	Да	-
PAGE_EXECUTE_READWRITE	0x40	Да	Да	Да	-
PAGE_EXECUTE_WRITECOPY	0x80	Да	Да	Да	Да, копирование при записи
PAGE_GUARD	0x100	Нет	Нет	Нет	Да, сторожевая страница

Если в таблице 6-1 в любом столбце разрешений указано **Yes**, это означает, что соответствующее действие может быть выполнено на данной странице памяти.

Например, если страница имеет защиту **PAGE\_READONLY**, программа может **читать** память на этой странице, но **не может записывать** в неё.

Константные строки, например, обычно хранятся на страницах с защитой **PAGE\_READONLY**. Другие постоянные данные, такие как виртуальные таблицы функций и заголовок **Portable Executable (PE)** (который содержит информацию о программе, такую как её тип, используемые библиотечные функции, размер и так далее), также хранятся на страницах **только для чтения**. Машинный код, с другой стороны, обычно располагается на страницах с защитой **PAGE\_EXECUTE\_READ**.

Большинство типов защиты включают лишь комбинации **чтения, записи и выполнения**. Пока можно не беспокоиться о специальных типах защиты; они описаны в разделе **“Special Protection Types”** на странице 126, если вам интересно. Однако только **очень сложные хаки** когда-либо потребуют знаний об этих типах защиты. А **базовые типы защиты** памяти будут **постоянно встречаться** в ваших попытках взлома игр.

Если тип защиты в таблице 6-1 имеет значение **Да** в любом столбце разрешений, это означает, что данное действие может быть выполнено на этой странице памяти. Например, если страница **PAGE\_READONLY** то программа может читать память на этой странице, но не может записывать в эту память.

Например, постоянные строки обычно хранятся с параметром **PAGE\_READONLY** защиты. Другие постоянные данные, такие как таблицы виртуальных функций и весь модуль **Portable Executable (PE)** заголовок (содержащий информацию о программе, такую как тип

приложения, используемые библиотечные функции, размер и т. д.), также хранятся на страницах, доступных только для чтения. Ассемблерный код, с другой стороны, хранится на страницах, защищенных с помощью PAGE\_EXECUTE\_READ.

Большинство типов защиты подразумевают лишь некоторую комбинацию защиты от чтения, записи и выполнения. Пока что вы можете спокойно игнорировать специальные типы защиты типы; я рассказываю о них в разделе "Специальные типы защиты" на странице 126 если вам интересно, но их знание потребуется только для очень продвинутых взломов. Однако основные типы защиты будут преобладать в ваших приключениях по взлому игр.

### СПЕЦИАЛЬНЫЕ ТИПЫ ЗАЩИТЫ

Два типа защиты в таблице 6-1 включают защиту **copy-on-write** (*копирование при записи*). Когда несколько процессов используют страницы памяти, которые идентичны (например, страницы с загруженными системными DLL), **copy-on-write** используется для экономии памяти. Фактические данные хранятся только в одном физическом месте, а операционная система практически сопоставляет все страницы памяти, содержащие эти данные, с этим физическим расположением. Если процесс, использующий разделяемую память, вносит в неё изменения, создаётся **копия** данных в физической памяти, изменения применяются, и страницы памяти для этого процесса переназначаются на новую физическую память. Когда происходит **copy-on-write**, защита для всех затронутых страниц **изменяется** следующим образом:

- **PAGE\_WRITECOPY** превращается в **PAGE\_READWRITE**,
- **PAGE\_EXECUTE\_WRITECOPY** превращается в **PAGE\_EXECUTE\_READWRITE**.

Я не встречал специфических для взлома игр способов использования **copy-on-write** страниц, но полезно понимать, как они работают.

Также страницы могут быть созданы с **защитой guard** (*охраные страницы*). **Охраняемые страницы** должны иметь вторичную защиту, например PAGE\_GUARD | PAGE\_READONLY. Когда программа пытается получить доступ к **охраняемой странице**, операционная система выдаёт исключение STATUS\_GUARD\_PAGE\_VIOLATION. После обработки исключения защита **guard** удаляется, остаётся только вторичный тип защиты.

Один из способов, которым операционная система использует этот тип защиты — это **динамическое развертывание стека вызовов** (*call stack*), размещая **охранную страницу** вверху и выделяя больше памяти при её срабатывании. Некоторые инструменты анализа памяти используют **охраные страницы** после **кучи** (*heap*), чтобы выявлять **повреждения памяти**. В контексте взлома игр **охранная страница** может быть использована как **ловушка**, позволяющая определить, когда игра пытается обнаружить ваш код в её памяти.

## Изменение защиты памяти

Когда вы хотите взломать игру, вам иногда нужно получить доступ к памяти, защищённой атрибутами страницы памяти. Это делает возможность изменять защиту памяти **на лету** крайне важной. К счастью, Windows API предоставляет для этой цели функцию **VirtualProtectEx()**. Вот её прототип:

---

```
BOOL VirtualProtectEx(
    HANDLE Process, LPVOID Address,
    DWORD Size, DWORD NewProtect,
    PDWORD OldProtect
);
```

---

Параметры Process, Address и Size принимают те же входные данные, что и функции ReadProcessMemory() и WriteProcessMemory().

NewProtect должен указывать новые флаги защиты памяти.

OldProtect может необязательно указывать на DWORD, в котором будут сохранены старые флаги защиты.

Самая минимальная единица защиты памяти — это страница памяти. Это означает, что VirtualProtectEx() изменит защиту для каждой страницы, находящейся между Address и Address + Size – 1.

**NOTE**

Функция VirtualProtectEx() имеет сестринскую функцию под названием VirtualProtect(). Они работают одинаково, но VirtualProtect() действует только в контексте вызывающего процесса и, следовательно, не имеет параметра дескриптора процесса.

Когда вы пишете свой код для изменения защиты памяти, я предлагаю сделать его гибким, создав шаблон. Обобщённая функция для VirtualProtectEx() должна выглядеть примерно так, как показано в Листинге 6-7.

---

```
template<typename T>
DWORD protectMemory(HANDLE proc, LPVOID adr, DWORD prot) {
    DWORD oldProt;
    VirtualProtectEx(proc, adr, sizeof(T), prot, &oldProt);
    return oldProt;
}
```

---

Листинг 6-7: Общая функция для изменения защиты памяти

С этим шаблоном, если вы захотите, например, записать DWORD в страницу памяти, у которой нет разрешения на запись, ваш код мог бы выглядеть так:

---

```
protectMemory<DWORD>(process, address, PAGE_READWRITE)
writeMemory<DWORD>(process, address, newValue)
```

---

Во-первых, это устанавливает защиту памяти, изменяя её на PAGE\_READWRITE. С предоставленным разрешением на запись можно вызвать writeMemory() и изменить данные по address.

Когда вы изменяете защиту памяти, лучше всего делать это только на время, пока это необходимо, и как можно быстрее восстановить исходную защиту. Это менее эффективно, но помогает убедиться, что игра не обнаружит ваш бот (например, если некоторые страницы кода стали записываемыми).

Обычная операция записи в только-читаемую память должна выглядеть примерно так:

```
DWORD oldProt =  
    protectMemory<DWORD>(process, address, PAGE_READWRITE);  
writeMemory<DWORD>(process, address, newValue);  
protectMemory<DWORD>(process, address, oldProt);
```

Этот код вызывает функцию `protectMemory()` из Листинга 6-7, чтобы изменить защиту памяти на `PAGE_READWRITE`. Затем он записывает `newValue` в память перед тем, как изменить защиту обратно на `oldProt`, которая была установлена на исходную защиту страницы при первом вызове `protectMemory()`. Функция `writeMemory()`, используемая здесь, та же, что определена в Листинге 6-5.

Последний важный момент заключается в том, что при манипуляции памятью игривполне возможно, что игра будет обращаться к памяти одновременно с вами. Если новая защита, которую вы установили, несовместима с оригинальной защитой, процесс игры получит исключение `ACCESS_VIOLATION` и завершится аварийно (`crash`).

Например, если изменить защиту памяти с `PAGE_EXECUTE` на `PAGE_READWRITE`, игра может попытаться выполнить код на странице, в то время как память больше не помечена как исполняемая.

В этом случае лучше установить защиту памяти на `PAGE_EXECUTE_READWRITE`, чтобы убедиться, что вы можете работать с памятью, но игра всё ещё может выполнять код на этих страницах.

## Рандомизация размещения адресного пространства (ASLR)

До сих пор я описывал адреса памяти как статические целевые числа, которые изменяются только при изменении бинарного кода. Эта модель корректна для Windows XP и более ранних версий.

Однако в более поздних версиях Windows адреса памяти являются статическими только относительно базового адреса игрового исполняемого файла, поскольку эти системы поддерживают функцию, называемую рандомизация размещения адресного пространства (Address Space Layout Randomization, ASLR).

Когда бинарный файл скомпилирован с поддержкой ASLR (включено по умолчанию в MSVC++ 2010 и многих других компиляторах), его базовый адрес может меняться при каждом запуске.

Наоборот, не-ASLR бинарные файлы всегда будут иметь фиксированный базовый адрес `0x400000`.

**НОТЕ** Поскольку ASLR не работает на XP, я буду называть `0x400000` XP-базой.

## Отключение ASLR для упрощения разработки ботов

Чтобы упростить разработку, можно отключить ASLR и использовать XP-базу адресов. Для этого выполните команду в Командной строке Visual Studio:

---

```
> editbin /DYNAMICBASE:NO "C:\path\to\game.exe"
```

---

Чтобы снова включить его, введите:

---

```
> editbin /DYNAMICBASE"C:\path\to\game.exe"
```

---

## Обход ASLR в продакшне

Отключение ASLR подходит для разработки ботов, но не для продакшена – нельзя ожидать, что конечные пользователи будут выключать ASLR вручную.

Вместо этого можно написать функцию, которая динамически перерассчитывает адреса во время выполнения.

Если вы используете XP-базу (0x400000), код для пересчёта адреса будет таким:

---

```
DWORD rebase(DWORD address, DWORD newBase) {
    DWORD diff = address - 0x400000;
    return diff + newBase;
}
```

---

Когда вы знаете базовый адрес игры (*newBase*), эта функция позволяет вам фактически игнорировать ASLR, перебазируя адрес.

Чтобы найти *newBase*, вам нужно использовать функцию **GetModuleHandle()**.

Когда параметр **GetModuleHandle()** равен **NULL**, он всегда возвращает дескриптор главного бинарного файла в процессе. Возвращаемый тип функции — **HMODULE**, но фактически это просто адрес, по которому загружается бинарный файл. Это и есть базовый адрес, так что вы можете привести его к **DWORD**, чтобы получить *newBase*. Поскольку вы ищете базовый адрес в другом процессе, вам нужно будет каким-то образом выполнить эту функцию в контексте этого процесса.

Чтобы сделать это, вызовите **GetModuleHandle()** с помощью API-функции **CreateRemoteThread()**, которая позволяет создавать потоки и выполнять код в удалённом процессе. Её прототип показан в Листинге 6-8.

---

```
HANDLE CreateRemoteThread(
    HANDLE Process,
    LPSECURITY_ATTRIBUTES ThreadAttributes,
    DWORD StackSize,
    LPTHREAD_START_ROUTINE StartAddress,
    LPVOID Param,
    DWORD CreationFlags,
    LPDWORD ThreadId
);
```

---

Листинг 6-8: Функция, порождающая поток

Созданный поток начнёт выполнение с **StartAddress**, рассматривая его как функцию с одним параметром, где **Param** будет передан в качестве входных данных, а возвращённое значение будет установлено как код выхода потока.

Это идеально, так как поток можно запустить с **StartAddress**, указывающим на адрес **GetModuleHandle()**, и **Param**, установленным в **NULL**. Затем можно использовать API-функцию **WaitForSingleObject()**, чтобы ждать завершения потока, а затем получить возвращённый базовый адрес с помощью API-функции **GetExitCodeThread()**.

Как только все эти элементы объединены, код для получения **newBase** из внешнего бота должен выглядеть, как в Листинге 6-9.

```
DWORD newBase;

// Получаем адрес kernel32.dll
HMODULE k32 = GetModuleHandle("kernel32.dll");

// Получаем адрес функции GetModuleHandle()
LPVOID funcAddr = GetProcAddress(k32, "GetModuleHandleA");
if (!funcAddr)
    funcAddr = GetProcAddress(k32, "GetModuleHandleW");

// Создаём поток
HANDLE thread =
    CreateRemoteThread(process, NULL, NULL,
                       (LPTHREAD_START_ROUTINE)funcAddr,
                       NULL, NULL, NULL);

// Ждём завершения потока
WaitForSingleObject(thread, INFINITE);

// Получаем код выхода (адрес базового модуля)
GetExitCodeThread(thread, &newBase);

// Закрываем дескриптор потока
CloseHandle(thread);
```

Листинг 6-9: Поиск базового адреса игры с помощью функций API

Функция **GetModuleHandle()** является частью **kernel32.dll**, которая имеет один и тот же базовый адрес в каждом процессе. Поэтому сначала этот код получает адрес **kernel32.dll**. Поскольку базовый адрес **kernel32.dll** одинаков во всех процессах, адрес **GetModuleHandle()** будет таким же как в игре, так и во внешнем боте. Зная базовый адрес **kernel32.dll**, этот код с лёгкостью находит адрес **GetModuleHandle()** с помощью API-функции **GetProcAddress()**. Далее он вызывает **CreateRemoteThread()** (из **Листинга 6-8**), даёт потоку выполнить свою работу и извлекает код завершения, чтобы получить **newBase**.

## Итоговые мысли

Теперь, когда ты знаешь, как манипулировать памятью с помощью своего кода, я покажу, как применять эти навыки в играх. Эти навыки являются основой для концепций, которые будут рассмотрены в следующих главах, так что убедись, что ты хорошо их усвоил. Если возникают сложности, поиграй с примерами кода и протестируй методы, чтобы лучше понять их работу.

Способ, которым **Листинг 6-9** заставляет игру выполнить `GetModuleHandle()`, является формой **внедрения кода** (*code injection*). Но это только поверхностный взгляд на то, что можно сделать с инъекциями. Если ты хочешь изучить этот процесс глубже, переходи к **Главе 7**, где эта тема рассматривается более подробно.

## **Часть 3: Управление процессами**

## 7) Инъекция кода



Представь, что ты можешь войти в офис игровой компании, сесть за компьютер и начать добавлять код в их игровой клиент. Представь, что ты можешь делать это для любой игры, когда захочешь и для любой функциональности, которую пожелаешь. Почти каждый игрок, с которым ты поговоришь, будет иметь идеи о том, как улучшить игру, но, насколько они знают, это всего лишь несбыточная мечта.

Но ты знаешь, что мечты существуют, чтобы их воплощать в жизнь, и теперь, когда ты немного изучил, как работает память, ты готов отбросить все правила. Используя инъекцию кода (code injection), ты можешь, по сути, стать таким же могущественным, как и разработчики игры.

Инъекция кода — это метод, позволяющий принудительно выполнять чужой код в любом процессе внутри его собственного адресного пространства и контекста выполнения. Я уже затрагивал эту тему ранее в разделе «Обход ASLR в продакшене» на странице 128, где показал, как удалённо обойти ASLR с помощью `CreateRemoteThread()`, но тот пример лишь слегка коснулся темы.

В первой части этой главы ты узнаешь, как создавать code caves (пещеры кода), внедрять новые потоки и перехватывать потоки выполнения, заставляя игры исполнять небольшие фрагменты ассемблерного кода.

Во второй части ты узнаешь, как внедрять в игры сторонние исполняемые файлы, заставляя их выполнять целые программы, которые ты создал.

### Внедрение кодовых пещер (code caves) с инъекцией потока

Первый шаг к внедрению кода в другой процесс — это написание позиционно-независимого ассемблерного кода, известного как **shellcode**, в форме массива байтов. Вы можете записать shellcode для удаления процессов и формирования **кодовых пещер** (code caves), которые действуют как точка входа

для нового потока, который вы хотите выполнить. Как только кодовая пещера создана, вы можете выполнить её, используя **инъекцию потока** (*thread injection*) или **угон потока** (*thread hijacking*). В этом разделе я покажу вам пример инъекции потока, а также продемонстрирую угон главного потока игры в «Перехват главного потока игры для выполнения кодовых пещер» на странице 138. Вы найдете примеры кода для этой главы в ресурсных файлах этой книги в каталоге **GameHackingExamples/Chapter7\_CodeInjection**. Откройте **main-codeInjection.cpp** для выполнения шагов вместе со мной, пока я объясняю, как создать упрощённую версию функции **injectCodeUsingThreadInjection()** из этого файла.

## Создание ассемблерной кодовой пещеры

В разделе «**Обход ASLR в продакшене**» на странице 128 я использовал инъекцию потока для вызова функции **GetModuleHandle()** через **CreateRemoteThread()** и получения дескриптора процесса. В том случае **GetModuleHandle()** действовала как кодовая пещера; она имела правильную кодовую структуру, чтобы служить точкой входа для нового потока. Однако инъекция потока не всегда так проста.

Например, представьте, что ваш внешний бот должен удалённо вызвать функцию в игре, и эта функция имеет следующий прототип:

---

```
DWORD __cdecl someFunction(int times, const char* string);
```

---

Есть несколько нюансов, которые делают удалённый вызов этой функции сложным. Во-первых, у неё есть два параметра, что означает, что вам нужно создать кодовую пещеру, которая установит стек и правильно выполнит вызов. **CreateRemoteThread()** позволяет передать **только один** аргумент в кодовую пещеру, и вы можете получить доступ к этому аргументу относительно регистра **ESP**, но второй аргумент всё равно придется жёстко прописывать в кодовой пещере. Жёстко задавать **первый аргумент** (*times*) проще всего. Дополнительно, вам нужно убедиться, что кодовая пещера правильно очищает стек.

**НОТЕ**

Помните, что при обходе ASLR в главе 6 я использовал **CreateRemoteThread()** для запуска новых потоков, выполняя любой произвольный код по заданному адресу и передавая в кодовую пещеру один параметр. Именно поэтому примеры здесь используют передачу одного параметра через стек.

В конечном итоге кодовая пещера для инъекции вызова **someFunction** в запущенный процесс игры будет выглядеть примерно так (в виде псевдокода):

```
PUSH DWORD PTR:[ESP+0x4]    // получить второй аргумент из стека
PUSH times
CALL someFunction
ADD ESP, 0x8
RETN
```

Эта кодовая пещера почти идеальна, но её можно сделать менее сложной. Операция **CALL** ожидает один из двух операндов: либо регистр с абсолютным адресом функции, либо непосредственное целое число, которое содержит смещение относительно адреса возврата. Это означает, что вам придется выполнить ряд расчетов смещений, что может быть утомительно.

Чтобы сделать кодовую пещеру независимой от позиции, измените её так, чтобы она использовала регистр, как показано в **Листинге 7-1**.

```
PUSH DWORD PTR:[ESP+0x4]    // получить второй аргумент из стека
PUSH times
MOV EAX, someFunction
CALL EAX
ADD ESP, 0x8
RETN
```

*Листинг 7-1: Код пещеры для вызова someFunction*

Поскольку вызывающий код знает, что вызываемая функция перезапишет **EAX** своим возвращаемым значением, он должен убедиться, что **EAX** не содержит никаких критических данных. Зная это, можно использовать **EAX** для хранения абсолютного адреса **someFunction**.

## Перевод ассемблера в шеллкод

Поскольку кодовые пещеры необходимо записывать в память другого процесса, их нельзя написать напрямую на ассемблере. Вместо этого вам нужно записывать их побайтно. Не существует стандартного способа определить, какие байты представляют собой определенный ассемблерный код, но есть несколько обходных методов.

Мой личный фаворит — скомпилировать пустое приложение на C++ с этим ассемблерным кодом внутри функции и затем использовать **OllyDbg**, чтобы проанализировать эту функцию. Альтернативный метод — открыть **OllyDbg** в любом произвольном процессе и просканировать дизассемблированный код, пока не найдёте байты для всех необходимых вам операций.

Этот метод на самом деле очень хорош, так как ваши кодовые пещеры должны быть написаны как можно проще. Это означает,

что все операции должны быть довольно стандартными. Вы также можете найти онлайн-таблицы опкодов ассемблера, но я считаю, что их довольно сложно читать; описанные выше методы проще в использовании.

Когда вы точно знаете, какие байты вам нужны, вы можете использовать **C++** для генерации правильного шеллкода. **Листинг 7-2** показывает завершённый скелет шеллкода для ассемблерного кода из **Листинга 7-1**.

---

```
BYTE codeCave[20] = {  
    0xFF, 0x74, 0x24, 0x04,      // PUSH DWORD PTR:[ESP+0x4]  
    0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0  
    0xB8, 0x00, 0x00, 0x00, 0x00, // MOV EAX, 0x0  
    0xFF, 0xD0,                  // CALL EAX  
    0x83, 0xC4, 0x08,            // ADD ESP, 0x08  
    0xC3                        // RETN  
};
```

---

Листинг 7-2: Скелет шеллкода

Этот пример создаёт массив **BYTE**, содержащий необходимые байты шеллкода. Однако аргумент **times** должен быть динамическим, и невозможно узнать адрес **someFunction** во время компиляции. Именно поэтому этот шеллкод записан в виде скелета.

Две группы из четырёх последовательных байтов **0x00** являются заполнителями для **times** и адреса **someFunction**. Вы можете вставить реальные значения в вашу кодовую пещеру во время выполнения, используя вызовы **memcpy()**, как показано во фрагменте в **Листинге 7-3**.

---

```
memcpy(&codeCave[5], &times, 4);  
memcpy(&codeCave[10], &addressOfSomeFunc, 4);
```

---

Листинг 7-3: Вставка раз и местоположение *someFunction* в кодовую пещеру

Оба значения **times** и адрес **someFunction** занимают 4 байта (напомним, что **times** — это **int**, а адреса — 32-битные значения). Они находятся в **codeCave[5-8]** и **codeCave[10-13]** соответственно.

Два вызова **memcpy()** передают эту информацию в качестве параметров для заполнения пропусков в массиве **codeCave**.

## Запись кодовой пещеры в память

С правильно созданным шеллкодом, вы можете поместить его внутрь целевого процесса, используя **VirtualAllocEx()** и **WriteProcessMemory()**. Листинг 7-4 показывает один из способов сделать это.

```

int stringlen = strlen(string) + 1; // +1, чтобы включить нулевой терминатор
int cavelen = sizeof(codeCave);

❶ int fulllen = stringlen + cavelen;
auto remoteString = // Выделить память с правами EXECUTE
❷ VirtualAllocEx(process, 0, fulllen, MEM_COMMIT, PAGE_EXECUTE);

auto remoteCave = // запомнить, куда будет помещена кодовая пещера
❸ (LPVOID)((DWORD)remoteString + stringlen);

// сначала записать строку
❹ WriteProcessMemory(process, remoteString, string, stringlen, NULL);

// затем записать кодовую пещеру
❺ WriteProcessMemory(process, remoteCave, codeCave, cavelen, NULL);
❻

```

Листинг 7-4: Запись финального шеллкода в память кодовой пещеры

Сначала этот код определяет, сколько байтов памяти потребуется для записи строкового аргумента и кодовой пещеры в память игры, и сохраняет это значение в fullLen ❶. Затем он вызывает API-функцию VirtualAllocEx() для выделения fullLen байтов внутри process с защитой PAGE\_EXECUTE (вы также можете использовать 0 и MEM\_COMMIT соответственно для второго и четвертого параметров), и сохраняет адрес выделенной памяти в remoteString ❷.

Затем remoteString увеличивается на stringLen байтов и результат сохраняется в remoteCave ❸, так как шеллкод должен быть записан непосредственно в память после строкового аргумента.

Наконец, он использует WriteProcessMemory() для заполнения выделенного буфера строкой string ❹ и машинным кодом ❺, сохраненным в codeCave.

Таблица 7-1 показывает, как может выглядеть дамп памяти кодовой пещеры, предполагая, что он размещен по адресу 0x030000, someFunction находится по адресу 0xDEADBEEF, times установлено в 5, а string указывает на текст "injected!".

Адрес	Представление кода	Сырые данные	Значение данных
0x030000	remoteString[0-4]	0x69 0x6E 0x6A 0x65 0x63	injec
0x030005	remoteString[5-9]	0x74 0x65 0x64 0x0A 0x00	ted!\0
0x03000A	remoteCave[0-3]	0xFF 0x74 0x24 0x04	PUSH DWORD PTR [ESP+0x4]
0x03000E	remoteCave[4-8]	0x68 0x05 0x00 0x00	PUSH 0x05
0x030013	remoteCave[9-13]	0xB8 0xEF 0xBE 0xAD 0xDE	MOV EAX, 0xDEADBEEF
0x030018	remoteCave[14-15]	0xFF 0xD0	CALL EAX
0x03001A	remoteCave[16-18]	0x83 0xC4 0x08	ADD ESP, 0x08
0x03001D	remoteCave[19]	0xC3	RETN

## Использование инъекции потока для выполнения кодовой пещеры

Когда кодовая пещера полностью записана в память, остается только выполнить её. В этом примере вы можете выполнить пещеру с помощью следующего кода:

---

```
HANDLE thread = CreateRemoteThread(process, NULL, NULL,
                                    (LPTHREAD_START_ROUTINE)remoteCave,
                                    remoteString, NULL, NULL);

WaitForSingleObject(thread, INFINITE);
CloseHandle(thread);
VirtualFreeEx(process, remoteString, fulllen, MEM_RELEASE)
```

---

Вызовы `CreateRemoteThread()`, `WaitForSingleObject()` и `CloseHandle()` работают совместно, чтобы внедрить и выполнить кодовую пещеру, а `VirtualFreeEx()` очищает следы бота, освобождая память, выделенную в коде, как показано в Листинге 7-4.

В самой простой форме это всё, что нужно для выполнения кодовой пещеры, внедренной в игру. Однако на практике вам следует проверять возвращаемые значения после вызовов `VirtualAllocEx()`, `WriteProcessMemory()` и `CreateRemoteThread()`, чтобы убедиться, что все операции прошли успешно.

Например, если `VirtualAllocEx()` возвращает `0x00000000`, это означает, что выделение памяти завершилось неудачно. Если вы не обработаете этот сбой, `WriteProcessMemory()` также завершится с ошибкой, а `CreateRemoteThread()` начнет выполняться с точкой входа `0x00000000`, что в конечном итоге приведет к сбою игры. То же самое относится к возвращаемым значениям `WriteProcessMemory()` и `CreateRemoteThread()`. Обычно эти функции будут завершаться с ошибкой, если

дескриптор процесса был открыт без необходимых флагов доступа.

## Захват основного потока игры для выполнения кодовых пещер

В некоторых случаях внедренные кодовые пещеры должны быть синхронизированы с основным потоком игрового процесса. Решение этой проблемы может быть довольно сложным, поскольку требует управления существующими потоками во внешнем процессе.

Вы могли бы просто приостановить выполнение основного потока до завершения кодовой пещеры, что могло бы сработать, но оказалось бы очень медленным. Накладные расходы, связанные с ожиданием завершения кодовой пещеры и последующим возобновлением потока, довольно высоки. Более быстрый вариант — заставить поток выполнять код за вас, используя процесс, называемый **захватом потока** (*thread hijacking*).

### NOTE

Откройте файл *main-codeInjection.cpp* в исходных файлах этой книги, чтобы следовать за процессом создания примера захвата потока. Это упрощенная версия *injectCodeUsingThreadHijacking()*.

## Создание кодовой пещеры на ассемблере

Как и при инъекции потока, первый шаг в **захвате потока** — это понимание того, что именно должно происходить в вашей кодовой пещере. Однако в этот раз вы **не знаете**, что поток будет выполнять в момент захвата, поэтому вам **необходимо сохранить его состояние** при запуске кодовой пещеры и восстановить его после завершения захвата. Это означает, что ваш шелл-код должен быть обернут в ассемблерные инструкции, как показано в Листинге 7-5.

```
PUSHAD // сохранить все общие регистры в стек  
PUSHFD // сохранить флаговый регистр EFLAGS в стек  
  
// здесь должен находиться шелл-код  
  
POPFD // восстановить EFLAGS из стека  
POPAD // восстановить общие регистры из стека  
  
// возобновить выполнение потока без использования регистров здесь
```

Листинг 7-5: Фреймворк для пещеры с потоковым захватом кода

Если бы вы вызвали ту же самую *someFunction*, что и при инъекции потока, вы могли бы использовать шелл-код, похожий на приведенный в **Листинге 7-2**. Единственное отличие в том, что вы не могли бы передать второй параметр вашему боту через стек, поскольку не использовали бы *CreateRemoteThread()*.

Но это не проблема; вы могли бы просто **поместить его в стек** так же, как и первый параметр. Часть кодовой пещеры, которая

выполняет вызов функции, должна выглядеть, как показано в **Листинге 7-6**:

---

```
PUSH string
PUSH times
MOV EAX, someFunction
CALL EAX
ADD ESP, 0x8
```

---

*Листинг 7-6: Скелет сборки для вызова someFunction*

Единственное изменение по сравнению с **Листингом 7-1** заключается в том, что в этом примере строка `string` **явно помещается в стек, и отсутствует RETN**. Вы не вызываете RETN в этом случае, потому что хотите, чтобы игровой поток **продолжил работу** так, как он работал до того, как был захвачен.

Чтобы **нормально возобновить выполнение потока**, кодовая пещера должна **перейти обратно к изначальному EIP** (указателю на команду) **без использования регистров**. К счастью, можно использовать функцию `GetThreadContext()`, чтобы получить EIP, заполнив каркас кода в C++. Затем можно **поместить его в стек** внутри кодовой пещеры и выполнить возврат.

**Листинг 7-7** показывает, как кодовая пещера должна завершаться.

---

```
PUSH originalEIP
RETN
```

---

*Листинг 7-7: Переход к EIP косвенным путем*

Возврат (RETN) прыгает к значению, находящемуся на вершине стека, поэтому выполнение этой операции сразу после помещения EIP в стек решит проблему.

Вы должны использовать этот метод вместо команды перехода (JMP), так как переходы требуют вычисления смещения, что усложняет генерацию шелл-кода.

Если объединить Листинги 7-5 — 7-7, то получится следующая кодовая пещера:

```

// сохранить состояние
PUSHAD           // поместить все основные регистры в стек
PUSHFD           // поместить флаги EFLAGS в стек

// выполнить работу шелл-кода
PUSH string
PUSH times
MOV EAX, someFunction
CALL EAX
ADD ESP, 0x8

// восстановить состояние
POPFD            // извлечь EFLAGS из стека
POPAD            // извлечь основные регистры из стека

// отмена перехвата: возобновление потока без использования регистров
PUSH originalEIP
RETN

```

Далее, следуйте инструкциям в разделе "Перевод ассемблера в шелл-код" на странице 135 и вставьте полученные байты в массив, представляющий вашу кодовую пещеру.

## Генерация базового шелл-кода и выделение памяти

Используя тот же метод, который показан в Листинге 7-2, вы можете сгенерировать шелл-код для этой кодовой пещеры, как показано в Листинге 7-8.

```

BYTE codeCave[31] = {
    0x60,                      // PUSHAD (сохранение всех регистров)
    0x9C,                      // PUSHFD (сохранение флагов процессора)
    0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0 (зарезервировано для remoteString)
    0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0 (зарезервировано для times)
    0xB8, 0x00, 0x00, 0x00, 0x00, // MOV EAX, 0x00000000 (зарезервировано для someFunction)
    0xFF, 0xD0,                 // CALL EAX (вызов функции)
    0x83, 0xC4, 0x08,           // ADD ESP, 0x08 (очистка стека)
    0x9D,                      // POPFD (восстановление флагов процессора)
    0x61,                      // POPAD (восстановление всех регистров)
    0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0 (зарезервировано для сохранения EIP)
    0xC3,                      // RETN (возврат из функции)
};

// Нам нужно добавить код, который разместит
// EIP потока в threadContext.Eip

memcpy(&codeCave[3], &remoteString, 4); // Вставка remoteString
memcpy(&codeCave[8], &times, 4);        // Вставка значения times
memcpy(&codeCave[13], &func, 4);         // Вставка адреса someFunction
memcpy(&codeCave[25], &threadContext.Eip, 4); // Вставка EIP потока

```

Листинг 7-8: Создание массива шеллкодов для взлома потоков

Как и в Листинге 7-3, `memcpuy()` используется для вставки переменных в скелет. Однако, в отличие от того листинга, здесь есть две переменные, которые нельзя сразу скопировать: `times` и `func` известны сразу, но `remoteString` является результатом выделения памяти, а `threadContext.Eip` станет известен только после того, как поток будет заморожен. Также логично выделять память перед заморозкой потока, поскольку вы не хотите, чтобы поток был заморожен дольше, чем это необходимо.

Вот как это может выглядеть:

```
int stringlen = strlen(string) + 1; // +1 для нулевого терминатора
int cavelen = sizeof(codeCave);      // Размер codeCave
int fulllen = stringlen + cavelen;   // Общий размер памяти

auto remoteString =
    VirtualAllocEx(process, 0, fulllen, MEM_COMMIT, PAGE_EXECUTE); // Выделение памяти

auto remoteCave =
    (LPVOID)((DWORD)remoteString + stringlen); // Определение области для codeCave
```

Код выделения памяти такой же, как и для внедрения через потоки, поэтому можно повторно использовать этот же фрагмент кода.

## Нахождение и заморозка основного потока

Код для заморозки основного потока немного сложнее. Сначала необходимо получить его уникальный идентификатор. Это работает примерно так же, как получение PID, и для этого можно использовать `CreateToolhelp32Snapshot()`, `Thread32First()` и `Thread32Next()` из `TlHelp32.h`.

Как обсуждалось в разделе «Получение идентификатора процесса игры» на странице 120, эти функции в основном используются для итерации по списку потоков. Процесс может иметь множество потоков, но в следующем примере предполагается, что первый созданный потоком процесс — это тот, который нужно захватить:

---

```
DWORD GetProcessThreadId(HANDLE Process) {
    THREADENTRY32 entry;
    entry.dwSize = sizeof(THREADENTRY32);
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);

    if (Thread32First(snapshot, &entry) == TRUE) {
        DWORD PID = GetProcessId(Process);
        while (Thread32Next(snapshot, &entry) == TRUE) {
            if (entry.th32OwnerProcessID == PID) {
                CloseHandle(snapshot);
                return entry.th32ThreadID;
            }
        }
    }
    CloseHandle(snapshot);
    return NULL;
}
```

---

Этот код просто перебирает список всех потоков в системе и находит первый, который соответствует PID игры. Затем он получает идентификатор потока из записи снапшота. Узнав идентификатор потока, получите его текущее состояние регистра следующим образом:

---

```
HANDLE thread = OpenThread(
    (THREAD_GET_CONTEXT | THREAD_SUSPEND_RESUME | THREAD_SET_CONTEXT),
    false, threadID);
SuspendThread(thread);
CONTEXT threadContext;
threadContext.ContextFlags = CONTEXT_CONTROL;
GetThreadContext(thread, &threadContext);
```

---

Этот код использует `OpenThread()`, чтобы получить дескриптор потока. Затем он приостанавливает поток с помощью `SuspendThread()` и извлекает значения его регистров с помощью `GetThreadContext()`.

После этого, код `memcpuy()` из Листинга 7-8 должен вставить все переменные, необходимые для завершения генерации шеллкода.

Когда шеллкод сгенерирован, кодовая пещера (*code cave*) может быть записана в выделенную память таким же способом, как это делалось в Листинге 7-4.

---

```
WriteProcessMemory(process, remoteString, string, stringlen, NULL);
WriteProcessMemory(process, remoteCave, codeCave, cavelen, NULL);
```

---

Как только кодовая пещера (*code cave*) загружена в память и готова к выполнению, всё, что нужно сделать, — это установить EIP потока на адрес кодовой пещеры и возобновить выполнение потока, следующим образом:

---

```
threadContext.Eip = (DWORD)remoteCave;
threadContext.ContextFlags = CONTEXT_CONTROL;
SetThreadContext(thread, &threadContext);
ResumeThread(thread);
```

---

Этот код заставляет поток возобновить выполнение с адреса кодовой пещеры. Из-за того, как написана кодовая пещера, поток не имеет ни малейшего представления о том, что что-то изменилось. Пещера сохраняет исходное состояние потока, выполняет полезную нагрузку (*payload*), восстанавливает первоначальное состояние и затем возвращается к оригинальному коду без каких-либо следов.

Когда вы используете любую форму инъекции кода, очень важно понимать, какие данные затрагивает кодовая пещера. Например, если кодовая пещера вызывает внутренние функции игры для создания и отправки сетевого пакета, вам необходимо убедиться, что все глобальные переменные, которые могут быть изменены (например, буфер пакета, маркер позиции пакета и другие), безопасно восстанавливаются после выполнения.

Вы никогда не знаете, что делает игра в момент выполнения

кодовой пещеры — она может в этот момент вызывать ту же самую функцию, что и ваш код!

## Инъекция DLL для полного контроля

Кодовые пещеры очень мощные (вы можете заставить игру делать что угодно, используя ассемблерный shellcode), но вручную писать shellcode — это непрактично. Было бы намного удобнее внедрять C++-код, не так ли? Это возможно, но процесс гораздо сложнее: код должен быть скомпилирован в машинный код, упакован в формат, не зависящий от расположения в памяти (position-agnostic format), осведомлён о любых внешних зависимостях, полностью загружен в память, затем выполнен в определённой точке входа.

## Коварный способ заставить процесс загрузить вашу DLL

К счастью, все эти вещи уже учтены в Windows. Изменив проект C++ так, чтобы он компилировался как динамическая библиотека, вы можете создать автономный, независимый от размещения двоичный файл, называемый динамически загружаемой библиотекой (DLL). Затем можно использовать комбинацию инъекции потока или перехвата и API-функцию `LoadLibrary()`, чтобы отобразить ваш DLL-файл в памяти игры.

Откройте `main-codeInjection.cpp` в каталоге `GameHackingExamples/Chapter7_CodeInjection` и `dllmain.cpp` в `GameHackingExamples/Chapter7_CodeInjection_DLL`, чтобы следовать за примерами кода во время прочтения этой секции. В `main-codeInjection.cpp` обратите внимание на функцию `LoadDLL()`.

## Обман процесса для загрузки вашей DLL

Используя кодовую пещеру, можно заставить удалённый процесс вызвать **`LoadLibrary()`** для загрузки DLL, тем самым фактически внедряя сторонний код в его адресное пространство. Так как **`LoadLibrary()`** принимает только один параметр, можно создать кодовую пещеру, которая вызовет её следующим образом:

```

// записать имя DLL в память
wchar_t* dllName = L"c:\\something.dll";
int namelen = wcslen(dllName) + 1;
LPVOID remoteString =
    VirtualAllocEx(process, NULL, namelen * 2, MEM_COMMIT, PAGE_EXECUTE);
WriteProcessMemory(process, remoteString, dllName, namelen * 2, NULL);

// получить адрес LoadLibraryW()
HMODULE k32 = GetModuleHandleA("kernel32.dll");
LPVOID funcAddr = GetProcAddress(k32, "LoadLibraryW");

// создать поток, который вызовет LoadLibraryW(dllName)
HANDLE thread =
    CreateRemoteThread(process, NULL, NULL,
    (LPTHREAD_START_ROUTINE)funcAddr,
    remoteString, NULL, NULL);

// дождаться завершения потока и очистить ресурсы
WaitForSingleObject(thread, INFINITE);
CloseHandle(thread);

```

Этот код представляет собой нечто среднее между кодом инъекции потока из раздела "Обход ASLR в продакшене" на странице 128 и кодом пещеры кода (code cave), вызывающего someFunction, как в Листингах 7-2 и 7-3. Как и в первом случае, этот пример использует тело API-функции с одним параметром, а именно LoadLibrary, в качестве тела пещеры кода. Однако, как и во втором случае, он должен внедрить строку в память, так как LoadLibrary ожидает указатель на строку в качестве первого аргумента.

Как только поток создан, он заставляет LoadLibrary загрузить DLL, имя которой было внедрено в память, эффективно внедряя сторонний код в игру.

**NOTE**

*Дайте любой DLL, которую вы планируете внедрять, уникальное имя, например, MySuperBotV2Hook.dll. Простые имена, такие как Hook.dll или Injected.dll, слишком распространены. Если имя DLL совпадёт с уже загруженной библиотекой, LoadLibrary() решит, что она уже загружена, и не загрузит её!*

Как только пещера кода LoadLibrary() загрузит вашу DLL в игру, точка входа этой DLL, известная как DllMain(), будет выполнена с причиной DLL\_PROCESS\_ATTACH. Когда процесс будет завершён или FreeLibrary() будет вызван для DLL, её точка входа будет вызвана с причиной DLL\_PROCESS\_DETACH. Обработка этих событий в точке входа может выглядеть следующим образом:

(следует код).

```
BOOL APIENTRY DllMain(HMODULE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved) {
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
            printf("DLL attached!\n");
            break;
        case DLL_PROCESS_DETACH:
            printf("DLL detached!\n");
            break;
    }
    return TRUE;
}
```

Эта примерная функция начинается с проверки, почему была вызвана DllMain(). Затем она выводит текст, указывающий, было ли это вызвано из-за подключения или отключения DLL, возвращая TRUE в любом случае.

Имейте в виду, что точка входа DLL выполняется внутри loader lock, который представляет собой глобальную блокировку синхронизации, используемую всеми функциями, которые читают или изменяют список загруженных в процесс модулей. Эта блокировка загрузчика используется такими функциями, как GetModuleHandle(), GetModuleFileName(), Module32First() и Module32Next(), что означает, что выполнение сложного кода из точки входа DLL может привести к взаимоблокировке (deadlock) и должно быть избегаемо.

Если вам нужно выполнить код из точки входа DLL, сделайте это из нового потока, следующим образом: (следует код).

```
DWORD WINAPI runBot(LPVOID lpParam) {
    // запускаем вашего бота
    return 1;
}

// выполняем это из DllMain() в случае DLL_PROCESS_ATTACH
auto thread = CreateThread(NULL, 0, &runBot, NULL, 0, NULL);
CloseHandle(thread);
```

Из DllMain(), этот код создает новый поток, который запускает функцию runBot(). Затем он немедленно закрывает дескриптор потока, так как выполнение любых дальнейших операций из DllMain() может привести к серьезным проблемам.

Изнутри runBot(), вы можете начать выполнение кода вашего бота. Код работает внутри игры, что означает, что вы можете напрямую манипулировать памятью, используя методы приведения типов. Вы также можете делать гораздо больше, как будет показано в Главе 8.

При внедрении DLL убедитесь, что у вас нет проблем с зависимостями. Если ваша DLL зависит от каких-либо

нестандартных DLL, например, вам нужно либо сначала внедрить эти DLL в игру, либо поместить их в папку, которую сможет найти `LoadLibrary()`, например, любую папку в переменной окружения `PATH`. Первый вариант будет работать только в том случае, если у DLL нет собственных зависимостей, тогда как второй вариант немного сложнее в реализации и подвержен конфликтам имен. Лучший вариант — статически связать все внешние библиотеки, чтобы они были скомпилированы непосредственно в вашу DLL.

## Доступ к памяти во внедренной DLL

Когда вы пытаетесь получить доступ к памяти игры из внедренной DLL, дескрипторы процессов и функции API являются помехой. Поскольку игра использует то же адресное пространство, что и весь код, внедренный в нее, вы можете получить доступ к памяти игры напрямую из внедренного кода. Например, чтобы получить доступ к значению `DWORD` из внедренного кода, можно написать следующее:

```
DWORD value = *((DWORD*)adr); // чтение DWORD из adr
*((DWORD*)adr) = 1234; // запись 1234 в DWORD adr
```

Это просто приводит адрес памяти `adr` к `DWORD*` и разыменовывает этот указатель в `DWORD`. Делать приведение типов прямо в коде — это нормально, но код доступа к памяти будет выглядеть чище, если функции будут абстрагированы и сделаны универсальными, как обертки Windows API.

Обобщенные функции для доступа к памяти внутри внедренного кода выглядят следующим образом:

```
template<typename T>
T readMemory(LPVOID adr) {
    return *((T*)adr);
}

template<typename T>
void writeMemory(LPVOID adr, T val) {
    *((T*)adr) = val;
}
```

Использование этих шаблонов аналогично использованию функций из раздела "Написание шаблонных функций доступа к памяти" на странице 123. Вот пример:

```
DWORD value = readMemory<DWORD>(adr); // чтение
writeMemory<DWORD>(adr, value++); // инкремент и запись
```

Эти вызовы почти идентичны вызовам в Листинге 6-6 на странице 124; разница лишь в том, что им не нужно передавать дескриптор процесса в качестве аргумента, потому что они вызываются изнутри самого процесса.

Вы можете сделать этот метод еще более гибким, создав третью шаблонную функцию под названием `pointMemory()`, как показано

ниже:

---

```
template<typename T>
T* pointMemory(LPVOID adr) {
    return ((T*)adr);
}
```

Эта функция пропускает шаг разыменования при чтении памяти и просто возвращает вам указатель на данные. Оттуда вы можете как читать, так и записывать в память, разыменовывая этот указатель самостоятельно, например так:

```
DWORD* pValue = pointMemory<DWORD>(adr); // указатель
DWORD value = *pValue; // 'чтение'
(*pValue)++; // инкремент и 'запись'
```

С функцией `pointMemory()` на месте, можно устраниТЬ вызовы `readMemory()` и `writeMemory()`. Вам все равно нужно будет заранее найти `adr`, но затем код для чтения значения, его изменения и записи обратно станет гораздо проще для понимания.

## Обход ASLR во внедренной DLL

Аналогично, так как код уже внедрен, нет необходимости внедрять поток в игру, чтобы получить базовый адрес. Вместо этого можно просто вызвать `GetModuleHandle()` напрямую, например так:

---

```
DWORD newBase = (DWORD)GetModuleHandle(NULL);
```

Более быстрый способ получить базовый адрес — использовать FS-сегмент памяти игры, который является еще одной суперспособностью, доступной во внедренном коде. Этот сегмент памяти указывает на структуру, называемую *thread environment block* (TEB) (блок окружения потока), а 0x30 байт внутри ТЕВ указывает на *process environment block* (PEB) (блок окружения процесса). Эти структуры используются операционной системой и содержат массу данных о текущем потоке и текущем процессе, но нас интересует только базовый адрес главного модуля, который хранится по смещению 0x8 внутри РЕВ. Используя встроенный ассемблер, можно пройти по этим структурам и получить `newBase`, как показано ниже:

---

```
DWORD newBase;
__asm {
    MOV EAX, DWORD PTR FS:[0x30]
    MOV EAX, DWORD PTR DS:[EAX+0x8]
    MOV newBase, EAX
}
```

Первая команда сохраняет адрес РЕВ в ЕАХ, вторая команда считывает базовый адрес главного модуля и сохраняет его в ЕАХ. Последняя команда копирует значение ЕАХ в `newBase`.

## **Заключительные мысли**

В главе 6 я показал, как считывать память удаленно и как внедренная DLL может напрямую получать доступ к памяти игры, используя указатели. В этой главе было продемонстрировано, как внедрять различные типы кода — от чистого ассемблерного байткода до полноценных бинарных файлов C++. В следующей главе вы узнаете, насколько мощным может быть доступ к пространству памяти игры. Если вам показалось, что внедрение ассемблерного кода — это круто, вам точно понравится то, что можно сделать при внедрении C++ с управлением потоком выполнения.

Пример кода в этой главе содержит доказательства концепции для всего, что мы обсуждали. Если какие-то темы остались непонятными, вы можете изучить код, чтобы точно разобраться, что происходит, и увидеть все трюки в действии.

## 8) Манипулирование потоком управления в игре



игрой?

Заставить игру выполнять посторонний код — это, безусловно, мощно, но что, если бы вы могли изменить способ выполнения собственного кода

Что, если бы вы могли заставить игру обходить код, который рисует туман войны, обмануть её так, чтобы враги становились видимыми сквозь стены, или изменять аргументы, которые она передаёт в функции? Управление потоком выполнения позволяет вам делать именно это, давая возможность изменять то, что делает процесс, перехватывая выполнение кода и контролируя, изменяя или предотвращая его.

Существует множество способов манипулирования потоком выполнения процесса, но почти все они требуют от вас изменения ассемблерного кода процесса. В зависимости от ваших целей, вам нужно либо полностью удалить код из процесса (называется *NOPing*), либо заставить процесс перенаправлять выполнение на внедрённые функции (называется *hooking*). В начале этой главы вы узнаете о

*NOPing*, различных типах *hooking* и других техниках манипуляции потоком выполнения. После объяснения основ я покажу, как применял эти принципы к распространённым игровым библиотекам, таким как Adobe AIR и Direct3D.

Откройте каталог *GameHackingExamples/Chapter8\_ControlFlow* в ресурсных файлах этой книги, чтобы увидеть полный пример кода для следующего раздела "Перехват выполнения игры с помощью хука" на странице 153.

### **NOPing для удаления ненужного кода**

Глава 7 описала, как внедрять новый код в игру, но обратный

процесс — удаление кода из игры — тоже может быть полезен. Некоторые хаки требуют, чтобы определённые части оригинального кода игры не выполнялись, и для этого вам нужно избавиться от них. Один из способов удалить код из процесса игры — это *NOPing*, который заключается в перезаписи оригинального x86-кода инструкциями *NOP*.

## Когда применять NOP

Рассмотрим игру, в которой не отображаются полоски здоровья у замаскированных врагов. Замаскированных врагов трудно заметить, и в бою вы получили бы огромное преимущество, если бы могли хотя бы видеть их полоски здоровья. Код для отрисовки полос здоровья часто выглядит так, как в Листинге 8-1.

---

```
for (int i = 0; i < creatures.size(); i++) {  
    auto c = creatures[i];  
    if (c.isEnemy && c.isCloaked) continue;  
    drawHealthBar(c.healthBar);  
}
```

---

Листинг 8-1: Цикл из *drawCreatureHealthBarExample()* функции

При отрисовке полос здоровья игра с замаскированными существами может использовать цикл *for*, чтобы проверить, находятся ли существа в границах экрана и замаскированы ли они. Если враг не замаскирован, цикл вызывает некоторую функцию (*drawHealthBar()* в этом примере), чтобы отобразить полосу здоровья врага.

Имея этот исходный код, вы могли бы заставить игру отображать полоски здоровья даже у замаскированных врагов, просто удалив *if (c.isEnemy && c.isCloaked) continue;* из кода. Но, будучи хакером игр, у вас есть только машинный код, а не исходный код. При упрощении ассемблерный эквивалент Листинга 8-1 будет выглядеть примерно так: (далее следует псевдокод).

```

startOfLoop:           ; for
    MOV i, 0          ; int i = 0
    JMP condition     ; первый проход, пропускаем инкремент
increment:
    ADD i, 1          ; i++
condition:
    CMP i, creatures.Size(); i < creatures.size()
    JNB endOfLoop      ; выход из цикла, если i >= creatures.size()
body:
    MOV c, creatures[i] ; auto c = creatures[i]
    TEST c.isEnemy, c.isEnemy ; if c.isEnemy
    JZ drawHealthBar    ; нарисовать полосу здоровья, если c.isEnemy == false
    TEST c.isCloaked, c.isCloaked ; && c.isCloaked
    JZ drawHealthBar    ; нарисовать полосу здоровья, если c.isCloaked == false
    JMP increment       ; continue

drawHealthBar:
    CALL drawHealthBar(c.healthBar) ; drawHealthBar(c.healthBar)
    JMP increment                 ; continue
endOfLoop:

```

Чтобы заставить игру отображать все полоски здоровья врагов, независимо от маскировки, вам нужно удалить команду JMP increment ● ①, которая выполняется, когда c.isEnemy && c.isCloaked равно true. Однако в ассемблере замена ненужного кода инструкциями, которые ничего не делают, проще, чем удаление кода. Именно для этого существует команда NOP.

Поскольку NOP представляет собой один байт (0x90), можно заменить 2-байтовую команду JMP increment двумя командами NOP. Когда процессор достигает этих NOP-команд, он просто их пропускает и выполняет drawHealthBar(), даже если c.isEnemy && c.isCloaked равно true.

## Как использовать NOP

Первым шагом к замене части ассемблерного кода на NOP является обеспечение возможности записи в участок памяти, где расположен код. Возможно, что код на той же странице памяти продолжает выполняться, пока вы записываете NOP-команды.

Поэтому также важно убедиться, что память остаётся выполняемой. Этого можно добиться, установив защиту памяти в PAGE\_EXECUTE\_READWRITE.

После того как защита памяти изменена, можно записать NOP-команды и завершить операцию. Формально, оставлять память в режиме записи не опасно, но хорошей практикой считается восстановление исходной защиты после завершения работы.

Если у вас уже есть функции для работы с памятью и её защиты (как описано в **Главе 6**), можно написать функцию, аналогичную той, что приведена в **Листинге 8-2**, для записи NOP-команд в память игры. (Следуйте за кодом в файле **NOPExample.cpp**).

---

```
template<int SIZE>
void writeNop(DWORD address)
{
    auto oldProtection =
        protectMemory<BYTE[SIZE]>(address, PAGE_EXECUTE_READWRITE);

    for (int i = 0; i < SIZE; i++)
        writeMemory<BYTE>(address + i, 0x90);

    protectMemory<BYTE[SIZE]>(address, oldProtection);
}
```

---

Листинг 8-2: Правильный NOP, с защитой памяти

В этом примере функция `writeNop()` устанавливает соответствующую защиту памяти, записывает количество NOP-команд, равное `SIZE`, и повторно применяет исходный уровень защиты памяти.

Функция `writeNop()` принимает количество инструкций NOP в качестве параметра шаблона, так как функции работы с памятью требуют правильно заданного типа во время компиляции. Передача целого числа `SIZE` указывает этим функциям работать с типом `BYTE[SIZE]` на этапе компиляции.

Чтобы указать динамический размер во время выполнения, просто уберите цикл и вместо этого вызовите `protectMemory<BYTE>` и передайте `address` и `address + SIZE` в качестве аргументов. Пока размер не превышает одной страницы (а на самом деле, вам не следует заменять NOP-ами целую страницу), это обеспечит правильную защиту памяти, даже если она находится на границе страницы.

Вызовите эту функцию с адресом, в котором вы хотите разместить NOP-ы, и количеством NOP-команд:

---

```
writeNop<2>(0xDEADBEEF);
```

---

Имейте в виду, что количество NOP-команд должно совпадать с размером (в байтах) удаляемой команды. Этот вызов `writeNop()` записывает две NOP-команды по адресу `0xDEADBEEF`.

## ПРАКТИКА NOP-ИНГА

Если вы ещё этого не сделали, откройте `NOPExample.cpp` в примерах кода этой главы и поэкспериментируйте с ним. Вы найдёте рабочую реализацию функции `writeNop()` и интересную

функцию `getAddressForNOP()`, которая сканирует память примера программы, чтобы определить, где должна быть размещена NOP-команда.

Чтобы увидеть NOP-команду в действии, запустите `NOPApplication` в отладчике Visual Studio, установив точки останова в начале и в конце функции `writeNop()`.

Когда сработает первая точка останова, нажмите **ALT+8**, чтобы открыть окно дизассемблера, введите `address` в поле ввода и нажмите **ENTER**. Это приведёт вас к целевому адресу NOP, где вы увидите полностью сохранённый код.

Нажмите **F5**, чтобы продолжить выполнение, что вызовет вторую точку останова после того, как приложение разместит NOP-ы. Затем снова вернитесь к `address` в дизассемблере и убедитесь, что код был заменён NOP-ами.

Вы можете изменить этот код для других интересных вещей. Например, попробуйте размещать NOP-ы на проверках условий вместо удаления `JMP` или даже изменять типы переходов или их направления.

Эти и другие альтернативные подходы тоже работают, но имейте в виду, что они могут создавать больше возможностей для ошибок по сравнению с простой заменой одиночного `JMP` на NOP. При изменении чужого кода делайте как можно меньше изменений, чтобы минимизировать вероятность ошибок.

## Перехват для перенаправления выполнения кода игры

До сих пор я показывал вам, как манипулировать играми, добавляя в них код, захватывая их потоки, создавая новые потоки и даже удаляя существующий код из их потока выполнения. Эти методы сами по себе очень мощные, но при объединении они образуют ещё более действенный метод манипуляции, называемый перехватом (*hooking*).

Перехват позволяет вам перехватывать определённые ветвления выполнения и перенаправлять их на внедрённый код, который вы написали, чтобы диктовать, что должна делать игра дальше. Существует множество видов перехвата. В этом разделе я научу вас четырём из самых мощных методов перехвата в взломе игр:

- перехват вызовов (*call hooking*),

- перехват таблицы виртуальных функций (*virtual function table hooking*),

- перехват таблицы импорта (*import address table hooking*),

- перехват переходов (*jmp hooking*).

## Перехват вызовов (Call Hooking)

Перехват вызова (*call hook*) напрямую изменяет цель операции `CALL`, заставляя её указывать на новый участок кода. В x86-ассемблере существует несколько вариаций `CALL`, но в большинстве случаев перехват применяется только к одному типу

– near call, который принимает непосредственный адрес в качестве операнда.

## Работа с Near Call в памяти

В ассемблере инструкция near call выглядит так:

---

CALL 0x0BADFOOD

Эта near call представлена байтом 0xE8, поэтому вы могли бы предположить, что она хранится в памяти вот так:

---

0xE8 0x0BADFOOD

Или, если разбить её на отдельные байты и поменять порядок байтов для учёта эндианности:

---

0xE8 0x0D 0xFO 0xAD 0x0B

Но анатомия near call в памяти не так проста. Вместо того чтобы хранить абсолютный адрес вызываемой функции, near call хранит смещение от адреса, расположенного непосредственно после вызова.

Поскольку near call занимает 5 байтов, адрес, на который указывает вызов, находится на 5 байтов дальше в памяти. Учитывая это, хранимый адрес можно вычислить следующим образом:

---

calleeAddress - (callAddress + 5)

Если CALL 0x0BADFOOD находится по адресу 0xDEADBEEF в памяти, то значение после 0xE8 будет следующим:

---

0x0BADFOOD - (0xDEADBEEF + 5) = 0x2D003119

В памяти эта инструкция CALL выглядит так:

---

0xE8 0x19 0x31 0x00 0x2D

Чтобы перехватить ближний вызов, вам сначала нужно изменить смещение, следующее за 0xE8 (то есть, в формате little-endian 0x19 0x31 0x00 0x2D), чтобы оно указывало на вашего нового вызываемого.

## Перехват ближнего вызова

Следуя тем же правилам защиты памяти, которые показаны в Листинге 8-2, можно перехватить ближний вызов следующим образом (следуйте инструкциям, открыв CallHookExample.cpp):

---

```
DWORD callHook(DWORD hookAt, DWORD newFunc)
{
    DWORD newOffset = newFunc - hookAt - 5;

    auto oldProtection =
        protectMemory<DWORD>(hookAt + 1, PAGE_EXECUTE_READWRITE);

    DWORD originalOffset = readMemory<DWORD>(hookAt + 1);
    writeMemory<DWORD>(hookAt + 1, newOffset);
    protectMemory<DWORD>(hookAt + 1, oldProtection);

❷    return originalOffset + hookAt + 5;
}
```

---

Эта функция принимает в качестве аргументов адрес инструкции CALL, которую нужно перехватить (hookAt), и адрес, на который следует перенаправить выполнение (newFunc). Затем она использует эти аргументы для вычисления смещения, необходимого для вызова кода по адресу, который содержит newFunc. После того как вы примените правильную защиту памяти, функция callHook() запишет новое смещение в память по hookAt + 1 ❶, восстановит старую защиту памяти, вычислит адрес исходного вызова ❷ и вернёт это значение вызывающей стороне.

Вот как можно использовать такую функцию в игровом хакинге:

---

```
DWORD origFunc = callHook(0xDEADBEEF, (DWORD)&someNewFunction);
```

---

Этот вызов перехватывает ближний вызов на 0x0BADF00D по адресу 0xDEADBEEF и перенаправляет его на адрес someNewFunction, то есть на код, который выполнит ваш хак. После вызова origFunc будет содержать 0x0BADF00D.

## Очистка стека

Новый вызываемый код также должен правильно обрабатывать стек, сохранять регистры и передавать корректные возвращаемые значения. Как минимум, это означает, что ваша заменяющая функция должна соответствовать исходной функции игры как по соглашению о вызовах, так и по количеству аргументов.

Допустим, вот как выглядел исходный вызов функции в ассемблере:

---

```
PUSH 1
PUSH 456
PUSH 321
CALL 0x0BADF00D
ADD ESP, 0x0C
```

---

Вы можете определить, что эта функция использует соглашение о вызовах C++ `__cdecl`, поскольку стек очищается вызывающей

стороной. Кроме того, 0x0C байтов, очищаемых из стека, показывают, что у функции три аргумента, что можно вычислить следующим образом:

$$\frac{0x0C}{\text{sizeof(DWORD)}} = 3$$

Конечно, можно также определить количество аргументов, проверяя, сколько значений помещается в стек: есть три команды PUSH, по одной на каждый аргумент.

## Написание перехвата вызова

В любом случае новый вызываемый код, someNewFunction, должен следовать соглашению `_cdecl` и иметь три аргумента. Вот пример скелета новой вызываемой функции:

```
DWORD __cdecl someNewFunction(DWORD arg1, DWORD arg2, DWORD arg3)
{
}
```

В Visual Studio программы на C++ по умолчанию используют соглашение о вызовах `_cdecl`, поэтому технически вы могли бы опустить его в определении функции. Однако, как я выяснил, лучше явно указывать его, чтобы вы привыкли быть точными.

Также помните, что если вызывающая сторона ожидает возврата значения, то тип возвращаемого значения вашей функции должен совпадать. В этом примере предполагается, что тип возвращаемого значения всегда `DWORD` или меньше. Поскольку возвращаемые типы такого размера передаются обратно через `EAX`, в дальнейших примерах также будет использоваться тип `DWORD`.

В большинстве случаев перехват завершает свою работу вызовом оригинальной функции и передачей её возвращаемого значения вызывающей стороне. Вот как всё это может выглядеть вместе:

```
typedef DWORD (_cdecl _origFunc)(DWORD arg1, DWORD arg2, DWORD arg3);

_origFunc* originalFunction =
    (_origFunc*)hookCall(0xDEADBEEF, (DWORD)&someNewFunction);

DWORD __cdecl someNewFunction(DWORD arg1, DWORD arg2, DWORD arg3)
{
    return originalFunction(arg1, arg2, arg3);
}
```

Этот пример использует `typedef` для объявления типа, представляющего оригинальный прототип функции, и создаёт указатель с этим типом на оригинальную функцию. Затем `someNewFunction()` использует этот указатель для вызова оригинальной функции с оригинальными аргументами и передаёт

возвращённое значение обратно вызывающему коду.

На данный момент `someNewFunction()` просто возвращает управление оригинальной функции. Но внутри вызова `someNewFunction()` вы можете делать что угодно. Можно изменять параметры, передаваемые в оригинальную функцию, или перехватывать и сохранять интересные параметры для дальнейшего использования. Если вы знаете, что вызывающая сторона не ожидает возвращаемого значения (или если вы умеете подделывать возвращаемое значение), то можно даже забыть про оригинальную функцию и полностью заменить, скопировать или улучшить её функциональность внутри нового вызываемого кода.

Как только вы освоите этот навык, вы сможете добавлять собственный нативный код на С или С++ в любую часть игры, какую захотите.

## Перехват таблицы виртуальных функций (VF Table Hooking)

В отличие от перехвата вызовов (call hooks), перехват таблицы виртуальных функций (VF table hooks) не изменяет код на ассемблере. Вместо этого он изменяет адреса функций, хранящиеся в таблицах виртуальных функций (VF) классов. (Если вам нужно освежить в памяти, что такое таблицы VF, см. раздел «Класс с виртуальными функциями» на странице 75.)

Все экземпляры одного и того же типа класса используют одну и ту же статическую таблицу VF, поэтому перехват таблицы VF будет перехватывать все вызовы функции-члена, независимо от того, из какого экземпляра класса игра вызывает эту функцию. Это может быть одновременно мощным и сложным инструментом.

### ПРАВДА О ТАБЛИЦАХ VF

Чтобы упростить объяснение, я немного склонялся, когда сказал, что перехват таблицы VF может перехватывать все вызовы функции. На самом деле таблица VF просматривается только тогда, когда виртуальная функция вызывается таким образом, который оставляет компилятору некоторую неоднозначность типа.

Например, таблица VF будет просматриваться, если функция вызывается в формате `inst->function()`. Однако таблица VF не будет просматриваться, если виртуальная функция вызывается так, что компилятор точно знает тип, например, `inst.function()` или аналогичным вызовом, так как компилятор уже знает адрес функции.

С другой стороны, вызов `inst.function()` из области видимости, где `inst` передаётся как ссылка (reference), приведёт к обходу таблицы VF.

Перед тем как пытаться перехватить таблицу VF, убедитесь, что функции, которые вы хотите перехватить, действительно имеют неоднозначность типа.

## Написание перехвата таблицы VF

Прежде чем мы углубимся в тему добавления перехвата таблицы VF, нам нужно снова обсудить эти надоедливые соглашения о вызовах. Таблицы VF используются экземплярами классов для вызова виртуальных функций-членов, и все функции-члены в них используют соглашение вызовов `_thiscall`.

## Написание перехвата таблицы VF

Прежде чем углубляться в то, как добавить перехват таблицы VF, нам нужно снова обсудить эти **соглашения о вызовах**. Таблицы VF используются экземплярами классов для вызова виртуальных функций-членов, и **все функции-члены** будут использовать соглашение вызова `_thiscall`.

Название `_thiscall` происходит от **указателя this**, который функции-члены используют для ссылки на активный экземпляр класса. Таким образом, функции-члены получают `this` в виде **псевдопараметра в регистре ECX**.

Можно подогнать **прототип `_thiscall`**, объявив специальный **класс-контейнер** для всех обработчиков `_thiscall`-хуков. Однако я не предполагаю этот метод. Вместо этого мне удобнее управлять данными, используя **встроенный ассемблер**.

Давайте разберём, как управлять данными при добавлении VF-хука в класс, который выглядит так:

---

```
class someBaseClass {
public:
    virtual DWORD someFunction(DWORD arg1) {}
};

class someClass : public someBaseClass {
public:
    virtual DWORD someFunction(DWORD arg1) {}
};
```

---

Класс `someBaseClass` содержит только один член — публичную виртуальную функцию.

Класс `someClass` унаследован от `someBaseClass` и переопределяет метод `someBaseClass::someFunction`.

Чтобы перехватить `someClass::someFunction`, необходимо повторить этот прототип в VF-хуке (как показано в Листинге 8-3 в файле проекта `VFHookExample.cpp`).

---

```
DWORD __stdcall someNewVFFunction(DWORD arg1)
{
①    static DWORD _this;
    __asm MOV _this, ECX
}
```

---

Листинг 8-3: Начало крючка таблицы VF

Эта функция работает как перехватчик, потому что `_thiscall` отличается от `_stdcall` только тем, что первый получает `this` в ECX.

Чтобы устраниТЬ это небольшое различие, функция-обработчик использует встроенный ассемблер (обозначенный как `_asm`), чтобы скопировать `this` из ECX в статическую переменную ①.

Так как статическая переменная фактически инициализируется как глобальная, единственный код, который выполняется перед `MOV _this, ECX`, — это код, который устанавливает стековый фрейм. Этот код никогда не затрагивает ECX, что гарантирует, что ECX содержит правильное значение при выполнении ассемблерного кода.

#### NOTE

Если несколько потоков начнут вызывать одну и ту же VF-функцию, перехватчик `someNewVFFunction()` сломается, потому что `_this` может быть изменён одним вызовом, в то время как всё ещё используется другим. Лично я никогда не сталкивался с этой проблемой, так как игры обычно не создают несколько экземпляров критически важных классов между потоками. Однако эффективное решение — хранить `_this` в потоковой локальной памяти (*thread local storage*), обеспечивая, что каждый поток будет иметь свою собственную копию.

Перед возвратом перехватчик VF-таблицы должен восстановить ECX, чтобы соответствовать соглашению `_thiscall`.

```
DWORD __stdcall someNewVFFunction(DWORD arg1)
{
    static DWORD _this;
    _asm MOV _this, ECX

    // do game modifying stuff here

    _asm ①MOV ECX, _this
}
```

После выполнения игрового хак-кода эта версия `someNewVFFunction()` восстанавливает ECX ① с обратной версией первого MOV, который был в Листинге 8-3.

В отличие от `_cdecl`-функций, не следует вызывать функции с соглашением `_thiscall` из чистого C++ с использованием только указателя на функцию и `typedef` (как при перехвате вызовов `call`).

При вызове оригинальной функции из перехватчика VF-таблицы необходимо использовать встроенный ассемблер.

✓ Это единственный способ гарантировать корректную передачу данных (в частности, `_this`).

Вот как продолжается выполнение перехвата `someNewVFFunction()`:

---

```
DWORD __stdcall someNewVFFunction(DWORD arg1)
{
    static DWORD _this, _ret;
    __asm MOV _this, ECX

    // do pre-call stuff here

    __asm {
        PUSH arg1
        MOV ECX, _this
   ❶     CALL [originalVFFunction]
   ❷     MOV _ret, EAX
    }

    // do post-call stuff here

    __asm MOV ECX, _this
    return _ret;
}
```

---

Теперь `someNewVFFunction()` сохраняет `this` в переменную `_this`, выполняет некоторый код, затем вызывает оригинальную игровую функцию ❶, которая перехвачена. Затем сохраняет её возвращаемое значение в `_ret` ❷, выполняет ещё дополнительный код, восстанавливает `ECX` ❸ и возвращает значение, сохранённое в `_ret`.

Вызываемая функция очищает стек для `__thiscall`-вызовов, поэтому, в отличие от перехвата `call`, переданный аргумент не нужно удалять вручную.

**НО ТЕ**

*Если нужно удалить один переданный в стек аргумент в любой момент, используйте ассемблерную инструкцию: `ADD ESP, 0x4`, так как один аргумент занимает 4 байта.*

## Использование перехвата VF-таблицы

С установленным соглашением вызова и скелетным обработчиком на месте, пришло время перейти к интересной части: фактическому использованию перехвата VF-таблицы.

Поскольку указатель на VF-таблицу класса является первым членом каждого экземпляра класса, установка перехвата VF-таблицы требует только адреса экземпляра класса и индекса функции, которую нужно перехватить. Используя эти два элемента, можно написать совсем небольшой код для установки перехвата.

---

```
DWORD hookVF(DWORD classInst, DWORD funcIndex, DWORD newFunc)
{
    DWORD VFTable = ❶readMemory<DWORD>(classInst);
    DWORD hookAt = VFTable + funcIndex * sizeof(DWORD);

    auto oldProtection =
        protectMemory<DWORD>(hookAt, PAGE_READWRITE);
    DWORD originalFunc = readMemory<DWORD>(hookAt);
    writeMemory<DWORD>(hookAt, newFunc);
    protectMemory<DWORD>(hookAt, oldProtection);

    return originalFunc;
}
```

---

Функция `hookVF()` находит VF-таблицу, считывая первый член экземпляра класса ❶ и сохраняя его в `VFTable`.

Так как VF-таблица — это просто массив адресов размера `DWORD`, этот код находит адрес функции, умножая индекс функции (`funcIndex`) на размер `DWORD` (4 байта) и прибавляя результат к адресу VF-таблицы.

После этого `hookVF()` выполняет следующие шаги:

Делает память доступной для записи.

Сохраняет оригинальный адрес функции.

Записывает новый адрес функции.

Восстанавливает оригинальные права доступа памяти.

Обычно перехват VF-таблицы выполняется в экземпляре класса, созданном игрой.

Чтобы установить перехват VF-таблицы, вызывается функция `hookVF()`.

---

```
DWORD origVFFunction =
    hookVF(classInstAddr, 0, (DWORD)&someNewVFFunction);
```

---

Как обычно, вам нужно заранее определить `classInstAddr` и аргумент `funcIndex`. Существуют действительно полезные случаи, когда перехваты VF-таблицы оказываются незаменимыми, но порой бывает довольно трудно найти нужные указатели на класс и смещения членов. Учитывая это, вместо того чтобы показывать надуманные примеры, я вернусь к теме перехватов VF-таблицы в разделе **"Applying Jump Hooks and VF Hooks to Direct3D"** на странице 175, когда расскажу об остальных видах перехватов.

Если вы хотите попрактиковаться с перехватами VF, прежде чем читать дальше, добавьте новые виртуальные функции в классы из ресурсных файлов этой книги и потренируйтесь их перехватывать. Вы даже можете создать второй класс, который наследуется от `someBaseClass`, и поставить перехват на виртуальную функцию, чтобы показать, как можно иметь два совершенно отдельных перехвата VF в двух классах, наследующих один и тот же базовый класс.

## IAT Hooking

Перехваты IAT фактически заменяют адреса функций в определённом типе VF-таблицы, называемой **таблицей импорта** (*import address table, IAT*). Когда код в процессе содержит **IAT** в своём заголовке PE, таблица IAT модуля хранит список всех адресов, от которых этот модуль зависит, а также список функций, к которым он может обращаться. Таблица IAT модуля используется как таблица поиска (lookup table) для API-функций, которые нужно вызывать.

Когда модуль загружается, загружаются и его зависимости. Загрузка зависимостей происходит рекурсивно: система продолжает загрузку, пока все зависимости не будут полностью разрешены. Как только все зависимости загружены, обновляются конечные базовые адреса в памяти. Затем модуль вызывает функцию из своей IAT, и эта функция вызывается путём разрешения (resolving) адреса из IAT.

## Оплата за переносимость (Paying for Portability)

Адреса функций всегда разрешаются из IAT во время выполнения, поэтому их перехват (hooking) похож на перехват VF-таблиц. Поскольку модуль в процессе ссылается на них по реальным именам, нет необходимости в инжиниринге или сканировании памяти; достаточно знать имя API, которое вы хотите перехватить, и можно перехватывать его. Более того, перехваты IAT позволяют с лёгкостью перехватывать вызовы Windows API, не требуя, чтобы пользователь вызывал VirtualProtect.

Но при этом перехват IAT влияет только на вызовы из основного модуля игры. Если другой модуль импортирует ту же функцию, ваш перехват не сработает для него. Поэтому, если игра загружает несколько DLL, которые тоже вызывают эту функцию напрямую, вы должны перехватывать IAT в **каждом** модуле, который использует данную функцию.

Код, необходимый для перехвата IAT, ничуть не сложнее, чем для перехвата VF-таблицы: нужно лишь найти запись в таблице импорта PE-заголовка модуля, соответствующую нужной функции. Поскольку PE-заголовок — это структурированный формат, вы можете перебрать записи импорта и найти ту, которая вам нужна, как показано в Листинге 8-4 (доступен в файле **IATHookExample.cpp** проекта).

---

```
DWORD baseAddr = (DWORD)GetModuleHandle(NULL);
```

---

Листинг 8-4: Получение базового адреса модуля

Когда вы найдёте базовый адрес, вам нужно удостовериться, что PE-заголовок действителен. Эта проверка может быть очень важна, так как некоторые игры пытаются предотвратить подобные перехваты, изменяя несущественные части своего PE-

заголовка после загрузки. Действительный PE-заголовок имеет в начале DOS-заголовок, указывающий на то, что файл является DOS MZ-исполняемым; DOS-заголовок идентифицируется магическим значением 0x5A4D. Элемент DOS-заголовка, называемый `e_lfanew`, затем указывает на «опциональный заголовок» (optional header), который содержит такие значения, как размер кода, номер версии и так далее, и идентифицируется магическим значением 0x10B.

В Windows API существуют структуры PE с названиями `IMAGE_DOS_HEADER` и `IMAGE_OPTIONAL_HEADER`, которые соответствуют DOS-заголовку и опциональному заголовку соответственно. Вы можете использовать их для проверки PE-заголовка с помощью кода, подобного приведённому в Листинге 8-5.

---

```
auto dosHeader = pointMemory<IMAGE_DOS_HEADER>(baseAddr);
if (dosHeader->e_magic != 0x5A4D)
    return 0;

auto optHeader =
    pointMemory<IMAGE_OPTIONAL_HEADER>(baseAddr + dosHeader->e_lfanew + 24);
if (optHeader->Magic != 0x10B)
    return 0;
```

---

Листинг 8-5: Убедитесь, что DOS и дополнительные заголовки действительны

Вызовы к `pointMemory()` создают указатели на два заголовка, которые нам нужны. Если условие `if (e_lfanew == 0)`, значит соответствующий заголовок не имеет правильного магического значения, а значит заголовок повреждён или недействителен. Ссылки на таблицу импорта (import table) берутся из **опционального заголовка** PE, а значит ассемблерные инструкции не нужны, чтобы найти нужную область. Это также означает, что перезапись **PE-заголовка** после загрузки может означать, что у модуля вообще нет импорта, что является способом защиты от перехватов IAT, но в большинстве игр этого не происходит.

Чтобы учесть такую возможность, ваш код должен убедиться, что в игре действительно есть **IAT** перед тем, как делать то, что показано в Листинге 8-5:

---

```
auto IAT = optHeader->DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
if (IAT.Size == 0 || IAT.VirtualAddress == 0)
    return 0;
```

---

Листинг 8-6: Проверка того, что IAT действительно существует

Заголовок PE содержит множество секций, в которых хранится информация об исполняемом файле: коды возвратов, таблицы релокаций и так далее. Часть кода в этих секциях называется **.idata**, что означает, что в ней хранятся многие структуры, связанные с импортом.

DataDirectory внутри **IMAGE\_OPTIONAL\_HEADER** — это массив, описывающий размер и виртуальный адрес каждой директории в **PE**. Windows API определяет константу **IMAGE\_DIRECTORY\_ENTRY\_IMPORT**, указывающую на индекс в DataDirectory, по которому хранится заголовок **IAT**.

Таким образом, этот код использует `optHeader->DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT]`, чтобы найти заголовок **IAT** и проверить, что поля `Size` и `VirtualAddress` не равны нулю, подтверждая тем самым, что IAT действительно существует.

## Проход по IAT

Как только вы убедились, что **IAT** не повреждён, вы можете начать его обход, и именно здесь перехват **IAT** начинает усложняться. **IAT** — это массив структур, называемых **import descriptors**. На один **import descriptor** приходится массив структур, называемых **thunks**, и каждый **import descriptor** ссылается на массив **thunks**, причём каждый **thunk** представляет собой функцию, импортированную из внешней зависимости.

К счастью, Windows API предоставляет структуры **IMAGE\_IMPORT\_DESCRIPTOR** и **IMAGE\_THUNK\_DATA** для работы с **import descriptors** и **thunks** соответственно. Наличие этих структур избавляет вас от необходимости самому изобретать велосипед, однако это не упрощает код для обхода IAT.

Чтобы понять, о чём речь, посмотрите **Листинг 8-7**, который основан на **Листингах 8-4 – 8-6**.

---

```
auto impDesc =
    pointMemory<IMAGE_IMPORT_DESCRIPTOR>(❶baseAddr + IAT.VirtualAddress);

❷ while (impDesc->FirstThunk) {
❸     auto thunkData =
        pointMemory<IMAGE_THUNK_DATA>(baseAddr + impDesc->OriginalFirstThunk);
        int n = 0;
❹     while (thunkData->u1.Function) {
            // the hook happens in here
            n++;
            thunkData++;
        }
        impDesc++;
}
```

---

Листинг 8-7: Итерация над IAT для поиска функции

Учитывая, что **import descriptors** хранятся относительно начала **PE-заголовка**, этот код добавляет **базовый адрес модуля** к виртуальному адресу, найденному в заголовке IAT ①, создавая указатель `impDesc`, указывающий на первый **import descriptor** модуля. Импортные дескрипторы (**import descriptors**) хранятся в последовательном массиве, и дескриптор с `FirstThunk`, равным `NULL`, указывает на конец массива. Понимая это, код использует цикл **while** ②, который продолжается до тех пор, пока `impDesc->FirstThunk` не будет равен `NULL`, увеличивая `impDesc` на единицу (через `impDesc++`) при каждом проходе.

Для каждого **import descriptor** код создаёт указатель под названием `thunkData` ③, который указывает на первый **thunk** внутри этого дескриптора. Используя знакомый цикл, код перебирает **thunks** ④, пока не найдёт тот, у которого `Function` равно `NULL`. В этом цикле также используется `int n`, чтобы отслеживать текущий индекс **thunk**, так как индекс важен при размещении перехвата (hook).

## Размещение IAT-хука

На этом этапе установка хука сводится к поиску правильного имени функции и замене её адреса. Имя можно найти внутри вложенного **while**-цикла, как показано в **Листинге 8-8**.

---

```
char* importFunctionName =  
    pointMemory<char>(baseAddr + (DWORD)thunkData->u1.AddressOfData + 2);
```

---

Листинг 8-8: Поиск имени функции

Имя функции для каждого **thunk** хранится в `thunkData->u1.AddressOfData + 2` байта внутри модуля, поэтому можно добавить это значение к **базовому адресу модуля**, чтобы определить **расположение имени функции в памяти**.

После получения указателя на имя функции используйте `strcmp()` для проверки, является ли оно целевой функцией, как показано ниже:

```
if (strcmp(importFuncName, funcName) == 0) {  
    // Здесь выполняется финальный шаг  
}
```

После того как вы нашли целевую функцию по её имени, просто замените её адрес на адрес вашей собственной функции. В отличие от имён функций, адреса функций хранятся в массиве в начале каждого import-дескриптора. Используя цикл `thunk`, теперь можно наконец установить хук, как показано в Листинге 8-9.

---

```
auto vfTable = pointMemory<DWORD>(baseAddr + impDesc->FirstThunk);
DWORD original = vfTable[n];

❶ auto oldProtection = protectMemory<DWORD>((DWORD)&vfTable[n], PAGE_READWRITE);
❷ vfTable[n] = newFunc;
protectMemory<DWORD>((DWORD)&vfTable[n], oldProtection);
```

---

Листинг 8-9: Нахождение адреса функции

Этот код находит таблицу VF для текущего дескриптора, добавляя адрес первого thunk'a к базовому адресу модуля. Таблица VF представляет собой массив адресов функций, поэтому код использует переменную n как индекс для поиска адреса целевой функции.

Как только адрес найден, код из Листинга 8-9 работает точно так же, как типичный VF-хук: он сохраняет оригинальный адрес функции, устанавливает защиту индекса n в таблице VF на PAGE\_READWRITE ❶, вставляет новый адрес функции в таблицу VF ❷, а затем восстанавливает старую защиту.

Если объединить код из Листингов 8-4 по 8-9, финальная функция IAT-хука будет выглядеть как в Листинге 8-10.

```
DWORD hookIAT(const char* funcName, DWORD newFunc)
{
    DWORD baseAddr = (DWORD)GetModuleHandle(NULL);

    auto dosHeader = pointMemory<IMAGE_DOS_HEADER>(baseAddr);
    if (dosHeader->e_magic != 0x5A4D)
        return 0;

    auto optHeader =
        pointMemory<IMAGE_OPTIONAL_HEADER>(baseAddr + dosHeader->e_lfanew + 24);
    if (optHeader->Magic != 0x10B)
        return 0;

    auto IAT =
        optHeader->DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
    if (IAT.Size == 0 || IAT.VirtualAddress == 0)
        return 0;

    auto impDesc =
        pointMemory<IMAGE_IMPORT_DESCRIPTOR>(baseAddr + IAT.VirtualAddress);

    while (impDesc->FirstThunk) {
        auto thunkData =
            pointMemory<IMAGE_THUNK_DATA>(baseAddr + impDesc->OriginalFirstThunk);
        int n = 0;
        while (thunkData->u1.Function) {
            char* importFuncName = pointMemory<char>
                (baseAddr + (DWORD)thunkData->u1.AddressOfData + 2);
            if (strcmp(importFuncName, funcName) == 0) {
                auto vftable = pointMemory<DWORD>(baseAddr + impDesc->FirstThunk);
                DWORD original = vftable[n];
                auto oldProtection =
                    protectMemory<DWORD>((DWORD)&vftable[n], PAGE_READWRITE);
                vftable[n] = newFunc;
                protectMemory<DWORD>((DWORD)&vftable[n], oldProtection);
                return original;
            }
            n++;
            thunkData++;
        }
        impDesc++;
    }
}
```

Листинг 8-10: Полная функция подключения IAT

Это самый сложный код, который мы собрали на данный момент, и его довольно трудно читать, когда он умещен на страницу. Если вы еще не разобрались, что он делает, вам стоит изучить пример кода из файлов ресурсов этой книги перед тем, как продолжить.

## Использование IAT-хука для синхронизации с потоком игры

С кодом из Листинга 8-10 перехват любой Windows API-функции становится простым – нужно всего лишь знать ее имя и правильный прототип. Функция `Sleep()` – это распространенный API, который перехватывают в гейм-хакинге, так как боты могут использовать хук `Sleep()` для синхронизации потока с основным игровым циклом.

## СИНХРОНИЗАЦИЯ С ПОТОКОМ ИГРЫ

Ваш внедренный код неизбежно должен синхронизироваться с основным потоком игры, иначе он не будет работать.

Когда вы читаете или записываете данные больше 4 байтов, рассинхронизация позволяет игре читать или записывать эти же данные одновременно с вами.

Вы будете "наступать игре на ноги", и наоборот, что приведет к гонкам потоков и порче данных.

Аналогично, если вы пытаетесь вызвать игровую функцию из своего потока, вы рискуете "уронить" игру, если эта функция не потокобезопасна.

Так как IAT-хуки — это потокобезопасные модификации PE-заголовка, их можно размещать из любого потока.

Перехватывая функцию, которая вызывается до или после главного игрового цикла, можно эффективно синхронизироваться с основным потоком игры.

Все, что нужно сделать, — установить хук и выполнить потокозависимый код в его обработчике.

Один из способов использовать hookIAT() для перехвата API Sleep()

```
VOID WINAPI newSleepFunction(DWORD ms)
{
    // Выполнять действия, зависящие от потока
    originalSleep(ms);
}

typedef VOID (WINAPI _origSleep)(DWORD ms);
_origSleep* originalSleep =
(_origSleep*)hookIAT("Sleep", (DWORD)&newSleepFunction);
```

## Почему это работает?

В конце главного игрового цикла игра может вызвать Sleep(), чтобы приостановить выполнение до отрисовки следующего кадра. Так как поток спит, можно безопасно выполнять любые действия без риска проблем синхронизации.

Некоторые игры не используют Sleep(), либо вызывают его из нескольких потоков, что требует другого метода перехвата.

Альтернативный вариант — перехватить API PeekMessageA(), так как игры часто вызывают эту функцию в главном цикле в ожидании ввода.

В этом случае бот может выполнять зависящие от потока операции в обработчике PeekMessageA() в главном потоке игры.

Также можно перехватить send() и recv() API, чтобы

анализировать сетевые пакеты.

## Перехват прыжком (Jump Hooking)

Перехват прыжком (Jump Hooking) позволяет перехватывать код даже там, где нет ветвлений. Вместо модификации существующего кода \*\*перехват заменяет его на безусловный jmp в так называемую трамплин-функцию (trampoline function).

Когда выполняется jmp, трамплин-функция:

Сохраняет все текущие регистры и флаги.

Вызывает вашу пользовательскую callback-функцию.

Восстанавливает регистры и флаги.

Исполняет заменённые инструкции.

Возвращается обратно в код сразу после хука.

Этот процесс показан на рисунке 8-1.

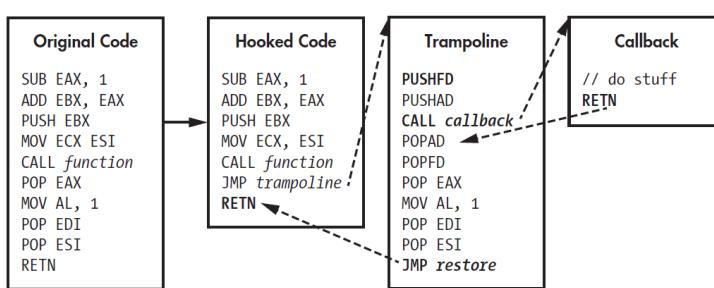


Рисунок 8-1: Крючок для прыжков

Оригинальный код показывает пример некоторого немодифицированного ассемблера, который можно найти в игре, а перехваченный код показывает, как этот ассемблер может выглядеть после перехвата прыжковым хуком. Блок трамплина показывает пример трамплин-функции на ассемблере, а блок обратного вызова представляет код, который вы пытаетесь выполнить через хук. В оригинальном коде ассемблер выполняется сверху вниз. В перехваченном коде, чтобы перейти от инструкции SUB EAX, 1 к инструкции RETN, выполнение должно следовать пути, показанному пунктирными стрелками.

### NOTE

Если ваш код обратного вызова прост, его можно встроить в трамплин вместо отдельного вызова. Также не всегда обязательно сохранять и восстанавливать регистры и флаги, но это хорошая практика.

## Размещение прыжка

Байтовый код безусловного перехода напоминает таковой у ближнего вызова, но первый байт – это 0xE9, а не 0xE8. (См. «Работа с ближними вызовами в памяти» на странице 153 для освежения в памяти.) В **Рисунке 8-1** безусловный прыжок JMP trampoline заменяет следующие четыре операции:

---

```
POP EAX
MOV AL, 1
POP EDI
POP ESI
```

---

В этом случае вам нужно заменить несколько последовательных операций, чтобы вместить 5-байтовый размер безусловного перехода. Вы можете столкнуться с ситуациями, когда размер операции (или операций), которую заменяют, превышает 5 байтов. В таких случаях оставшиеся байты заменяются NOP-инструкциями.

Теперь давайте посмотрим, как заменить эти операции. В **Листинге 8-11** показан код для установки прыжкового хука.

```
DWORD hookWithJump(DWORD hookAt, DWORD newFunc, int size)
{
    if (size > 12) // не должно потребоваться заменять более 12 байтов
        return 0;

    ❶ DWORD newOffset = newFunc - hookAt - 5;

    auto oldProtection =
        protectMemory<DWORD[3]>(hookAt + 1, PAGE_EXECUTE_READWRITE);

    ❷ writeMemory<BYTE>(hookAt, 0xE9);

    ❸ writeMemory<DWORD>(hookAt + 1, newOffset);
    for (unsigned int i = 5; i < size; i++)
        writeMemory<BYTE>(hookAt + i, 0x90);

    protectMemory<DWORD[3]>(hookAt + 1, oldProtection);

    return hookAt + 5;
}
```



Листинг 8-11: Как установить крючок для прыжков

Эта функция принимает адрес, который нужно перехватить, адрес функции обратного вызова и размер памяти, которую нужно перезаписать (в байтах), в качестве аргументов.

Сначала вычисляется смещение между местом перехвата и трамплином, и результат сохраняется в newOffset ❶. Затем к изменяемой памяти применяются права PAGE\_EXECUTE\_READWRITE.

Безусловный переход (`0xE9`) ❷ и адрес функции обратного вызова ❸ затем записываются в память, а цикл `for` записывает NOP-инструкции (`0x90`) в любые оставшиеся байты. После повторного применения старых прав функция `hookWithJump()` возвращается к исходному адресу.

Обратите внимание, что функция `hookWithJump()` гарантирует, что `size` не превышает 12 перед установкой перехода. Эта проверка важна, потому что `jmp` занимает 5 байтов, а это значит, что он может заменить до пяти команд, если первые четыре команды состоят из одного байта. Если первые четыре команды — однобайтовые, то пятая команда должна занимать более 8 байтов, чтобы сработало условие `if (size > 12)`.

Так как 9-байтовые операции встречаются очень редко, 12 — это безопасный, но гибкий предел. Этот лимит предотвращает множество багов, особенно если ваш бот динамически определяет размер параметра. Если бот ошибется и передаст `size = 500,000,000`, например, проверка предотвратит затирание NOP-ами всей вселенной.

## Написание функции трамплина

Используя функцию из Листинга 8-11, вы можете воспроизвести перехват, показанный на Рисунке 8-1, но сначала вам нужно создать функцию трамплина следующим образом:

---

```
DWORD restoreJumpHook = 0;
void __declspec(naked) myTrampoline()
{
    __asm {
        ❶ PUSHFD
        ❷ PUSHAD
        ❸ CALL jumpHookCallback
        ❹ POPAD
        ❺ POPFD
        ❻ POP EAX
        MOV AL, 1
        POP EDI
        ❼ POP ESI
        ❽ JMP [restoreJumpHook]
    }
}
```

---

Точно так же, как трамплин, описанный вместе с Рисунком 8-1, этот трамплин сохраняет все текущие флаги ❶ и значения регистров ❷, вызывает функцию обратного вызова ❸, восстанавливает регистры ❹, восстанавливает флаги ❺, выполняет код, который был заменён перехватом в ❻ и ❼, а затем, наконец, прыгает обратно в оригинальный код сразу после прыжка и NOP-ов ❽.

**НОТЕ**

Чтобы убедиться, что компилятор не сгенерирует автоматически дополнительный код внутри трамплина, всегда объявляйте трамплин, используя соглашение `_declspec(naked)`.

## Завершение перехвата прыжка

Как только вы создадите трамплин, определите функцию обратного вызова и установите перехват следующим образом:

```
void jumpHookCallback() {
    // Выполняем действия
}

restoreJumpHook = hookWithJump(0xDEADBEEF, &myTrampoline, 5);
```

Наконец, внутри функции `jumpHookCallback()` выполняется код, который зависит от перехвата. Если вашему коду нужно читать или записывать значения регистров в том виде, в котором они были при срабатывании перехвата, вам повезло. Команда `PUSHAD` помещает их в стек в порядке **EAX, ECX, EDX, EBX, исходные ESP, EBP, ESI и EDI**. Трамплин вызывает `PUSHAD` непосредственно перед вызовом `jumpHookCallback()`, поэтому можно передавать значения регистров в качестве аргументов, например так:

```
void jumpHookCallback(DWORD EDI, DWORD ESI, DWORD EBP, DWORD ESP,
                      DWORD EBX, DWORD EDX, DWORD ECX, DWORD EAX) {

    // Выполняем действия
}

restoreJumpHook = hookWithJump(0xDEADBEEF, &myTrampoline, 5);
```

Поскольку трамплин использует `POPAD` для непосредственного восстановления регистров из значений в стеке, любые изменения, которые вы вносите в параметры, будут применены к реальным регистрам при их восстановлении из стека.

Как и перехваты таблицы виртуальных функций (VF table hooks), перехваты с помощью `jmp hooks` редко необходимы, и их может быть сложно смоделировать на простом примере. Чтобы помочь вам разобраться с ними, я рассмотрю реальный практический случай использования в разделе "**Applying Jump Hooks and VF Hooks to Direct3D**" на странице 175.

## ПРОФЕССИОНАЛЬНЫЕ БИБЛИОТЕКИ ДЛЯ ПЕРЕХВАТА API

Существуют готовые библиотеки для перехвата API, такие как Microsoft's Detours и MadCHook, которые используют только jump hooks. Эти библиотеки могут автоматически обнаруживать и отслеживать другие перехваты, знают, сколько инструкций нужно заменить, и создают для вас функции трамплина.

Они могут делать это потому, что умеют разбирать (disassemble) код и проходить через инструкции ассемблера, чтобы определить длину инструкций, места назначения прыжков и так далее. Если вам нужно использовать перехваты с такой мощностью, скорее всего, лучше воспользоваться одной из этих библиотек, чем создавать свою собственную.

## Применение перехватов вызовов (Call Hooks) к Adobe AIR

Adobe AIR — это среда разработки, позволяющая создавать кроссплатформенные игры в окружении, похожем на Adobe Flash. AIR является распространённым фреймворком для онлайн-игр, так как позволяет разработчикам писать кроссплатформенный код на удобном высокуюровневом языке ActionScript.

ActionScript — это интерпретируемый язык, и AIR выполняет код внутри виртуальной машины, что делает невозможным перехват (hook) кода, специфичного для конкретной игры, работающей в AIR. Вместо этого проще перехватывать сам AIR.

Пример кода для этого раздела можно найти в GameHackingExamples/Chapter8\_AdobeAirHook в исходниках к этой книге. Код основан на моём старом проекте и работает на любой игре, использующей Adobe AIR.dll версии 3.7.0.1530. Я добился его работоспособности и на других версиях, но не могу гарантировать, что он будет работать на гораздо более новых или более старых версиях, так что воспринимайте это как исследовательский кейс (case study).

## Доступ к золотой жиле RTMP (Accessing the RTMP Goldmine)

Протокол Real-Time Messaging Protocol (RTMP) — это текстовый сетевой протокол, который ActionScript использует для сериализации и отправки объектов по сети. RTMP работает поверх HyperText Transfer Protocol (HTTP), а его защищённая версия, RTMPS, использует HTTP Secure (HTTPS).

RTMPS позволяет разработчикам удобно отправлять и получать целые объекты по безопасному соединению с минимальными сложностями. Это делает RTMP предпочтительным сетевым протоколом для игр, работающих на AIR.

**НОТЕ**

Данные, отправленные через RTMP/RTMPS, сериализуются с помощью Action Message Format (AMF), а разбор пакетов AMF выходит за рамки данной книги. Найдите в интернете "AMF3 Parser", и вы найдёте много кода, который это делает.

Данные, отправленные через **RTMP** и **RTMPS**, очень насыщенные. Пакеты содержат информацию о типах объектов, именах и значениях. Это настоящая золотая жила. Если вы сможете перехватывать эти данные в реальном времени, вы сможете мгновенно реагировать на изменения в состоянии игры, видеть огромное количество критически важной информации, даже не считывая её из памяти, и находить полезные фрагменты данных, о существовании которых вы даже не подозревали.

Некоторое время назад я работал над инструментом, который требовал большого количества информации о состоянии игры. Получение столь большого объёма данных напрямую из памяти было бы крайне сложным, если не невозможным. После некоторых исследований я понял, что игра использует **RTMPS** для связи с сервером, и это подтолкнуло меня к изучению этой "золотой жилы".

Так как **RTMPS** зашифрован, я знал, что мне придётся каким-то образом перехватывать криптографические функции, используемые **AIR**, прежде чем я смогу получить какие-либо полезные данные. После поиска в интернете я нашёл исходный код небольшого инструмента под названием **airlog**, созданного другим игровым хакером, который, как и я, пытался логировать пакеты, передаваемые через **RTMPS**. Хотя этот инструмент перехватывал именно те функции, которые мне были нужны, код был устаревшим, неаккуратным и, что хуже всего, не работал с версией **AIR**, которую я пытался взломать.

Но это не означало, что он был бесполезен. **Airlog** не только перехватывал две нужные мне функции, но и находил их, сканируя характерные байтовые паттерны в библиотеке **Adobe AIR**. Однако эти байтовые паттерны были трёхлетней давности, и они больше не работали. В новых версиях **Adobe AIR** код изменился настолько, что байты в ассемблере больше не совпадали. Эта разница в байтах была проблемой для **airlog**, но не для меня.

Внутри блока **inline-ассемблера** вы можете задать сырье байты с помощью следующего вызова функции:

---

`_emit BYTE`

Если заменить **BYTE** на, скажем, **0x03**, код скомпилируется так, что **0x03** будет трактоваться как байт в ассемблерном коде, независимо от того, имеет ли это смысл. Используя этот трюк, я скомпилировал массивы байтов обратно в ассемблерный код. Этот код ничего не делал, и он не должен был ничего делать; этот приём просто позволил мне подключить моё тестовое приложение к **OllyDBG** и инспектировать байты, которые удобно представлялись как чистый дизассемблированный код.

Так как эти байты представляли код, окружающий нужные мне функции, мне также понадобился их дизассемблированный вариант. Код был довольно стандартным и, похоже, не собирался меняться, поэтому я переключил внимание на **константы**. В коде было несколько непосредственных значений, передаваемых как смещения в командах. Зная, насколько часто такие значения могут изменяться, я переработал алгоритм **поиска паттернов airlog**, чтобы он поддерживал **wildcard'ы** (символы-заполнители), обновил паттерны так, чтобы любые константы рассматривались как **wildcard'ы**, и затем запустил поиск.

После нескольких корректировок паттернов и небольшого "копания" в повторяющихся результатах поиска я **вычислил нужные мне функции**. Я дал им осмысленные названия — **encode()** и **decode()** — и начал работать над инструментом, похожим на **airlog**, но **лучше**.

## Хук на функцию encode() в RTMPS

Я обнаружил, что функция **encode()**, которая используется для шифрования данных в исходящих пакетах, является невиртуальной и использует **\_thiscall**, то есть вызывается через **near call**. Более того, этот вызов происходит внутри цикла. Весь цикл выглядит как Листинг 8-12, взятый напрямую из окна дизассемблирования OllyDBG.

---

```
loop:
    MOV EAX, [ESI+3C58]
    SUB EAX, EDI
    PUSH EAX
①   LEA EAX, [ESI+EDI+1C58]
    PUSH EAX
    MOV ECX, ESI
②   CALL encode
    CMP EAX, -1
③   JE SHORT endLoop
    ADD EDI, EAX
④   CMP EDI, [ESI+3C58]
    JL loop
endLoop:
```

---

Листинг 8-12: The encode() loop

С помощью небольшого анализа и некоторых рекомендаций от **airlog** я определил, что функция **encode()**, вызываемая в ①, принимает массив байтов и длину буфера (назовем их **buffer** и **size**, соответственно) в качестве параметров.

Функция возвращает **-1**, если произошел сбой, и **size** в противном случае.

Функция обрабатывает данные порциями по 4 096 байт, поэтому работает внутри цикла.

Преведенный в более читаемый псевдокод, цикл, вызывающий encode(), выглядит так (числа соответствуют соответствующим инструкциям ассемблера в Листинге 8-12):

---

```
for (EDI = 0; EDI < ❶[ESI+3C58]; ) {
    EAX = ❷encode(❸&[ESI+EDI+1C58], [ESI+3C58] - EDI);
    if (EAX == -1) ❹break;
    EDI += EAX;
}
```

---

Меня не интересовало, что делает encode(), но мне нужен был весь буфер, по которому он выполнял цикл, и перехват encode() был моим способом получения этого буфера. Рассматривая реальный цикл в Листинге 8-12, я понял, что:

Полный буфер вызывающего объекта хранится по адресу ESI+0x1C58, Полный размер хранится по адресу ESI+0x3C58, EDI содержит счетчик цикла.

Я создал перехват с учетом этих особенностей, в итоге создав перехват из двух частей. Первая часть моего перехвата — это функция reportEncode(), которая логирует весь буфер на первой итерации цикла. Вот полная реализация функции reportEncode():

---

```
DWORD __stdcall reportEncode(
    const unsigned char* buffer,
    unsigned int size,

    unsigned int loopCounter)
{
    if (loopCounter == 0)
        printBuffer(buffer, size);
    return origEncodeFunc;
}
```

---

Эта функция принимает **buffer**, **size** и **loopCounter** в качестве параметров и возвращает адрес функции, которую я назвал **encode()**.

Прежде чем получить этот адрес, вторая часть моего перехвата, функция **myEncode()**, выполняет всю грязную работу по извлечению **buffer**, **size** и **loopCounter**, как показано ниже:

```

void __declspec(naked) myEncode()
{
    __asm {
        MOV EAX, DWORD PTR SS:[ESP + 0x4] // получить buffer
        MOV EDX, DWORD PTR DS:[ESI + 0x3C58] // получить полный размер
        PUSH ECX // сохранить ECX
        PUSH EDI // сохранить текущую позицию (счетчик цикла)
        PUSH EDX // сохранить size
        PUSH EAX // сохранить buffer
        CALL reportEncode // зарегистрировать вызов encode
        POP ECX // восстановить ECX
        JMP EAX // перейти к encode
    }
}

```

Функция **myEncode()** — это чистая ассемблерная функция, которая заменяет оригинальный вызов функции **encode()** с помощью перехвата **near call**. Сначала она сохраняет **ECX** в стеке, затем **myEncode()** получает **buffer**, **size** и **loopCounter** и передает их в функцию **reportEncode()**. После вызова **reportEncode()** функция **myEncode()** восстанавливает **ECX** и сразу переходит в **encode()**, заставляя оригинальную функцию выполняться и корректно возвращать управление в цикл.

Поскольку **myEncode()** очищает все, что использует из стека, стек все еще содержит исходные параметры и адрес возврата в правильном месте после выполнения **myEncode()**. Именно поэтому **myEncode()** переходит непосредственно в **encode()**, а не вызывает ее как функцию: стек уже настроен с правильными параметрами и адресом возврата, так что **encode()** будет думать, что все произошло как обычно.

## Перехват функции RTMPS decode()

Функция, которую я назвал **decode()**, предназначена для расшифровки входящих данных и также была **\_thiscall**, вызываемой в цикле. Она работала с блоками по 4 096 байт и принимала **buffer** и **size** в качестве параметров. Цикл был немного сложнее, содержал несколько вызовов функций, вложенные циклы и выходы из них, но сам процесс перехвата работал примерно так же, как для так называемой **encode()**.

Причина дополнительной сложности не имеет отношения к перехвату функции, но делает код сложным для краткого изложения, поэтому я не показываю оригинальную функцию здесь. Итог таков: после устранения всей сложности **decode()** выполнял тот же цикл, что и **encode()**, но в обратном порядке.

Однажды я снова разработал двухкомпонентный перехват **near call**. Первая часть, **reportDecode()**, представлена ниже:

```
void __stdcall reportDecode(const unsigned char* buffer, unsigned int size)
{
    printBuffer(buffer, size);
}
```

Функция регистрирует каждый проходящий пакет. У меня не было индекса цикла в тот момент, поэтому я решил, что можно логировать каждый отдельный частичный пакет.

Вторая часть перехвата, функция **myDecode()**, действует как новый вызываемый код и выполняет всю грязную работу, как показано ниже:

```
void __declspec(naked) myDecode()
{
    __asm {
        MOV EAX, DWORD PTR SS:[ESP + 0x4] // получить буфер
        MOV EDX, DWORD PTR SS:[ESP + 0x8] // получить размер
        PUSH EDX // сохранить размер
        PUSH EAX // сохранить буфер
        CALL [origDecodeFunc] // ❶ вызов оригинальной функции декодирования

        MOV EDX, DWORD PTR SS:[ESP + 0x4] // получить буфер

        PUSH EAX // ❷ сохранить eax (возвращаемое значение)
        PUSH ECX // сохранить ecx
        PUSH EAX // сохранить размер
        PUSH EDX // сохранить буфер
        CALL reportDecode // передать результаты в reportDecode
        POP ECX // восстановить ecx
        POP EAX // ❸ восстановить eax (возвращаемое значение)
        RETN 8 // возврат и очистка стека
    }
}
```



Я знал, что буфер расшифровывался на месте, что означало, что зашифрованный фрагмент будет перезаписан расшифрованным сразу после завершения вызова decode(). Это означало, что myDecode() должен был вызывать оригинальную функцию decode() ❶ перед вызовом функции reportDecode(), которая предоставляла результаты декодирования. В конечном итоге, myDecode() также должен был вернуть то же значение, которое вернула бы оригинальная функция decode(), и очистить стек, а инструкции POP ❷ и RETN ❸ позаботились об этом.

## Размещение хуков (Placing the Hooks)

Следующая проблема, с которой я столкнулся, заключалась в том, что хуки предназначались для кода внутри модуля *Adobe AIR.dll*, который не являлся основным модулем игры. Из-за расположения кода мне нужно было найти базовые адреса для хуков немного иначе. Кроме того, поскольку мне было необходимо, чтобы эти хуки работали в нескольких различных версиях *Adobe*

*AIR*, мне также нужно было найти правильные адреса для каждой версии.

Вместо того чтобы пытаться получить все разные версии *Adobe AIR*, я воспользовался ещё одной тактикой из арсенала *airlog* и решил программно определить адреса, написав небольшой сканер памяти. Прежде чем я смог написать сканер памяти, мне нужно было получить как базовый адрес, так и размер *Adobe AIR.dll*, чтобы ограничить поиск в памяти только этой областью.

Я нашёл эти значения, используя *Module32First()* и *Module32Next()*, следующим образом:

---

```
MODULEENTRY32 entry;
entry.dwSize = sizeof(MODULEENTRY32);
HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, NULL);

DWORD base, size;
if (Module32First(snapshot, &entry) == TRUE) {
    ❶    while (Module32Next(snapshot, &entry) == TRUE) {
        std::wstring binaryPath = entry.szModule;
    ❷        if (binaryPath.find("Adobe AIR.dll") != std::wstring::npos) {
            size = (DWORD)entry.modBaseSize;
            base = (DWORD)entry.modBaseAddr;
            break;
        }
    }
}

CloseHandle(snapshot);
```

---

Этот код проходит через все модули в процессе, пока не найдёт *Adobe AIR.dll* ❶. Когда он находит правильную запись модуля ❷, он извлекает свойства *modBaseSize* и *modBaseAddr* из неё и немедленно выходит.

Следующим шагом было нахождение последовательности байтов, которую я мог бы использовать для идентификации функций. Я решил использовать байтовый код, окружающий каждый вызов. Мне также нужно было убедиться, что каждая последовательность была уникальной, избегая использования любых констант в шаблонах, чтобы обеспечить переносимость кода.

В списке 8-13 показаны байтовые последовательности, которые я получил.

---

```
const char encodeSeq[16] = {
    0x8B, 0xCE,           // MOV ECX, ESI
    0xE8, 0xA6, 0xFF, 0xFF, 0xFF, // CALL encode
    0x83, 0xF8, 0xFF,           // CMP EAX, -1
    0x74, 0x16,           // JE SHORT endLoop
    0x03, 0xF8,           // ADD EDI, EAX
    0x3B, 0xBE};           // part of CMP EDI, [ESI+0x3C58]
const char decodeSeq[12] = {
    0x8B, 0xCE,           // MOV ECX, ESI
    0xE8, 0x7F, 0xF7, 0xFF, 0xFF, // CALL decode
    0x83, 0xF8, 0xFF,           // CMP EAX, -1
    0x89, 0x86};           // part of MOV [ESI+0x1C54], EAX
```

---

Листинг 8-13: The *encode()* and *decode()* byte sequences

Обратите внимание на инструкцию *CALL* в каждом шаблоне; это вызовы функций *Adobe AIR*, которые я назвал *encode()* и *decode()*. Я сканировал эти последовательности с помощью следующей функции:

---

```
DWORD findSequence(
    DWORD base, DWORD size,
    const char* sequence,
    unsigned int seqLen){
    for (DWORD adr = base; adr <= base + size - seqLen; adr++) {
        if (memcmp((LPVOID)sequence, (LPVOID)adr, seqLen) == 0)
            return adr;
    }
    return 0;
}
```

---

Обрабатывая память *Adobe AIR.dll* как массив байтов, функция *findSequence()* ищет последовательность байтов как подмножество этого массива байтов и возвращает адрес первого найденного совпадения. С написанной функцией *findSequence()* поиск адресов, необходимых для перехвата *encode()* и *decode()*, стал простым.

Бот как выглядели эти вызовы:

---

```
DWORD encodeHookAt =
    findSequence(base, size, encodeSeq, 16) + 2;
DWORD decodeHookAt =
    findSequence(base, size, decodeSeq, 12) + 2;
```

---

## Применение Jump-хуков и VF-хуков к Direct3D (Applying Jump Hooks and VF Hooks to Direct3D)

В отличие от хука *Adobe AIR*, который я только что описал, хуки для *Direct3D* (3D-графического компонента API *Microsoft DirectX*) являются очень распространёнными и хорошо

задокументированными. *Direct3D* повсеместно используется в мире игр: большинство компьютерных игр используют эту библиотеку, а это значит, что её перехват даёт мощный способ для перехвата данных и манипулирования графическими слоями множества различных игр.

Вы можете использовать хук *Direct3D* для множества задач, таких как обнаружение местоположения скрытых противников, увеличение освещения в тёмных игровых сценах и плавное отображение дополнительной графической информации. Эффективное использование хука *Direct3D* требует знания API, но в этой книге достаточно информации, чтобы помочь вам начать.

В этом разделе я дам вам общий обзор игрового цикла, который использует *Direct3D*, прежде чем сразу перейти к реализации хука *Direct3D*. Вместо того чтобы подробно разбирать внутреннее устройство и приводить аналитический контекст, как я сделал с хуком *Adobe AIR*, я рассмотрю самый популярный метод хука *Direct3D*, так как он хорошо задокументирован и используется большинством игровых хакеров.

Онлайн-ресурсы для этой книги включают два примера кода для этого раздела; найдите эти файлы сейчас, если хотите следовать за материалом. Первая часть — это пример приложения *Direct3D 9* для тестирования хука, который можно найти в *GameHackingExamples/Chapter8\_Direct3DApplication*. Вторая часть, содержащая сам хук, находится в *Chapter8\_Direct3DHook*.

Существует несколько версий *Direct3D*, используемых в разное время, и существуют способы перехвата каждой из них. В этой книге я сосредоточусь на перехвате *Direct3D 9*, так как это единственная широко используемая версия, поддерживаемая *Windows XP*.

**NOTE**

Хотя срок службы *Windows XP* завершён, многие люди в менее развитых странах всё ещё используют её в качестве основной игровой системы.

*Direct3D 9* работает на всех версиях *Windows* и почти так же мощен, как его преемники, поэтому многие игровые компании по-прежнему предпочитают использовать его вместо новых версий, которые не обладают такой же обратной совместимостью.

## Цикл отрисовки (The Drawing Loop)

Давайте сразу перейдём к краткому курсу о том, как работает *Direct3D*. В исходном коде игры на *Direct3D* вы найдёте бесконечный цикл, который обрабатывает ввод и рендерит графику. Каждая итерация этого цикла отрисовки называется *кадром (frame)*. Если убрать весь лишний код и сосредоточиться только на базовом скелете, то можно представить основной цикл игры следующим кодом:

```

int WINAPI WinMain(args)
{
    /* здесь должен быть код для
       настройки Direct3D и инициализации
       игры. Опущен для краткости. */

    MSG msg;
    while(TRUE) {
        /* здесь должен быть код для обработки
           входящих сообщений от мыши и клавиатуры. */
        drawFrame(); // это функция, которая нас интересует
    }
    /* здесь должен быть код для очистки
       перед выходом. */
}

```

Эта функция является точкой входа в игру. Проще говоря, она инициализирует игру, а затем переходит в основной игровой цикл. Внутри основного цикла выполняется код, отвечающий за обработку ввода пользователя, прежде чем вызвать *drawFrame()* для перерисовки экрана с использованием *Direct3D*. (Ознакомьтесь с кодом в *GameHackingExamples/Chapter8\_Direct3DApplication*, чтобы увидеть полностью функциональный игровой цикл.)

Каждый раз, когда вызывается функция *drawFrame()*, она перерисовывает весь экран. Код выглядит примерно так:

---

```

void drawFrame()
{
    device->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0, 0, 0), 1.0f, 0);
    device->BeginScene();
    // drawing will happen here
    device->EndScene();
    device->Present(NULL, NULL, NULL, NULL);
}

```

---

После очистки экрана с помощью *device->Clear* ❶, функция *drawFrame()* вызывает *device->BeginScene()*, чтобы разблокировать сцену для рисования. Затем выполняется некоторый код отрисовки (что именно делает этот код, сейчас не важно) и блокирует сцену вызовом *device->EndScene()*. В завершение сценарендерится на экран с помощью вызова функции *device->Present()*.

Обратите внимание, что все эти функции вызываются как члены некоторого экземпляра под названием *device*. Это просто объект, представляющий устройство *Direct3D*, который используется для вызова различных функций отрисовки. Также обратите внимание, что эта функция не содержит никакого фактического кода рисования, но это не проблема. На самом деле, важно лишь понимать общие концепции циклов отрисовки, кадров и устройства *Direct3D*.

Подытожим, у игр есть основной цикл с двумя обязанностями: Обработка входящих сообщений

## Отрисовка игры на экране

Каждая итерация в этом цикле называется *кадром (frame)*, и каждый кадр рисуется с помощью устройства. Контроль над устройством даёт вам доступ к наиболее чувствительным и детальным аспектам состояния игры; то есть, вы сможете заглянуть в состояние игры после того, как данные были разобраны, обработаны и выведены на экран. Более того, вы сможете модифицировать выходные данные этого состояния.

Эти две суперспособности позволяют вам совершать самые разные крутые хаки.

## Нахождение устройства Direct3D (Finding the Direct3D Device)

Чтобы получить контроль над устройством *Direct3D*, вы перехватываете функции-члены в таблице *VF* устройства. Однако, к сожалению, использование *Direct3D API* для создания собственного экземпляра того же класса *device* из внедрённого кода не означает, что вы будете разделять таблицу *VF* с экземпляром игры.

Устройства *Direct3D* используют индивидуальную реализацию *VF*-таблиц во время выполнения, и каждое устройство получает свою уникальную *VF*-таблицу. Более того, устройства иногда переписывают свои *VF*-таблицы, удаляя любые хуки и восстанавливая оригинальные адреса функций.

Обе эти особенности *Direct3D* оставляют вам только один неизбежный вариант: вам необходимо найти адрес устройства игры и модифицировать его *VF*-таблицу напрямую.

Вот как это сделать:

1. Создайте устройство *Direct3D* и просканируйте его *VF*-таблицу, чтобы найти настоящий адрес *EndScene()*.
2. Установите временный *jmp hook* на *EndScene()*.
3. Когда вызывается обратный вызов *jmp hook*, сохраните адрес устройства, которое использовалось для вызова функции, удалите хук и восстановите нормальное выполнение.
4. После этого используйте *VF*-хуки для перехвата любой функции-члена устройства *Direct3D*.

## Перехват *EndScene()* с помощью *Jump Hook* (Jump Hooking *EndScene()*)

Поскольку каждое устройство будет вызывать *EndScene()* в конце каждого кадра, вы можете перехватить *EndScene()* с помощью *jmp hook* и перехватить устройство игры через ваш обратный вызов. Уникальные устройства могут иметь свои собственные уникальные *VF* таблицы, но разные таблицы всё равно указывают на одни и те же функции, так что вы можете найти адрес *EndScene()* в *VF* таблице любого произвольного устройства. Используя стандартные вызовы *Direct3D API*, вы можете создать своё собственное устройство следующим образом:

---

```
LPDIRECT3D9 pD3D = Direct3DCreate9(D3D_SDK_VERSION);
if (!pD3D) return 0;

D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.hDeviceWindow = hWnd;

LPDIRECT3DDEVICE9 device;
HRESULT res = pD3D->CreateDevice(
    D3DADAPTER_DEFAULT,
    D3DDEVTYPE_HAL,
    hWnd,
    D3DCREATE_SOFTWARE_VERTEXPROCESSING,
    &d3dpp, &device);
if (FAILED(res)) return 0;
```

---

Объяснение того, как работает всё в *Direct3D*, выходит за пределы этой книги, так что просто знайте, что вы можете скопировать этот код для создания устройства *Direct3D*, которое содержит функцию *EndScene()* как член. Адрес *EndScene()* находится на индексе 42 в VF таблице устройства (см. раздел «Значение устройства, Direct3D и VF хуки» для того, чтобы узнать, как найти этот индекс), и вы можете прочитать его, используя подмножество кода для перехвата VF таблицы из раздела «Использование VF Table Hook» на странице 159, следующим образом:

---

```
DWORD getVF(DWORD classInst, DWORD funcIndex)
{
    DWORD VFTable = readMemory<DWORD>(classInst);
    DWORD hookAddress = VFTable + funcIndex * sizeof(DWORD);
    return readMemory<DWORD>(hookAddress);
}
DWORD EndSceneAddress = getVF((DWORD)device, 42);
```

---

Как только вы получили адрес, ваше устройство для поиска выполнило свою задачу, и его можно уничтожить с помощью вызова функции *Release()*:

---

```
pD3D->Release();
device->Release();
```

---

## **Значение устройства, Direct3D и VF хуков (THE MEANING OF DEVICE, DIRECT3D, AND VF HOOKS)**

**Если вы задаётесь вопросом, как я узнал, что индекс функции `EndScene()` равен 42, то вы пришли в правильное место. Поскольку Direct3D 9 — это свободно доступная библиотека, вы можете на самом деле увидеть многое из того, что происходит «под капотом». Основной заголовочный файл для этой библиотеки — `d3d9.h`. Если вы откроете этот файл в вашем редакторе и выполните поиск по слову `EndScene`, то окажетесь внутри большого определения класса, который специфицирует несколько функций с использованием макросов C. Это базовый класс для всех реализаций устройства Direct3D 9 и он определяет виртуальные функции, используемые этим классом.**

**VF таблица строится в том же порядке, что и функции, определённые в коде, так что вы можете определить индекс любой функции-члена, просто посчитав строки. Вы можете прокрутить вверх до начала определения класса (на строке 426 в моей версии библиотеки и, вероятно, в вашей тоже), затем отметить строку, где объявлена первая функция (строка 429), и затем прокрутить до определения `EndScene()` и отметить эту строку (строка 473). Наконец, посчитайте количество пустых или закомментированных строк (две у меня) и выполните некоторые математические действия:  $473 - 429 - 2 = 42$ .**

**Вот и всё! Функция `EndScene()` — это 43-я объявленная функция, так что она находится на 42-й позиции в VF таблице. Ещё одно преимущество наличия этого заголовочного файла — это то, что вы можете увидеть имя, типы аргументов, имена аргументов и тип возвращаемого значения каждой функции в классе устройства. Так что, когда вы будете писать свои собственные хуки в будущем, вы точно будете знать, куда смотреть. Каждой функции в классе устройства. Так что в будущем, когда вы будете писать свои собственные крючки, вы будете точно знать, где их искать.**

## **Размещение и удаление Jump хука (Placing and Removing the Jump Hook)**

Поскольку вы используете хук только для нахождения устройства, вам нужно вызывать его всего один раз. После получения устройства вы удалите *jump hook* и восстановите выполнение с начала `EndScene()`, чтобы цикл отрисовки мог продолжить свою работу. Поверьте или нет, это значительно облегчит вашу задачу. Поскольку код будет восстановлен немедленно, вам не нужно, чтобы ваш «трамплин» выполнял команды, заменённые хаком, и нет необходимости дополнительного заполнять хак с помощью *NOP*. Всё, что вам нужно сделать, — это сохранить оригинальные байты и разместить хук. Для этого вы используете немного изменённую версию кода перехвата хука из списка 8-11:

---

```
unsigned char* hookWithJump(DWORD hookAt, DWORD newFunc)
{
    DWORD newOffset = newFunc - hookAt - 5;
    ❶ auto oldProtection = protectMemory<BYTE[5]>(hookAt, PAGE_EXECUTE_READWRITE);
    unsigned char* originals = new unsigned char[5];
    for (int i = 0; i < 5; i++)
        ❷     originals[i] = readMemory<unsigned char>(hookAt + i);
    ❸     writeMemory<BYTE>(hookAt, 0xE9);
    writeMemory<DWORD>(hookAt + 1, newOffset);
    protectMemory<BYTE[5]>(hookAt, oldProtection);
    return originals;
}
```

---

Как и функция из списка 8-11, эта функция делает память доступной для записи ❶, размещает хук ❷ и восстанавливает защиту памяти. Перед размещением хука она выделяет 5-байтный буфер, называемый *originals* ❸, и заполняет его оригинальными байтами. После того как хук размещён, она возвращает *originals* вызывающей функции.

Когда приходит время удалить хук, передайте *originals* в следующую функцию:

```
void unhookWithJump(DWORD hookAt, unsigned char* originals)
{
    auto oldProtection = protectMemory<BYTE[5]>(hookAt, PAGE_EXECUTE_READWRITE);
    for (int i = 0; i < 5; i++)
        writeMemory<BYTE>(hookAt + i, originals[i]);
    protectMemory<BYTE[5]>(hookAt, oldProtection);
    delete [] originals;
}
```

---

Этот код просто поочередно проходит по *originals* и тихо размещает эти 5 байтов обратно туда, где они были найдены, чтобы всё работало, как ожидалось, когда выполнение вернётся в функцию *EndScene()*. Когда придёт время, вы сможете разместить и удалить ваш настоящий хук с помощью двух строк кода, как показано ниже:

```
auto originals = hookWithJump(EndSceneAddress, (DWORD)&endSceneTrampoline);
unhookWithJump(EndSceneAddress, originals);
```

---

Как только у вас есть функции *hookWithJump()* и *unhookWithJump()*, пришло время подготовить обратный вызов и найти устройство.

## Написание обратного вызова и "трамплина" (Writing the Callback and Trampoline)

Несмотря на то, что вы можете получить адрес *EndScene()* из VF таблицы, функция *EndScene()* на самом деле не следует соглашению *\_thiscall*. Классы *Direct3D* являются простыми

оболочками вокруг C API, и все вызовы функций-членов перенаправляются на функции `_stdcall`, которые принимают экземпляр класса в качестве первого параметра. Это означает, что вашему "трамплину" нужно только извлечь устройство из стека, передать его в обратный вызов и затем вернуться к `EndScene()`. Обратный вызов должен только удалить *jmp hook*, прежде чем вернуться к "трамплину".

Конечный код для обратного вызова и "трамплина" для этого *jmp hook* выглядит следующим образом:

```
LPDIRECT3DDEVICE9 discoveredDevice;
DWORD __stdcall reportInitEndScene(LPDIRECT3DDEVICE9 device)
{
    discoveredDevice = device;
    unhookWithJump(EndSceneAddress, originals);
    return EndSceneAddress;
}
__declspec(naked) void endSceneTrampoline()
{
    __asm {
        MOV EAX, DWORD PTR SS:[ESP + 0x4]
        PUSH EAX // give the device to the callback
①     CALL reportInitEndScene
        JMP EAX // jump to the start of EndScene
    }
}
```

Используя функцию `hookWithJump()`, вы можете разместить *jmp hook* на `EndScene()`, который вызывает функцию `endSceneTrampoline()`. Когда устройство игры вызывает функцию `EndScene()`, функция `trampoline` вызывает функцию `reportInitEndScene()` ①. Функция `reportInitEndScene()` сохраняет захваченный указатель устройства в глобальную переменную, называемую `discoveredDevice`, удаляет хук, вызывая `unhookWithJump()`, и возвращает адрес `EndScene()` в `trampoline`. В завершение `trampoline` переходит непосредственно к `EAX`, который будет содержать адрес, возвращённый из отчётной функции.

**NOTE** Вы можете использовать *jmp hooks*, чтобы полностью избежать перехвата через VF таблицу, которую я вам покажу, но использование «тупых» *jmp hooks* на часто перехватываемых API-функциях очень ненадёжно. Постоянное получение хороших результатов только с использованием *jmp hooks* требует профессиональных библиотек для перехвата, и я бы предпочёл научить вас делать это полностью самостоятельно.

На этом этапе всё, что осталось сделать, — это перехватить VF таблицу `discoveredDevice`, чтобы взломать игру. Следующие два раздела проведут вас через перехваты функций `EndScene()` и `Reset()`, которые необходимы, если вы хотите стабильный хук.

## Написание хука для *EndScene()* (Writing a Hook for *EndScene()*)

Хук на *EndScene()* полезен, потому что он позволяет перехватывать завершённый кадр сразу перед его рендерингом; вы можете эффективно выполнить свой собственный код рендеринга внутри игрового цикла. Как вы видели при нахождении адреса этой функции в разделе «*Jump Hooking EndScene()*» на странице 178, эта функция находится на индексе 42 в VF таблице. Вы можете перехватить *EndScene()* с использованием VF hook следующим образом:

```
typedef HRESULT (WINAPI* _endScene) (LPDIRECT3DDEVICE9 pDevice);
_endScene origEndScene =
(_endScene)hookVF ((DWORD)discoveredDevice, 42, (DWORD)&myEndScene);
HRESULT WINAPI myEndScene(LPDIRECT3DDEVICE9 pDevice)
{
    // рисуйте свой собственный код здесь
    return origEndScene(pDevice);
}
```

Этот код использует функцию *hookVF()* из раздела «*Using a VF Table Hook*» на странице 159 для перехвата *EndScene()* на индексе 42 устройства *discoveredDevice*, используя *myEndScene()* как функцию обратного вызова. Прямое устройство *Direct3D* будет время от времени переподключать свою собственную VF таблицу и восстанавливать оригинальные адреса функций. Это обычно происходит внутри функции *EndScene()*, что означает, что вам также нужно переподключить VF таблицу после вызова оригинальной функции *EndScene()*. Существуют несколько изменений, которые вы можете внести в этот хук, чтобы справиться с этим, как показано в списке 8-14.

```
_endScene origEndScene = NULL;
void placeHooks()
{
    auto ret = hookVF((DWORD)discoveredDevice, 42, (DWORD)&myEndScene);
    if (ret != (DWORD)&myEndScene) // не указывает на ваш хук
        origEndScene = (_endScene)ret;
}
placeHooks();

HRESULT WINAPI myEndScene(LPDIRECT3DDEVICE9 pDevice)
{
    // рисуйте свой собственный код здесь
    auto ret = origEndScene(pDevice);
    placeHooks(); // обновить хуки
    return ret;
}
```

*Листинг 8-14: Финальный код для подключения `EndScene()`*

Код для размещения хука был перемещён в функцию под названием `placeHooks()`, чтобы его можно было вызывать несколько раз с лёгкостью. Функция обратного вызова всё равно перенаправляет вызов к оригинальной функции, но перед возвратом она обязательно вызывает `placeHooks()`. Это гарантирует, что хук всегда активен, даже если оригинальная функция `EndScene()` удалит его.

Ещё один момент, который стоит отметить, это то, что `placeHooks()` обновляет адрес `origEndScene()` каждый раз, когда хук заменяется, при условии, что адрес, возвращённый из `hookVF()`, не является адресом функции `myEndScene()`. Это выполняет две различные задачи. Во-первых, это позволяет другим приложениям перехватывать `EndScene()`, не вмешиваясь в их работу, так как оно обновит `origEndScene()` на тот адрес, который указан в VF таблице. Во-вторых, это гарантирует, что значение `origEndScene()` никогда не будет адресом нашего обратного вызова, предотвращая возможный бесконечный цикл. Бесконечный цикл возможен в другом случае, потому что `origEndScene()` не всегда фиксирует VF таблицу устройства, что означает, что `placeHooks()` может быть вызвано, когда VF таблица всё ещё содержит функцию `myEndScene()`.

## Написание хука для `Reset()` (*Writing a Hook for `Reset()`*)

Когда вы используете хук `Direct3D` в продакшене, вам предстоит выполнять множество задач, таких как отрисовка собственного текста, отображение изображений, связанных с вашим ботом, и взаимодействие с вызовами функций из игры. Эти задачи потребуют от вас создания собственных объектов `Direct3D`, связанных с устройством игры, что может стать проблемой. Время от времени игра может полностью сбросить своё устройство через функцию `Reset()`. Когда устройство сбрасывается, вам нужно будет обновить все объекты (чаще всего шрифты и спрайты), которые вы создали для устройства, используя их функции-члены `OnLostDevice()`.

Поскольку `Reset()` вызывается из VF таблицы устройства, вы можете использовать хук на эту функцию, чтобы узнать, когда устройство было сброшено. `Reset()` принимает два параметра и находится на индексе 16 в VF таблице. Вы можете добавить этот код в `placeHooks()` из списка 8-14, чтобы перехватить функцию `Reset()`:

---

```
auto ret = hookVF((DWORD)discoveredDevice, 16, (DWORD)&myReset);
if (ret != (DWORD)&myReset)
    origReset = (_reset)ret;
```

---

И вот объявление для использования с `origReset`:

---

```
typedef HRESULT (WINAPI* _reset)(  
    LPDIRECT3DDEVICE9 pDevice,  
    D3DPRESENT_PARAMETERS* pPresentationParameters);  
_reset origReset = NULL;
```

---

Когда сброс выполняется успешно, оригинальная функция возвращает *D3D\_OK*. Ваша функция хука распознаёт это и вызывает *OnLostDevice()* соответствующим образом:

```
HRESULT WINAPI myReset(  
    LPDIRECT3DDEVICE9 pDevice,  
    D3DPRESENT_PARAMETERS* pPresentationParameters)  
{  
    auto result = origReset(pDevice, pPresentationParameters);  
    if (result == D3D_OK) {  
        // Вызовите onLostDevice() для всех ваших объектов  
    }  
    return result;  
}
```

## Что дальше? (What's Next?)

Теперь, когда я показал вам, как взять под контроль устройство *Direct3D* игры, вы, вероятно, задаётесь вопросом, что с ним можно делать. В отличие от других примеров в книге, код в этом разделе и примерный код не имеют строгой одноразовой корреляции, но функциональность остается той же. Вот общий взгляд на соотношение этого раздела и кода в примере проекта *Chapter8\_Direct3DHook*.

Файл *DirectXHookCallbacks.h* содержит функции обратного вызова для *EndScene()* и *Reset()*, две функции обратного вызова для других общих функций, а также *trampoline* и *reporter* функции для временного *jmp hook*. Эти функции в целом такие же, как описано в этом разделе, за исключением того, что они вызывают синглтон-класс, определённый в *DirectXHook.h* и *DirectXHook.cpp*. Этот синглтон-класс отвечает за передачу вызовов в оригинальные функции.

Этот класс также отвечает за всю основную работу, и в нём содержится код для создания устройства поиска, размещения хуков, рисования текста, обработки сбросов устройства и отображения изображений. Более того, он позволяет добавлять внешний код для добавления пользовательских функций обратного вызова для каждого хука, как вы можете видеть в *main.cpp*. Здесь вы увидите множество различных функций обратного вызова, которые рисуют пользовательский текст, добавляют новые изображения на экран и изменяют текстуры моделей, рисуемых игрой. Рекомендую покопаться в коде, чтобы лучше понять, что происходит, но не увлекайтесь слишком сильно. Мы углубимся в этот код в разделе 9, чтобы поговорить обо всех

крутых хаков, которые он может выполнять.

#### Дополнительные исправления для стабильности (OPTIONAL FIXES FOR STABILITY)

Хуки `Reset()` и `EndScene()`, описанные в этом разделе, должны хорошо работать для любой игры, использующей Direct3D 9, но они немного нестабильны. Если игра попытается выполнить `EndScene()`, когда установлен `jmp` hook, она выйдет из строя, потому что байты изменяются. Есть два способа исправить это. Во-первых, вы можете разместить `jmp` hook через IAT hook на `PeekMessage()`. Это будет работать, потому что размещение IAT hook является безопасной операцией для потоков, но это предполагает, что `PeekMessage()` вызывается только из того же потока, который выполняет рендеринг в Direct3D.

Более безопасный, но более сложный вариант — это пройти по всем потокам игры (подобно тому, как это работает при захвате потока) и использовать `SuspendThread()` для приостановки всех потоков в игре (кроме того, который устанавливает хук, разумеется). Прежде чем приостановить поток, вы должны убедиться, что его EIP не выполняет первые 5 байтов `EndScene()`. После того как хук будет установлен, необходимо использовать `ResumeThread()`, чтобы восстановить выполнение с вашим хуком.

### Заключительные мысли (Closing Thoughts)

Манипуляция потоком управления — это очень важный навык в взломе игр, и многие хаки в этой книге зависят от этого. В следующих двух главах вы научитесь создавать общие хаки, используя хук *Direct3D*, и получите лучшее представление о общих сценариях использования хуков. Даже если вы чувствуете себя немного неуверенно, продолжайте изучать Главу 9. Примеры кода в этой главе сосредоточены на хуке *Direct3D* и помогут вам ещё больше ознакомиться с техникой перехвата.

## **Часть 4: Создание бота**

## 9) Использование экстрасенсорного восприятия для устранения тумана войны



Туман войны (*Fog of war*) (часто сокращается просто до *fog*) — это механизм, который разработчики игр обычно используют для ограничения ситуационной осведомлённости игрока и скрытия информации об игровом окружении. Туман часто представляет собой буквальное отсутствие видимости в многопользовательских онлайн-аренах (МОВА), но этот концепт также включает в себя любое отсутствие или неясность важной игровой информации. Замаскированные фигуры, тёмные комнаты и враги, прячущиеся за стенами, — всё это формы тумана.

Хакеры игр могут уменьшить или даже полностью удалить туман с помощью хака **экстрасенсорного восприятия (ESP)**. Хак *ESP* использует перехват, манипуляции с памятью или оба метода, чтобы заставить игру отображать скрытую информацию. Эти хаки используют тот факт, что некоторые виды тумана часто реализуются на стороне клиента, а не сервера, что означает, что клиент игры всё ещё содержит информацию (полную или частичную) о скрытых элементах.

В этой главе вы узнаете, как реализовать различные виды *ESP*-хаков. Сначала вы научитесь освещать тёмные окружения. Затем вы будете использовать рентгеновское зрение, чтобы видеть сквозь стены. Наконец, вы узнаете о хаках на изменение масштаба, настройке интерфейса и других простых *ESP*-хаках, которые могут раскрывать всевозможную полезную (но скрытую) информацию об

игре, в которую вы играете.

## Основные знания (*Background Knowledge*)

Эта глава знаменует переход от взлома, управления марионетками и реверс-инжиниринга к программированию. С этого момента вы будете учиться, как на самом деле писать код для своих хаков. Чтобы не отходить от темы, всё, о чём я говорил до сих пор, будет рассматриваться как базовые знания. Если вы видите используемую технику, которую не совсем помните, например, сканирование памяти, установку точек останова в памяти, перехват или запись памяти, вернитесь к соответствующим главам и изучите их немного подробнее, прежде чем продолжать. В тексте вы найдёте заметки, которые помогут вам вспомнить, где можно освежить определённые темы.

В частности, в этой главе будет много говориться о *Direct3D*. В разделе «*Applying Jump Hooks and VF Hooks to Direct3D*» на странице 175 я объяснил, как перехватывать игровой цикл *Direct3D*. Пример кода для этой главы включает полностью функциональный движок перехвата *Direct3D* в файле *GameHackingExamples/Chapter8\_Direct3DHook*. Многие хаки в этой главе строятся на этом перехвате, а их примерный код можно найти в файле *main.cpp* с кодом перехвата *Direct3D*. Вы можете запустить скомпилированное приложение из *GameHackingExamples/Chapter8\_Direct3DApplication*, чтобы увидеть хаки в действии на тестовом приложении.

## Раскрытие скрытых деталей с помощью Lighthacks (Revealing Hidden Details with Lighthacks)

*Lighthacks* увеличивают освещённость в тёмных окружениях, позволяя вам чётко видеть врагов, сундуки с сокровищами, пути и всё остальное, что обычно скрыто в темноте. Освещение часто представляет собой косметическое изменение, добавляемое на графический уровень игры, и его обычно можно изменить напрямую, используя перехват на графическом слое.

Оптимальное освещение зависит от ориентации камеры, структуры окружения и даже от особенностей игрового движка, и вы можете манипулировать любым из этих факторов, чтобы создать *Lighthacks*. Но самый простой способ — просто добавить больше света в комнату.

## Добавление центрального источника окружающего света (Adding a Central Ambient Light Source)

Онлайн-ресурсы для этой книги включают два небольших примера *Lighthack*. Первый — это функция *enableLightHackDirectional()* в файле *main.cpp*, которая показана в списке 9-1.

---

```
void enableLightHackDirectional(LPDIRECT3DDEVICE9 pDevice)
{
    D3DLIGHT9 light;

    ZeroMemory(&light, sizeof(light));
    light.Type = D3DLIGHT_DIRECTIONAL;
    light.Diffuse = D3DXCOLOR(0.5f, 0.5f, 0.5f, 1.0f);
    light.Direction = D3DXVECTOR3(-1.0f, -0.5f, -1.0f);

    pDevice->SetLight(0, &light);
    pDevice->LightEnable(0, TRUE);
}
```

---

Листинг 9-1: Направленный фонарь

**NOTE**

В примере приложения свет светит вверх и вправо от нижнего левого угла сцены. Возможно, вам потребуется изменить это расположение в зависимости от того, как рендерится ваша целевая игра.

Заметьте, что установка света на индекс 0 работает для демонстрации концепции, но это не всегда будет работать. В играх обычно определено несколько источников света, и установка вашего света на индекс, который использует игра, может переопределить важные эффекты освещения. На практике вы можете попробовать установить индекс на произвольно большое число. Однако у этого типа *lighthack* есть проблема: направленные источники света будут блокироваться объектами, такими как стены, существа и рельеф, что означает, что тени всё равно могут отбрасываться. Направленный свет хорошо работает для открытых пространств, но не так хорошо для узких извилистых коридоров или подземных пещер.

## Увеличение абсолютного окружающего света (Increasing the Absolute Ambient Light)

Другой метод *lighthack*, представленный в функции *enableLightHackAmbient()*, гораздо более агрессивен, чем тот, что показан в списке 9-1. Он влияет на уровень освещения глобально, а не добавляет дополнительный источник света. Вот как выглядит код:

---

```
void enableLightHackAmbient(LPDIRECT3DDEVICE9 pDevice)
{
    pDevice->SetRenderState(D3DRS_AMBIENT, D3DCOLOR_XRGB(100, 100, 100));
}
```

---

Этот *lighthack* устанавливает абсолютный окружающий свет (что указывается передачей D3DRS\_AMBIENT в функцию SetRenderState()) на белый цвет средней интенсивности. Макрос D3DCOLOR\_XRGB задаёт эту интенсивность, принимая 100 в

качестве параметров для уровней красного, зелёного и синего цветов. Это освещает объекты, используя всенаправленный белый свет, эффективно раскрывая всё, но за счёт теней и других деталей, зависящих от освещения

## Создание других типов Lighthacks (Creating Other Types of Lighthacks)

Существует множество способов создания *lighthacks*, но они отличаются от игры к игре. Один из творческих способов повлиять на освещение в игре — это *NOP* кода, который игра использует для вызова функции *device->SetRenderState()*. Поскольку эта функция используется для установки глобальной интенсивности окружающего света, отключение вызовов к ней оставляет *Direct3D* на настройках освещения по умолчанию и делает всё видимым. Это, возможно, самый мощный тип *lighthack*, но он требует, чтобы ваш бот знал адрес кода освещения для его *NOP*.

Также существуют *lighthacks*, основанные на изменении памяти. В некоторых играх игроки и существа излучают свет разного цвета и интенсивности, часто в зависимости от таких атрибутов, как их экипировка, ездовое животное или активные заклинания. Если вы понимаете структуру списка существ в игре, вы можете напрямую изменять значения, которые определяют уровень освещения существ.

Например, представьте игру, в которой персонажи излучают голубоватый шар света, когда находятся под воздействием заклинания лечения или усиления. Где-то в памяти игры хранятся значения, связанные с каждым существом, которые говорят игре, какого цвета и с какой интенсивностью свет должен излучаться. Если вы сможете найти эти значения в памяти, вы сможете изменить их так, чтобы существа эффективно излучали световые шары. Этот тип *lighthack* часто используется в играх с **2D-видом сверху**, так как шары вокруг отдельных существ создают красивый художественный эффект, освещая важные части экрана. В **3D-играх**, однако, этот хак просто превращает существ в светящиеся пятна, которые бегают по экрану.

Вы также можете перехватить функцию *SetLight()* в **индексе 51** в **VF-таблице** устройства *Direct3D* игры. Затем, каждый раз, когда вызывается ваш обработчик перехвата, вы можете изменять свойства перехваченной структуры *D3DLIGHT9* перед её передачей в оригинальную функцию. Например, вы можете изменить все источники света на тип *D3DLIGHT\_POINT*, заставляя любой существующий источник света в игре излучать свет во всех направлениях, как лампочка. Этот тип *lighthack* очень мощный и точный, но он может создавать странные визуальные эффекты. Также он часто выходит из строя в средах, где нет освещения, и непрозрачные препятствия всё равно будут блокировать точечные источники света.

*Lighthacks* очень мощны, но они не раскрывают скрытую информацию. Если информация скрыта за препятствием, а не темнотой, вам потребуется *wallhack*, чтобы её увидеть.

## Обнаружение скрытых врагов с помощью Wallhacks (Revealing Sneaky Enemies with Wallhacks)

Вы можете использовать *wallhacks*, чтобы показывать врагов, скрытых за стенами, полами и другими препятствиями. Существует несколько способов создания таких хаков, но наиболее распространённый метод использует тип рендеринга, известный как *z-buffering*.

## Рендеринг с Z-буферизацией (Rendering with Z-Buffering)

Большинство графических движков, включая *Direct3D*, поддерживают *z-buffering*, который используется для того, чтобы при наложении объектов в сцене рисовался только верхний объект. *Z-buffering* работает за счёт "отрисовки" сцены в двумерный массив, который описывает, насколько близко объект в каждом пикселе экрана находится к зрителю. Представьте, что индексы массива — это оси: они соответствуют оси *X* (вправо и влево) и оси *Y* (вверх и вниз) для каждого пикселя на экране. Каждое значение, хранящееся в массиве, представляет собой координату по оси *Z* для конкретного пикселя.

Когда появляется новый объект, *z-буфер* решает, будет ли он фактически нарисован на экране. Если место в массиве по координатам *X* и *Y* уже занято другим значением, это означает, что в этом пикселе на экране уже присутствует другой объект. Новый объект появится только в том случае, если его значение по оси *Z* будет меньше (то есть он окажется ближе к зрителю), чем пиксель, который там уже находится. Когда сцена завершает отрисовку в массив, данные *z-буфера* отправляются на экран.

Чтобы проиллюстрировать это, представьте трёхмерное пространство, которое нужно отрисовать на двумерном экране в игре с окном просмотра размером **4×4 пикселя**. *Z-буфер* для этого сценария будет выглядеть так, как показано на Рисунке 9-1.

(0,0)	(3,0)
$z = 0$ No color	$z = 0$ No color
$z = 0$ No color	$z = 0$ No color
$z = 0$ No color	$z = 0$ No color
$z = 0$ No color	$z = 0$ No color
(0,3)	(3,3)

Рисунок 9-1: Пустой z-буфер

Сначала игра рисует синий фон, который полностью заполняет область просмотра (*viewport*) и расположен как можно дальше по оси  $Z$ ; допустим, самое высокое значение  $Z$  равно 100. Затем игра рисует **2×2-пиксельный красный прямоугольник** в позиции  $(0,0)$  с  $Z$ -координатой 5. Наконец, игра рисует **2×2-пиксельный зелёный прямоугольник** в позиции  $(1,1)$  с  $Z$ -координатой 3. Теперь  $Z$ -буфер будет выглядеть, как показано на Рисунке 9-2.

(0,0)	(3,0)
$z = 5$ Red	$z = 5$ Red
$z = 5$ Red	$z = 3$ Green
$z = 100$ Blue	$z = 3$ Green
$z = 100$ Blue	$z = 100$ Blue
(0,3)	(3,3)

Figure 9-2: Заполненный z-буфер

## Дословный перевод:

*Z*-буфер аккуратно обработал перекрывающиеся объекты, основываясь на их координатах по оси *Z*. **Зелёный квадрат**, который находится ближе всего по оси *Z*, перекрывает **красный квадрат**, который расположен немного дальше, а оба квадрата перекрывают **синий фон**, который находится очень далеко.

Такое поведение позволяет игре отрисовывать свою карту, игроков, существ, детали и частицы, не беспокоясь о том, что на самом деле видно игроку. Это огромная оптимизация для разработчиков игр, но она открывает большую область для атак. Поскольку все игровые модели *всегда* передаются в графический движок, вы можете использовать перехваты (*hooks*), чтобы обнаруживать объекты, которые игрок на самом деле не видит.

## Создание Direct3D Wallhack (Creating a Direct3D Wallhack)

Вы можете создавать *wallhacks*, которые манипулируют *z*-буферизацией в *Direct3D*, используя перехват функции *DrawIndexedPrimitive()*, которая вызывается, когда игра рисует **3D-модель** на экране. Когда отрисовывается модель вражеского игрока, *wallhack* такого типа **отключает z-буфер**, вызывает оригинальную функцию для отрисовки модели, а затем **снова включает z-буфер**. Это приводит к тому, что модель врага рисуется **поверх всего остального** в сцене, независимо от того, что находится перед ней. Некоторые *wallhacks* также могут рендерить определённые модели в **однотонный цвет**, например, красный для врагов и зелёный для союзников.

## Переключение Z-буферизации (Toggling Z-Buffering)

*Direct3D*-перехват в файле *main.cpp* из *GameHackingExamples/Chapter8\_Direct3DHook* содержит следующий пример *wallhack* в функции *onDrawIndexedPrimitive()*:

```
void onDrawIndexedPrimitive(
    DirectXHook* hook,
    LPDIRECT3DDEVICE9 device,
    D3DPRIMITIVETYPE primType,
    INT baseVertexIndex, UINT minVertexIndex,
    UINT numVertices, UINT startIndex, UINT primCount)
{
    if (numVertices == 24 && primCount == 12) {
        // это враг, выполняем wallhack
    }
}
```

Эта функция используется в качестве обратного вызова (*callback*) для перехвата *DrawIndexedPrimitive()* в **VF-индексе 82** устройства *Direct3D* игры. Каждая модель, отрисовываемая в игре, проходит через эту функцию вместе с определёнными свойствами модели. Анализируя подмножество свойств, а именно значения *numVertices* и *primCount*, перехват обнаруживает, когда

отрисовывается модель врага, и запускает wallhack. В этом примере значения, представляющие вражескую модель, равны 24 и 12.

Магия происходит внутри if(). Используя всего несколько строк кода, wallhack рисует модель таким образом, чтобы игнорировать z-буферизацию, например, так:

```
device->SetRenderState(D3DRS_ZENABLE, false); // отключаем z-буферизацию
DirectXHook::origDrawIndexedPrimitive(          // отрисовываем модель
    device, primType, baseVertexIndex,
    minVertexIndex, numVertices, startIndex, primCount);
device->SetRenderState(D3DRS_ZENABLE, true); // включаем z-буферизацию
```

Проще говоря, этот код отключает z-буферизацию перед отрисовкой модели врага и включает её снова после. При выключенной z-буферизации модель врага рисуется поверх всего остального.

## Изменение текстуры врага (*Changing an Enemy Texture*)

Когда модель рендерится на экране, используется текстура (texture), которая применяется к модели. Текстуры представляют собой 2D-изображения, которые натягиваются на 3D-модели, добавляя цвета и узоры, формирующие 3D-графику модели. Чтобы изменить внешний вид врага при его отрисовке в вашем wallhack, можно указать для него другую текстуру, как показано в следующем примере:

```
// при инициализации перехвата
LPDIRECT3DTEXTURE9 red;
D3DXCreateTextureFromFile(device, "red.png", &red);

// перед отрисовкой примитива
device->SetTexture(0, red);
```

Первый блок этого кода загружает текстуру из файла и выполняется только один раз – во время инициализации перехвата. Полный пример кода делает это в функции initialize(), которая вызывается при первом срабатывании EndScene(). Второй блок кода выполняется непосредственно перед вызовом оригинальной функции DrawIndexedPrimitive() в wallhack-коде, и заставляет модель отрисовываться с новой текстурой.

## Фингерпринтинг модели, которую вы хотите раскрыть (*Fingerprinting the Model You Want to Reveal*)

Самая сложная часть при создании хорошего wallhack – это поиск правильных значений для numVertices и primCount. Для этого можно создать инструмент, который логирует каждую уникальную комбинацию этих двух переменных и позволяет вам перебирать список с клавиатуры. Рабочий пример кода для такого инструмента не пригодится в тестовом приложении, прилагаемом

к этой главе, но ниже приведены основные принципы реализации.

Во-первых, в глобальной области необходимо объявить структуру, содержащую следующие данные:

numVertices и primCount

std::set этой структуры (назовём её seenParams)

Экземпляр этой структуры (назовём его currentParams)

Структура std::set требует функтор-компаратор, поэтому необходимо объявить функтор сравнения, вызывающий temstrcmp() для сравнения двух структур с использованием temstrcmp().

Каждый раз, когда вызывается DrawIndexedPrimitive(), wallhack может создавать экземпляр структуры с перехваченными значениями, передавать их в seenParams.insert(), который добавит пару в список, если её там ещё нет.

Используя функцию GetAsyncKeyState() API Windows, можно определить нажатие пробела и выполнить следующий псевдокод:

```
auto current = seenParams.find(currentParam);
if (current == seenParams.end())
    current = seenParams.begin();
else
    current++;
currentParams = *current;
```

Это установит currentParams на следующую пару в seenParams, когда будет нажат пробел. С таким кодом можно использовать логику wallhack'a, чтобы изменять текстуры моделей, соответствующих currentParams.numVertices и currentParams.primCount. Этот инструмент также мог бы выводить эти значения на экран, чтобы вы могли их увидеть и записать.

С помощью такого инструмента поиск нужных моделей будет таким же простым, как запуск игры в режиме, где ваш персонаж не умирает (игра против друга, режим кастомизации и т. д.), запуск бота и нажатие пробела, пока не будут выделены все нужные модели.

Как только будут известны нужные значения, необходимо изменить проверку numVertices и primCount в вашем wallhack'e, чтобы он знал, какие модели выделять.

#### НОТЕ

Модели персонажей часто состоят из более мелких моделей для отдельных частей тела, и игры часто используют разные модели персонажей на разных расстояниях. Это означает, что в игре может быть 20 или более моделей для одного типа персонажа. В этом случае выбор только одной модели (например, торса врага) для отображения в вашем wallhack'e может быть достаточным.

## Использование NOPing Zoomhacks

Игры MOBA и RTS обычно позволяют игрокам изменять масштаб в определенных пределах. Самый простой способ zoomhack'a заключается в нахождении переменной, отвечающей за zoom factor (коэффициент масштабирования, изменяющийся при смене уровня зума, обычно float или double) и замене его на большее значение.

Чтобы найти zoom factor, откройте Cheat Engine и поиск выполните по неизвестному начальному значению типа float. (Чтобы освежить знания по Cheat Engine, откройте "Cheat Engine's Memory Scanner" на стр. 5). Далее повторяйте этот процесс, пока не останется всего несколько значений:

1. Откройте окно игры и приблизьте камеру.
2. Выполните поиск увеличенного значения в Cheat Engine.
3. Откройте окно игры и отдалите камеру.
4. Выполните поиск уменьшенного значения в Cheat Engine.

Попробуйте оставить одно возможное значение. Чтобы убедиться, что найденное значение — это zoom factor, заморозьте его в Cheat Engine и проверьте поведение камеры в игре; если заморозка отключает масштабирование, значит вы нашли правильное значение. Если поиск по float'у не дал результатов, повторите его, но используйте double.

Если оба поиска не помогли, попробуйте снова, но меняйте местами поиск увеличенных и уменьшенных значений. Когда найдете zoom factor в памяти, можно написать небольшой бот, который будет изменять его на удобное значение.

Более продвинутые zoomhacks NOP'ят код игры, отвечающий за ограничение zoom factor в определенном диапазоне. Вы можете найти этот код в OllyDbg. Установите memory-on-write breakpoint на zoom factor, затем измените масштаб в игре, чтобы активировать breakpoint, и проанализируйте код. (Чтобы улучшить навыки работы с memory breakpoint'ами в OllyDbg, обратитесь к "Controlling OllyDbg Through the Command Line" на стр. 43).

Вы должны увидеть код, который изменяет zoom factor. Ограничение зума обычно легко обнаружить: константы, которые задают минимальные и максимальные значения зума, — верный признак.

Если таким способом не удалось найти код ограничения, оно может применяться при перерисовке графики на новом уровне зума, а не при изменении zoom factor. В таком случае поменяйте breakpoint на memory-on-read и ищите те же признаки.

## **Поверхностный взгляд на Hooking Zoomhacks (Scratching the Surface of Hooking Zoomhacks)**

Вы также можете создавать zoomhacks, используя Direct3D-хук для функции `device->SetTransform(type, matrix)`, но это требует глубокого понимания того, как игра настраивает перспективу игрока. Существует несколько способов управления перспективой, но уровень зумирования можно контролировать через *view* (тип трансформации `D3DTS_VIEW`) или *projection* (тип трансформации `D3DTS_PROJECTION`).

Правильное управление матрицами трансформации, которые контролируют вид и проекцию, требует довольно глубоких знаний математики, лежащей в основе 3D-графики. Поэтому я стараюсь избегать этого метода любой ценой — и никогда не сталкивался с проблемами при обычном изменении *zoom factor*. Если вас интересует такой тип хака, я рекомендую прочитать книгу по программированию 3D-игр, чтобы сначала изучить основы 3D-математики.

Но иногда даже zoomhack'a недостаточно. Некоторая полезная информация может оставаться скрытой как часть внутреннего состояния игры или просто быть трудно различимой для игрока с первого взгляда. В таких ситуациях идеальным инструментом будет heads-up display (HUD).

## **Отображение скрытых данных с помощью HUD (Displaying Hidden Data with HUDs)**

*Heads-up display (HUD)* — это тип ESP-хака, который отображает критически важную информацию об игре в виде наложенного слоя. HUD часто напоминает существующий интерфейс игры, предназначенный для отображения информации, такой как оставшиеся боеприпасы, мини-карта, уровень здоровья, активные перезарядки способностей и так далее. HUD обычно отображает либо исторические, либо агрегированные данные, и чаще всего используется в MMORPG. Чаще всего такие интерфейсы основаны на тексте, но некоторые также содержат спрайты, формы и другие небольшие визуальные эффекты.

HUD, которые вы можете создать, зависят от доступных данных в игре. Обычные точки данных включают:

- Опыт, получаемый в час (*exp/h*).
- Убийства существ в час (*KPH*).
- Урон в секунду (*DPS*).
- Добыча золота в час (*GPH*).
- Исцеление в минуту.
- Ориентировочное время до следующего уровня.
- Количество золота, потраченного на припасы.
- Общая стоимость собранных предметов в золоте.

Более продвинутые пользовательские HUD могут отображать большие таблицы с содержимым, включая собранные предметы,

использованные припасы, количество убийств для каждого типа существ и имена игроков, которые недавно были замечены.

Помимо того, что вы уже узнали о чтении памяти, хука графических движков и отображении настроенных данных, не так уж много, что я могу добавить о создании HUD. В большинстве игр архитектура достаточно проста, чтобы вы могли легко извлекать из памяти нужную информацию. Затем вы можете выполнять базовые почасовые, процентные или суммирующие вычисления, чтобы привести данные в удобный формат.

## Создание HUD опыта (Creating an Experience HUD)

Представьте, что вам нужен HUD, который отображает ваш текущий уровень, почасовой опыт и время, оставшееся до повышения уровня персонажа. Сначала вы можете использовать Cheat Engine, чтобы найти переменные, содержащие ваш уровень и опыт. Когда у вас есть эти значения, вы можете использовать либо специфичный для игры алгоритм, либо жестко заданную таблицу опыта для расчета количества опыта, необходимого для следующего уровня.

Когда вы знаете, сколько опыта нужно для повышения уровня, вы можете рассчитать почасовой опыт. В псевдокоде этот процесс может выглядеть так:

```
// этот пример предполагает, что время хранится в миллисекундах
// для секунд уберите "1000 * "
timeUnitsPerHour = 1000 * 60 * 60
timePassed = (currentTime - startTime)

❶ timePassedToHourRatio = timeUnitsPerHour / timePassed
❷ expGained = (currentExp - startExp)
    hourlyExp = expGained * timePassedToHourRatio

❸ remainingExp = nextExp - currentExp
❹ hoursToGo = remainingExp / hourlyExp
```

Чтобы найти ваш почасовой опыт (`hourlyExp`), вы должны сохранить ваш опыт и время, когда ваш HUD впервые запускается; это `startExp` и `startTime` соответственно. В этом примере также предполагается, что `currentLevel` и `currentExp` уже определены, где `currentLevel` — это уровень персонажа, а `currentExp` — текущее количество опыта.

С этими значениями `hourlyExp` можно вычислить, умножив коэффициент ❶ количества временных единиц в часе на прошедшее время и разделив на полученный опыт с момента `startTime` ❷. В этом случае временная единица — это миллисекунда, поэтому временные единицы умножаются на 1000.

Далее `currentExp` вычитается из `nextExp`, чтобы определить

оставшийся опыт ③ до повышения уровня. Чтобы вычислить, сколько часов осталось до повышения уровня, оставшийся опыт делится на ваш почасовой опыт ④.

Когда у вас есть вся эта информация, вы можете наконец отобразить её на экране. Используя Direct3D-хук, представленный в коде примера этой книги, вы можете нарисовать текст с помощью этого вызова внутри обработчика хука EndScene().

```
hook->drawText(  
    10, 10,  
    D3DCOLOR_ARGB(255, 255, 0, 0),  
    "Will reach level %d in %.20f hours (%d exp per hour)",  
    currentLevel, hoursToGo, hourlyExp);
```

Это всё, что вам нужно для работающего HUD, отслеживающего опыт. Вариации этих же уравнений можно использовать для вычисления КРН, DPS, GPH и других полезных показателей, основанных на времени.

Кроме того, вы можете использовать функцию drawText() хука Direct3D для отображения любой информации, которую вы можете найти и нормализовать. Хук также содержит функции addSpriteImage() и drawSpriteImage(), которые можно использовать для рисования собственных изображений, позволяя вам сделать ваш HUD таким стильным, как вам хочется.

## Использование хуков для поиска данных (Using Hooks to Locate Data)

Чтение памяти — не единственный способ получить данные для кастомного HUD. Вы также можете собирать информацию, подсчитывая, сколько раз определённая модель отрисовывается функцией **DrawIndexedPrimitive()**, перехватывая внутренние функции игры, отвечающие за рендеринг определённого типа текста, или даже перехватывая вызовы функций, обрабатывающих пакеты данных с игрового сервера.

Методы, которые вы будете использовать для этого, будут значительно различаться в каждой игре, и их поиск потребует от вас объединить все знания из этой книги с вашей собственной находчивостью и интуицией программирования.

Например, чтобы создать HUD, который отображает количество врагов на карте, вы можете использовать методы **идентификации моделей**, применяемые в **wallhack**, чтобы посчитать количество врагов и вывести это число на экран. Этот метод лучше, чем создание способа чтения списка врагов из памяти, так как он не требует поиска новых адресов памяти после каждого обновления игры.

Другим примером является отображение списка перезарядок у врагов, что потребует от вас перехвата входящих пакетов, которые сообщают клиенту, какие визуальные эффекты заклинаний отображать. Затем можно соотнести определённые заклинания с определёнными врагами на основе местоположения врага, типа заклинания и других параметров, а затем использовать эту информацию для отслеживания, какие заклинания использовал

каждый враг. Если сопоставить эти данные с базой данных времени перезарядки, можно точно показать, когда враг сможет снова использовать заклинание. Это особенно эффективно, поскольку большинство игр не хранят перезарядки врагов в памяти.

## Обзор других ESP-хаков

Помимо хаков, рассмотренных в этой главе, существует ряд ESP-хаков, которые не имеют общепринятых названий и специфичны для определённых жанров или даже отдельных игр. Я кратко объясню теорию, предысторию и архитектуру некоторых из этих хаков.

### **Хаки дальности**

Хаки дальности используют метод, схожий с wallhack, для определения моментов, когда в игре отрисовываются модели разных типов чемпионов или героев. Затем они рисуют круги на земле вокруг каждой модели героя. Радиус каждого круга соответствует максимальной дальности атаки окружаемого чемпиона или героя, эффективно показывая зоны, где вас может атаковать каждый противник.

### **HUD-ы экрана загрузки**

HUD-ы экрана загрузки распространены в MOBA и RTS-играх, которые требуют от всех игроков ожидания на экране загрузки, пока игра запускается. Эти хаки используют тот факт, что такие игры часто имеют веб-сайты, где можно запрашивать историческую статистику игроков. Можно написать бота, который автоматически получает статистику каждого игрока в матче и бесшовно отображает информацию в виде оверлея на экране загрузки, позволяя вам изучать своих врагов перед боем.

### **HUD-ы фазы выбора**

HUD-ы фазы выбора похожи на своих собратьев с экрана загрузки, но отображаются во время предматчевой стадии, когда каждый игрок выбирает чемпиона или героя для игры. Вместо отображения статистики врагов, HUD-ы фазы выбора показывают статистику союзников. Это позволяет вам быстро оценить сильные и слабые стороны вашей команды, чтобы принять более обоснованное решение о том, какого персонажа выбрать.

### **Хаки шпионского пола**

Хаки шпионского пола распространены в старых 2D-играх с видом сверху, где есть несколько отдельных этажей или платформ. Если вы находитесь на верхнем этаже, вам может понадобиться узнать, что происходит внизу, прежде чем спускаться туда. Можно написать хак шпионского пола, который изменяет значение текущего этажа (обычно это unsigned int) на другой этаж выше или ниже, что позволяет вам шпионить за другими этажами.

Игры часто пересчитывают значение текущего этажа каждый кадр на основе позиции игрока, поэтому иногда требуется использование NOP-инструкций, чтобы предотвратить его сброс при каждом перерисовывании кадра. Поиск текущего значения этажа и кода, который необходимо NOP-ировать, будет аналогичен

поиску коэффициента масштабирования, как описано в разделе "Использование NOP-хака масштабирования" на странице 197.

## **Заключительные мысли**

ESP-хаки — это мощный способ получения дополнительной информации об игре. Некоторые из них можно реализовать довольно просто с помощью Direct3D-хуков или простого редактирования памяти. Другие требуют изучения внутренних структур данных игры и перехвата закрытых функций, что даёт вам повод применять свои навыки реверс-инжиниринга.

Если вы хотите поэкспериментировать с ESP-хаками, изучите и настройте пример кода из этой главы. Чтобы потренироваться с более специфическими ESP-хаками, я советую вам попробовать разные игры и поиграть с возможностями хакинга.

## 10) Отзывчивые хаки



Среднестатистический игрок имеет время реакции 250 миллисекунд, или четверть секунды.

Профессиональные геймеры в среднем реагируют за пятую часть

секунды, но некоторые могут среагировать за шестую часть секунды. Эти показатели основаны на онлайн-тестах, которые измеряют время реакции игроков на одиночные, предсказуемые события.

Однако в реальных играх игрокам приходится реагировать на десятки различных событий, таких как потеря здоровья, входящие умения, способности, выходящие из перезарядки, атаки врагов и многие другие. Только очень опытные игроки могут поддерживать реакцию на уровне четверти или пятой части секунды в таких динамичных условиях; единственный способ быть быстрее — это быть компьютером.

В этой главе вы узнаете, как создать ботов, которые реагируют быстрее любого игрока. Сначала я покажу вам некоторые шаблоны кода, которые можно встроить в бота для обнаружения определённых событий в игре. Затем вы научитесь создавать бота, который перемещает вашего персонажа, лечит или применяет заклинания самостоятельно. После того как вы освоите эти базовые техники, я помогу вам объединить их, чтобы реализовать одни из самых распространённых и мощных отзывчивых хаков.

### Наблюдение за игровыми событиями

Всего за несколько секунд игры большинство людей могут сделать важные наблюдения о игровом окружении. Вы можете ясно видеть, когда ракеты летят в сторону вашего персонажа, когда у вас слишком мало здоровья и когда способности выходят из перезарядки. Однако для бота такие, казалось бы, интуитивные наблюдения не так просто осуществить. Бот должен обнаруживать каждое событие, отслеживая изменения в памяти, анализируя визуальные подсказки или перехватывая сетевой трафик.

### Мониторинг памяти

Чтобы обнаружить простые события, например, падение уровня здоровья, можно запрограммировать бота на периодическое чтение значения здоровья из памяти и сравнение его с

минимальным допустимым значением, как в примере кода 10-1.

```
// Выполняется каждые 10 миллисекунд (100 раз в секунду)
auto health = readMemory<int>(HEALTH_ADDRESS);
if (health <= 500) {
    // код, который подскажет боту, как реагировать
}
```

Листинг 10-1: Сайт `if` оператор, который проверяет состояние здоровья

Данный адрес здоровья вашего персонажа можно проверять так часто, как это необходимо; каждые 10 миллисекунд — обычно хорошая частота. (Вернитесь к Главе 1, если вам нужно освежить в памяти, как находить значения в памяти.) Как только **health** (здоровье) опустится ниже определенного значения, потребуется запустить код реакции, чтобы применить лечащую способность или выпить зелье. Позже в этой главе будет рассказано, как это сделать.

Если вам нужно, чтобы бот получал более детальную информацию и имел возможность более гибкого реагирования, его можно запрограммировать на реакцию **на любое** изменение здоровья, а не только на достижение порогового значения. Для этого измените код в Листинге 10-1, добавив сравнение текущего здоровья с его значением в предыдущем цикле выполнения, как показано ниже:

```
// по-прежнему выполняем каждые 10 миллисекунд
static int previousHealth = 0;
auto health = readMemory<int>(HEALTH_ADDRESS);
if (health != previousHealth) {
    if (health > previousHealth) {
        // реагировать на увеличение
    } else {
        // реагировать на уменьшение
    }
    previousHealth = health;
}
```

Теперь этот код использует статическую переменную **previousHealth**, чтобы отслеживать значение **health** (здоровья) в предыдущей итерации. Если **previousHealth** и **health** различаются, бот не только реагирует на изменение здоровья, но и по-разному реагирует

на его увеличение и уменьшение. Этот метод является самым простым и наиболее распространенным способом реагирования на изменения в состоянии игры. Зная правильные адреса памяти, можно использовать этот шаблон кода для отслеживания

изменений здоровья, маны, откатов способностей и другой важной информации.

## Обнаружение визуальных подсказок (Detecting Visual Cues)

Здоровье относительно просто для проверки ботом, потому что это всего лишь число, но некоторые элементы игры должны передаваться боту по-другому. Например, когда на персонажа действуют отрицательные эффекты или баффы, самый простой способ определить это — просто искать индикатор статуса на экране, и то же самое верно для ботов.

Когда чтения памяти недостаточно, можно обнаруживать определенные события, перехватывая графический движок игры и ожидая, когда игра отрендерит определенную модель. (Обратитесь к разделу **“Применение прыжковых хуков и VF хуков в Direct3D”** на странице 175 и **“Создание Direct3D Wallhack”** на странице 194, чтобы освежить информацию о хуках Direct3D.) Когда модель отрисована, можно запланировать реакцию, которая будет выполнена после рендера кадра, как показано ниже:

```
// ниже представлен хук drawIndexedPrimitive
void onDrawIndexedPrimitive(...) {
    if (numVertices == EVENT_VERT && primCount == EVENT_PRIM) {
        // реагировать, желательно после завершения отрисовки
    }
}
```

Используя тот же метод "снятия отпечатков" модели, что и код wallhack в Главе 9, этот код обнаруживает, когда конкретная модель отображается на экране, и реагирует соответствующим образом. Однако этот код реагирует на событие каждый кадр, и это может сделать вашу игру неиграбельной. Вам, вероятно, понадобится внутренний кулдаун, чтобы избежать спама реакцией. В случаях, когда модель индикатора отображается постоянно (то есть не мигает), можно отслеживать ее на протяжении нескольких кадров, чтобы определить моменты появления и исчезновения.

Вот фрагмент кода, который также выполняет отслеживание:

```

bool eventActive = false;
bool eventActiveLastFrame = false;
// ниже находится хук drawIndexedPrimitive
void onDrawIndexedPrimitive(...) {
    if (numVertices == EVENT_VERT && primCount == EVENT_PRIM)
        eventActive = true;
}

// ниже находится хук endScene
void onDrawFrame(...) {
    if (eventActive) {
        if (!eventActiveLastFrame) {
            // реагировать на появление модели события
        }
        eventActiveLastFrame = true;
    } else {
        if (eventActiveLastFrame) {
            // реагировать на исчезновение модели события
        }
        eventActiveLastFrame = false;
    }
    eventActive = false;
}

```

Функция `onDrawIndexedPrimitive()` по-прежнему проверяет, была ли нарисована определённая модель, но теперь два логических значения отслеживают, была ли модель нарисована в этом кадре или в предыдущем. Затем, когда кадр полностью отрисован, бот может проверить эти переменные и среагировать на появление или исчезновение модели.

Этот метод отлично подходит для обнаружения визуальных индикаторов состояния, которые появляются только тогда, когда ваш персонаж находится под действием оглушения, замедления передвижения, ловушек, ядов и так далее. Вы также можете использовать его, чтобы определить, когда враги появляются и исчезают в МОВА- и RTS-играх, поскольку такие игры отрисовывают только врагов, которые находятся в зоне видимости союзного юнита или игрока.

## Перехват сетевого трафика (Intercepting Network Traffic)

Один из самых надёжных способов наблюдать за событиями — это использовать тот же метод, что и игровой клиент: ожидать, пока сервер игры сообщит об их возникновении. В таком типе взаимодействия игровой сервер отправляет байтовые массивы, называемые **пакетами (packets)**, через сеть клиенту, используя сокеты. Эти пакеты обычно зашифрованы и содержат массивы данных, сериализованных в проприетарном формате.

## Обычная функция разбора пакетов (A Typical Packet-Parsing Function)

Чтобы получать и обрабатывать пакеты, игровой клиент выполняет что-то подобное **Листингу 10-2** перед тем, как отрисовать кадр.

```
void parseNextPacket() {  
    if (!network->packetReady()) return; // если пакеты отсутствуют, выйти из функции  
  
    auto packet = network->getPacket(); // получить пакет из сети  
    auto data = packet->decrypt(); // расшифровать данные пакета  
    switch (data->getType()) { // определить тип пакета  
        case PACKET_HEALTH_CHANGE: // изменение здоровья  
            onHealthChange(data->getMessage()); // обработать изменение здоровья  
            break;  
        case PACKET_MANA_CHANGE: // изменение маны  
            onManaChange(data->getMessage()); // обработать изменение маны  
            break;  
        // другие случаи для других типов пакетов  
    }  
}
```



Листинг 10-2: Упрощенный взгляд на то, как игра разбирает пакеты

## Точный код для каждой игры может отличаться

Но поток управления всегда один и тот же: получить пакет, расшифровать его, определить, какое сообщение он содержит, и вызвать функцию, которая знает, что с ним делать. Некоторые игровые хакеры перехватывают необработанные сетевые пакеты и воспроизводят эту функциональность в своих ботах. Этот метод работает, но требует глубоких знаний шифрования, полного понимания того, как игра хранит данные внутри пакетов, способности выполнять **man-in-the-middle**-атаку на сетевое соединение и способа определения ключей расшифровки, используемых игровым клиентом.

Хукать функции, которые отвечают за обработку пакетов после их расшифровки и разбора, — гораздо лучший подход. В **Листинге 10-2** такими функциями являются **onHealthChange()** и **onManaChange()**. Этот метод использует встроенную способность игры обрабатывать пакеты, позволяя боту оставаться неосведомлённым о различных сетевых механиках, используемых игрой. Он также даёт возможность выборочно перехватывать только те сетевые данные, которые вам нужны, так как можно подключаться только к нужным обработчикам.

**НОТЕ**

Перехват целых пакетов иногда может быть выгодным – например, в любой игре, использующей Adobe AIR и взаимодействующей через RTMPS. Поскольку RTMPS хорошо задокументирован, нет необходимости проводить обратный инжиниринг формата или шифрования. В Главе 8 подробно объясняется, как хукать RTMPS.

## Несколько хитростей для поиска функции парсинга

Есть несколько трюков, которые можно использовать, чтобы легко найти функцию парсинга и, в конечном итоге, оператор **switch()**, который распределяет пакеты по их обработчикам. Самый полезный метод, который я нашёл, — это установить **breakpoint** на функцию, которую игра использует для получения данных из сети, а затем проанализировать поток выполнения программы, когда срабатывает точка останова.

Давайте разберём, как это можно сделать с помощью **OllyDbg**, подключенного к вашей целевой игре. В **Windows** функция **recv()** — это API-функция для получения данных из сокета. В командной строке **OllyDbg** можно установить точку останова на **recv()**, введя команду `bpr recv`. Когда **breakpoint** срабатывает, можно подниматься вверх по **call stack**, используя **CTRL+F9** (ярлык для выполнения до возврата) и **F8** (ярлык для пошагового выполнения).

Эта комбинация позволяет программе выполнятся до возврата вызванной функции к вызывающей, что даёт возможность просматривать стек вызовов параллельно с работой игры. На каждом уровне стека можно инспектировать код вызывающих функций, пока не найдётся та, в которой есть большой оператор **switch()**. Скорее всего, это и будет функция парсинга пакетов.

## Более сложный парсер

В зависимости от архитектуры игры поиск функции парсинга может быть не таким простым. Рассмотрим игру с функцией парсинга, которая выглядит следующим образом:

```
packetHandlers[PACKET_HEALTH_CHANGE] = onHealthChange;
packetHandlers[PACKET_MANA_CHANGE] = onManaChange;

void parseNextPacket()
{
    if (!network->packetReady()) return;
    auto packet = network->getPacket();
    auto data = packet->decrypt();
    auto handler = packetHandlers[data->getType()];
    handler->invoke(data->getMessage());
}
```

Поскольку функция **parseNextPacket()** не содержит оператор **switch()**, в памяти нет очевидного способа её идентификации. Если не быть внимательным, можно легко пропустить её в **call stack**. В играх, где функция парсера устроена таким образом,

попытки определить, как именно она выглядит, могут быть бесполезны. Если при анализе **call stack** для **recv()** не обнаружен **switch()**, придётся записывать все вызываемые функции на стеке.

Вместо того чтобы просто подниматься вверх по **call stack** после срабатывания **breakpoint**, можно перейти ко всем адресам, помеченым как **RETURN** ниже **ESP** в **OllyDbg**. Эти адреса представляют собой точки возврата в вызывающие функции для каждого вызова. На каждом адресе возврата нужно определить верхнюю часть вызывающей функции в **assembly-панели OllyDbg** и записать её адрес. В результате получится список всех вызовов функций, ведущих к **recv()**.

Следующим шагом будет повторение этого процесса, но уже с **breakpoint'ами** на нескольких функциях-обработчиках игры. Найти обработчик можно, отслеживая память, в которую он неизбежно запишет данные. Например, обработчик **пакетов изменения здоровья** обновляет значение **здоровья** в памяти.

С помощью **OllyDbg** можно установить **memory on write breakpoint** для адреса здоровья. Когда **breakpoint** сработает, это означает, что **обработчик обновил здоровье** в памяти. Такой же подход работает и для других важных значений, контролируемых сервером: **мана, уровень персонажа, предметы** и т. д.

После записи **call stack** из **recv()** и нескольких функций-обработчиков можно сравнить их и определить местоположение **функции парсинга**. Например, в **Таблице 10-1** рассмотрены три примера **псевдо-call stack** для таких случаев.

Таблица 10-1: Стеки псевдовызовов для трех функций, связанных с пакетами

recv() stack	onHealthChange() stack	onManaChange() stack
0x0BADFOOD	0x101E1337	0x14141414
0x40404040	0x50505050	0x60606060
0xDEADBEEF	0xDEADBEEF	0xDEADBEEF
0x30303030	0x30303030	0x30303030
0x20202020	0x20202020	0x20202020
0x10101010	0x10101010	0x10101010

Эти стеки показывают, как может выглядеть память во время вызова **recv()**, а также гипотетических функций игры **onHealthChange()** и **onManaChange()**. Обратите внимание, что каждая функция исходит из цепочки из четырёх общих вызовов функций (выделены **жирным**). Самый глубокий общий адрес, **0xDEADBEEF**, является адресом парсера. Для лучшего понимания этой структуры обратитесь к дереву вызовов, представленному на **Рисунке 10-1**.

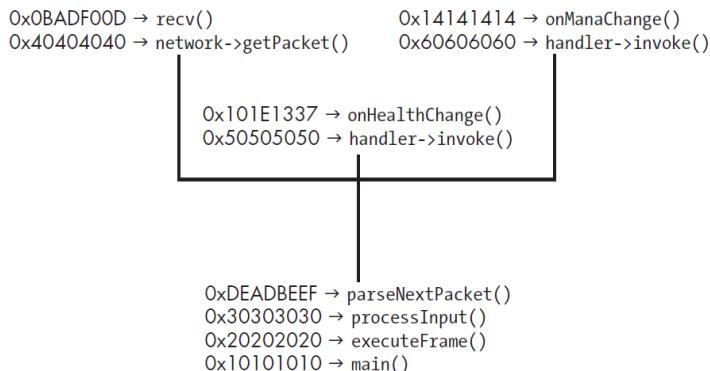


Figure 10-1: Древовидное представление наших трех стеков вызовов

Каждый стек вызовов функций разветвляется из функции по адресу **0xDEADBEEF**, что означает, что эта функция является общей точкой происхождения для всех трёх вызовов. Пример функции **parseNextPacket()** отвечает за вызов этих функций, поэтому она должна быть самым недавним общим предком по адресу **0xDEADBEEF**.

#### НОТЕ

Эти стеки вызовов гипотетические и упрощены по сравнению с тем, что вы обычно встретите. Реальные стеки вызовов, вероятно, будут содержать гораздо большие вызовов функций, и их сравнение будет не таким простым.

## Гибридная система парсинга (A Hybrid Parsing System)

Третий вариант цикла парсинга может представлять собой гибрид двух предыдущих, использующий оператор **switch()** после вызова функции. Вот ещё одна гипотетическая функция:

```

void processNextPacket()
{
    if (!network->packetReady()) return;
    auto packet = network->getPacket();
    auto data = packet->decrypt();
    dispatchPacket(data);
}

void dispatchPacket(data)
{
    switch (data->getType()) {
        case PACKET_HEALTH_CHANGE:
            processHealthChangePacket(data->getMessage());
            break;

        case PACKET_MANA_CHANGE:
            processManaChangePacket(data->getMessage());
            break;

        // больше случаев для других типов данных
    }
}
}

```

Функция `processNextPacket()` получает новый пакет и вызывает `dispatchPacket()` для обработки данных. В этом случае функция `dispatchPacket()` присутствует в стеке вызовов каждого обработчика, но отсутствует в стеке вызовов функции `recv()`. Посмотрите на гипотетические стеки в Таблице 10-2, например.

Таблица 10-2: Стеки псевдовызовов для трех функций, связанных с пакетами

<code>recv() stack</code>	<code>onHealthChange() stack</code>	<code>onManaChange() stack</code>
0x0BADFOOD	0x101E1337	0x14141414
0x40404040	0x00ABCDEF	0x00ABCDEF
0xDEADBEEF	0xDEADBEEF	0xDEADBEEF
0x30303030	0x30303030	0x30303030
0x20202020	0x20202020	0x20202020
0x10101010	0x10101010	0x10101010

Хотя у этих трёх функций первые четыре адреса в стеке вызовов одинаковые, только у двух обработчиков есть ещё один общий адрес (снова выделен **жирным**). Это 0x00ABCDEF, и это адрес функции `dispatchPacket()`. Снова можно представить их в виде древовидного отображения, как на **Рисунке 10-2**.

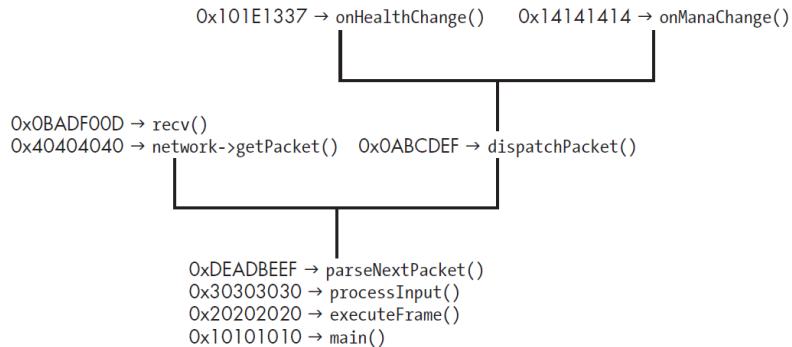


Figure 10-2: Древовидное представление наших трех стеков вызовов

## Взлом парсера (A Parser Hack)

Как только вы обнаружите функцию, отвечающую за передачу пакетов их обработчикам, вы сможете выявить каждый обработчик, который может быть вызван.

Вы можете определить назначение обработчика, установив на него точку останова и наблюдая, какие значения изменяются в памяти во время его выполнения. Затем можно перехватить любой обработчик, на который ваш бот должен реагировать. (Вернитесь к **Главе 8**, если вам нужно напоминание о том, как можно перехватывать эти функции.)

Конечно, существует бесконечное множество способов реализации сетевого поведения. Я не могу охватить их все, но понимание этих трёх распространённых методик поможет вам освоить сам подход. Независимо от того, с какой игрой вы работаете, установка точки останова на `recv()` должна стать шагом в правильном направлении.

## Выполнение игровых действий (Performing In-Game Actions)

Прежде чем бот сможет реагировать на события, ему нужно научиться играть в игру. Он должен уметь применять заклинания, передвигаться и активировать предметы. В этом отношении боты не сильно отличаются от людей: им просто нужно сказать, какие кнопки нажимать. Нажатие кнопок — простая задача и во многих случаях достаточная, но в более сложных ситуациях бот может потребовать связи по сети и передачи серверу информации о своих действиях.

Чтобы следовать примерам из этого раздела и затем изучить их самостоятельно, откройте файлы в папке *GameHackingExamples/Chapter10\_ResponsiveHacks/* в ресурсах этой книги.

## Эмуляция клавиатуры (Emulating the Keyboard)

Чаще всего в игре вам приходится нажимать клавиши клавиатуры, и есть несколько способов научить ваш бот набирать текст.

### Функция SendInput() (The SendInput() Function)

Один из распространённых способов эмуляции клавиатуры — использование функции **SendInput()** из Windows API. Эта функция отправляет ввод с клавиатуры и мыши в верхнее окно и имеет следующий прототип:

---

```
UINT SendInput(UINT inputCount, LPINPUT inputs, int size);
```

---

Первый параметр, **inputCount**, указывает количество отправляемых входных событий. В примерах этой книги я всегда буду использовать значение **1**. Второй параметр, **inputs**, — это указатель на структуру (или массив структур, длина которого соответствует значению **inputCount**) с предопределённым типом **INPUT**. Последний параметр — это размер входных данных в памяти, который вычисляется по формуле:

**size = inputCount × sizeof(INPUT).**

Структура **INPUT** сообщает функции **SendInput()**, какой тип ввода отправлять. В следующем коде показано, как можно инициализировать экземпляр **INPUT** для нажатия клавиши **F1**:

---

```
INPUT input = {0};  
input.type = INPUT_KEYBOARD;  
input.ki.wVk = VK_F1;
```

---

Чтобы ваш бот действительно нажал F1, вам нужно отправить этот ввод дважды, например:

```
SendInput(1, &input, sizeof(input));  
// изменить ввод на отпускание клавиши  
input.ki.dwFlags |= KEYEVENTF_KEYUP;  
SendInput(1, &input, sizeof(input));
```

Первый вызов **SendInput()** нажимает **F1**, а второй — отпускает её. Отпускание происходит не потому, что ввод был отправлен дважды, а потому, что второй вызов был сделан с установленным флагом **KEYEVENTF\_KEYUP** в поле флагов клавиатуры параметра ввода.

Поскольку настройка ввода даже для одной клавиши довольно громоздка, лучше всего обернуть весь процесс в функцию. Итоговый код выглядит примерно так, как показано в **Листинге 10-3**.

```

void sendKeyWithSendInput(WORD key, bool up)
{
    INPUT input = {0};
    input.type = INPUT_KEYBOARD;
    input.ki.wVk = key;
    input.ki.dwFlags = 0;

    if (up)
        input.ki.dwFlags |= KEYEVENTF_KEYUP;
    SendInput(1, &input, sizeof(input));
}

sendKeyWithSendInput(VK_F1, false); // нажать
sendKeyWithSendInput(VK_F1, true); // отпустить

```

Листинг 10-3: Обертка для эмуляции нажатия клавиши с помощью `SendInput()`

Эта функция инициализирует `input` с заданной `key`, включает флаг `KEYEVENTF_KEYUP`, если `up` установлен, и вызывает функцию `SendInput()`. Это означает, что `sendKeyWithSendInput()` должна быть вызвана второй раз для отправки отпускания клавиши, даже если оно всегда требуется.

Функция написана таким образом, потому что комбинации клавиш, включающие модификаторы, такие как **SHIFT**, **ALT** или **CTRL**, должны отправляться немного иначе: нажатие модификатора должно происходить перед нажатием основной клавиши, а его отпускание — после её отпускания.

Следующий код показывает, как использовать `sendKeyWithSendInput()`, чтобы сказать боту нажать **SHIFT+F1**:

```

sendKeyWithSendInput(VK_LSHIFT, false); // нажать shift
sendKeyWithSendInput(VK_F1, false); // нажать F1
sendKeyWithSendInput(VK_F1, true); // отпустить F1
sendKeyWithSendInput(VK_LSHIFT, true); // отпустить shift

```

Вам нужно вызвать `sendKeyWithSendInput()` четыре раза, но это всё равно проще, чем использовать код без обёрточной функции.

## Функция `SendMessage()`

Альтернативный метод отправки нажатий клавиш основан на функции `SendMessage()` Windows API. Эта функция позволяет отправлять ввод в любое окно, даже если оно свернуто или скрыто, путем отправки данных напрямую в очередь сообщений целевого окна. Это преимущество делает ее методом выбора для игровых хакеров, так как она позволяет пользователям заниматься другими делами, пока их бот играет в игру в фоновом режиме.

`SendMessage()` имеет следующий прототип:

---

```
LRESULT SendMessage(
    HWND window,
    UINT message,
    WPARAM wparam,
    LPARAM lparam);
```

---

Первый параметр, **window**, — это дескриптор окна, в которое отправляется ввод. Второй параметр, **message**, — это тип отправляемого ввода; для ввода с клавиатуры этот параметр принимает значения **WM\_KEYUP**, **WM\_KEYDOWN** или **WM\_CHAR**. Третий параметр, **wparam**, должен содержать код клавиши. Четвертый параметр, **lparam**, должен быть **0**, если сообщение — **WM\_KEYDOWN**, и **1** в противном случае.

Перед использованием функции **SendMessage()** необходимо получить дескриптор главного окна целевого процесса. Зная заголовок окна, вы можете получить этот дескриптор, используя функцию **FindWindow()** Windows API, следующим образом:

---

```
auto window = FindWindowA(NULL, "Title Of Game Window");
```

---

С действительным дескриптором окна (window handle) вызов функции **SendMessage()** (англ. *SendMessage()*) выглядит следующим образом:

---

```
SendMessageA(window, WM_KEYDOWN, VK_F1, 0);
SendMessageA(window, WM_KEYUP, VK_F1, 0);
```

---

Первый вызов нажимает клавишу F1, а второй вызов отпускает её. Однако помните, что эта серия вызовов работает только для клавиш, которые не вводят текст, таких как F1, INSERT или TAB. Чтобы ваш бот нажимал клавиши, вводящие текст, вам также нужно отправить сообщение **WM\_CHAR** между сообщениями нажатия и отпускания клавиши. Например, для ввода буквы W вы должны сделать следующее:

---

```
DWORD key = (DWORD)'W';
SendMessageA(window, WM_KEYDOWN, key, 0);
SendMessageA(window, WM_CHAR, key, 1);
SendMessageA(window, WM_KEYUP, key, 1);
```

---

Это создаёт переменную **key**, чтобы можно было легко изменить нажатую букву. Затем она выполняет те же шаги, что и в примере с F1, но с добавлением сообщения **WM\_CHAR** между ними.

**NOTE**

Фактически можно отправить только сообщение **WM\_CHAR** и получить тот же результат, но наилучшей практикой является отправка всех трёх сообщений. Разработчики игр могут легко заблокировать ботов, изменив код игры так, чтобы он игнорировал сообщения **WM\_CHAR**, которые не сопровождаются **WM\_KEYDOWN**, и даже могут использовать это как способ обнаружить вашего бота и забанить его.

Как я показал в технике `SendInput()`, вы можете создать обёртку вокруг этой функциональности, чтобы ваш код бота было проще использовать. Обёртка выглядит примерно так:

```
void sendKeyWithSendMessage(HWND window, WORD key, char letter)
{
    SendMessageA(window, WM_KEYDOWN, key, 0);
    if (letter != 0)
        SendMessageA(window, WM_CHAR, letter, 1);
    SendMessageA(window, WM_KEYUP, key, 1);
}
```

В отличие от Листинга 10-3, эта обёртка действительно отправляет как нажатие, так и отпускание клавиши. Это потому, что `SendMessage()` не может использоваться для отправки нажатий с модификаторами, поэтому между двумя вызовами никогда не нужно вставлять код.

**НОТ** Существует несколько способов, которыми игра может проверять, нажата ли клавиша-модификатор. Возможно, вы сможете отправить модификаторные клавиши в некоторые игры, вызывая функцию `SendMessage()`, но это зависит от того, как эти игры определяют модификаторы.

Вы можете использовать эту обёртку аналогично той, что в Листинге 10-3. Например, этот код отправляет F1, а затем W:

```
sendKeyWithSendMessage(window, VK_F1, 0);
sendKeyWithSendMessage(window, 'W', 'W');
```

Этот пример, как и весь код с `SendMessage()`, который я показывал до сих пор, просто выполняет задачу. Он может вводить текст, но не отправляет правильные сообщения.

Есть множество мелких деталей, которые необходимо учесть, если вы хотите отправлять на 100% корректные сообщения с помощью `SendMessage()`. Например, первые 16 битов lParam должны содержать количество раз, которое клавиша была автоматически повторена из-за удержания. Следующие 8 битов должны содержать scan code, идентификатор клавиши, уникальный для каждого производителя клавиатуры. Следующий бит, номер 24, должен быть установлен, если кнопка находится в расширенной части клавиатуры, например, на цифровом блоке. Следующие 4 бита не документированы, а следующий бит должен быть установлен, только если ALT был нажат во время исходного сообщения. Последние 2 бита — это флаг предыдущего состояния и флаг переходного состояния. Флаг предыдущего состояния устанавливается, если клавиша была нажата ранее, а флаг переходного состояния устанавливается, только если клавиша ранее находилась в состоянии, противоположном текущему (то есть, если сейчас клавиша отпущена, а раньше была нажата, или наоборот).

К счастью, средняя игра не учитывает большинство этих значений. Более того, средний программный продукт тоже о них не заботится. Если вам нужно заполнить все эти значения правильными данными, чтобы заставить ваш бот работать, значит, вы движетесь не в том направлении. Существует множество других способов выполнения действий, большинство из которых проще, чем попытка эмулировать точное поведение обработчика/диспетчера ввода с клавиатуры на уровне ядра операционной системы. Фактически, уже есть функция, которая делает это, и я уже говорил о ней: функция `SendInput()`.

Вы также можете управлять мышью с помощью функций `SendInput()` и `SendMessage()`, но я настоятельно рекомендую избегать этого. Любые команды, которые вы отправляете, будут влиять на любые законные движения мыши, клики или нажатия клавиш, совершаемые игроком, и наоборот. То же самое относится и к вводу с клавиатуры, но такие сложности встречаются гораздо реже.

## Отправка пакетов (Sending Packets)

Прежде чем игра отрисует кадр, она проверяет ввод с клавиатуры и мыши. Когда она получает ввод, который приводит к действию, например, движению персонажа или применению заклинания, она проверяет, возможно ли выполнить это действие, и, если да, сообщает серверу игры, что действие было выполнено. Код игры, отвечающий за проверку событий и уведомление сервера, часто выглядит следующим образом:

```

void processInput() {
    do {
        auto input = getNextInput();
        if (input.isKeyboard())
            processKeyboardInput(input);
        // обрабатывает другие типы ввода (например, мышь)
    } while (!input.isEmpty());
}

void processKeyboardInput(input) {
    if (input.isKeyPress()) {
        if (input.getKey() == 'W')
            step(FORWARD);
        else if (input.getKey() == 'A')
            step(BACKWARD);
        // обрабатывает другие нажатия клавиш (например, 'S' и 'D')
    }
}

void step(int direction) {
    if (!map->canWalkOn(player->position))
        return;
    playerMovePacket packet(direction);
    network->send(packet);
}

```

Функция `processInput()` вызывается каждый кадр. Эта функция перебирает все ожидающие ввода и отправляет разные типы ввода их соответствующим обработчикам. В этом случае, когда получен ввод с клавиатуры, он передается в функцию `processKeyboardInput()`. Этот обработчик затем проверяет, является ли нажатая клавиша `W` или `S`, и, если да, вызывает `step()` для перемещения игрока в соответствующем направлении.

Так как `step()` используется для выполнения действия, она называется **actor-функцией**. Вызов actor-функции называется **actuation**. Вы можете напрямую вызывать actor-функции игры из своего бота, чтобы выполнить действие, полностью обойдя уровень обработки ввода.

Прежде чем вы сможете вызвать actor, вам нужно найти его адрес. Чтобы сделать это, вы можете прикрепить **OllyDbg** к игре, открыть командную строку и ввести `bp send`. Это создаст точку останова на функции `send()`, которая используется для отправки данных по сети. Когда вы играете в игру, каждый раз, когда вы делаете шаг, произносите заклинание, подбираете предмет или выполняете любое другое действие, ваша точка останова должна срабатывать, и вы можете отметить каждую функцию в **стеке вызовов**.

**НОТЕ**

Игра должна вызывать `send()` каждый раз, когда вы что-либо делаете во время игры. Обратите внимание на то, что вы делали перед каждым срабатыванием точки останова `send()`, так как это даст вам приблизительное представление о том, какое действие каждый вызов передает серверу, и, в конечном итоге, за что отвечает найденный вами `actor`.

Как только у вас появится несколько различных стеков вызовов, вы сможете сравнить их, чтобы найти `actor`-функции. Чтобы понять, как определить `actor`-функции, давайте сравним два аннотированных стека вызовов на **рисунке 10-3**.

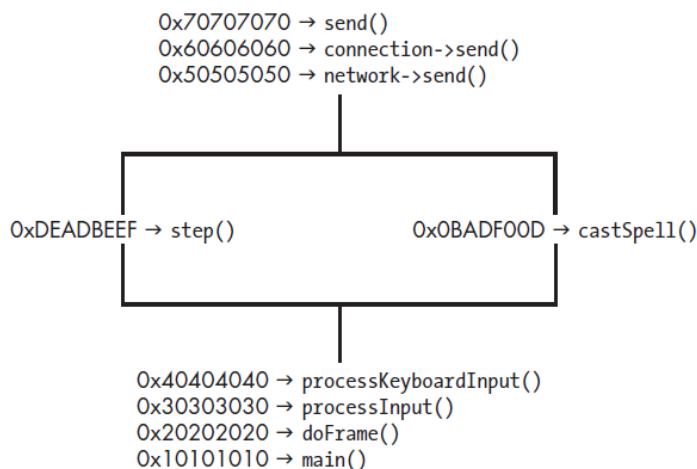


Figure 10-3: Древовидное представление стеков вызовов двух функций-акторов

Как и в случае с этими двумя стеками, найденные вами стеки вызовов должны быть идентичны в верхней части, разделяя несколько общих функций, отвечающих за стандартную сетевую передачу. Они также должны быть идентичны в нижней части, так как каждый вызов `send()` должен был исходить из функции `processInput()`. Однако каждый стек должен содержать несколько уникальных функций между этими идентичными областями, и именно эти функции являются `actor`-функциями, которые вы ищете. Обычно интересующая вас функция находится сразу под общими сетевыми вызовами. В данном случае двумя `actor`-функциями являются `step()` и `castSpell()`.

После работы с одной и той же игрой в течение некоторого времени вы научитесь определять, насколько высоко в стеке находятся `actor`-функции относительно вызова `send()`. Например, на **рисунке 10-3** видно, что `actor`-функции вызываются за три вызова до `send()`. Зная это, вы можете просто подняться вверх по стеку в **OllyDbg** (CTRL+F9, затем F8) три раза в момент срабатывания точки останова на `send()` и оказаться внутри `actor`-функции, которая отправила данные.

Как только вы нашли actor-функцию, вы можете вызвать ее из загружаемой DLL. Вот как вы можете вызвать `step()`, если обнаружили его по адресу 0xDEADBEEF:

```
typedef void _step(int direction);
auto stepActor = (_step*)0xDEADBEEF;

stepActor(FORWARD);
```

Поскольку бот не знает фактического имени этой игровой функции, код присваивает содержимое памяти по адресу 0xDEADBEEF удобной переменной с именем `stepActor`. Затем код просто вызывает `stepActor()` как любую другую функцию.

Если у вас есть правильный адрес, прототип функции и параметры, это должно работать идеально: вы сможете автоматизировать действия, как если бы у вас был доступ к исходному коду игры. Единственное, что важно — вызывать actor-функции из того же потока, что и игра, иначе могут возникнуть проблемы с многопоточностью. Лучший способ сделать это — вызывать actor-функции из хука на крупной функции, такой как `EndScene()` в Direct3D или `PeekMessage()` в Windows API, так как эти функции, как правило, вызываются только из основного потока игры.

#### Использование `this` для вызова `_thiscall`

Если вы попытаетесь вызвать actor-функцию, которая является нестатическим членом класса, она будет использовать соглашение о вызовах `_thiscall`, что означает, что вам нужно передать экземпляр класса через регистр ECX. (Вы можете освежить в памяти соглашения о вызовах в разделе "Function Calls" на странице 94.) Передача экземпляра — это простая задача, но сначала вам нужно найти цепочку указателей к самому экземпляру класса.

Чтобы найти цепочку указателей, можно установить точку останова на actor-функцию, захватить значение экземпляра класса из регистра ECX в момент срабатывания точки останова, а затем вставить это значение в Cheat Engine для поиска указателей. Далее, чтобы вызвать функцию, вам нужно будет пройти по цепочке указателей, получить текущий адрес экземпляра и использовать inline-ассемблер для установки ECX перед самим вызовом функции. Этот процесс работает аналогично тому, как VF-хуки вызывают свои оригинальные функции, как описано в разделе "Writing a VF Table Hook" на странице 156.

## Связываем все воедино

После того как вы создали фреймворки для наблюдения за событиями и выполнения действий, вы можете объединить их, чтобы создать **реактивные хаки**. Существует множество видов таких хаков, но есть несколько наиболее распространенных.

## Создание идеального целителя

Один из самых популярных ботов среди геймеров — **автолечение (autohealing)**, хак, который автоматически использует **лечащее заклинание**, когда здоровье игрока резко падает или опускается ниже определенного порога. Если у вас есть способ обнаруживать изменения здоровья и функция-актор для применения заклинаний, **автолечилка** может выглядеть следующим образом:

```
void onHealthDecrease(int health, int delta) {  
    if (health <= 500)           // здоровье ниже 500  
        castHealing();  
    else if (delta >= 400)      // резкое снижение здоровья  
        castHealing();  
}
```

Эта функция **автолечения** довольно проста, но работает хорошо. Более продвинутые **автолечилки** могут иметь **разные уровни лечения** и обучаться в процессе. На странице [222](#) в разделе “Контрольная теория и игровое хакерство” (*Control Theory and Game Hacking*) вы найдете рабочий код и **глубокий разбор** продвинутых автолечилок.

## Сопротивление атакам контроля толпы (Crowd-Control)

Анти-контроль-толпы (Anti-crowd-control) хаки обнаруживают атаки контроля толпы (crowd-control attacks) и автоматически применяют заклинания, которые уменьшают их эффект или полностью их нейтрализуют. Атаки контроля толпы так или иначе ограничивают действия игроков, поэтому, если враги накладывают их на вас, это может принести серьезные неудобства.

Имея возможность обнаруживать входящие или активные эффекты контроля толпы (например, с помощью детектирования моделей Direct3D или перехвата входящего сетевого пакета), а также функцию-актор для применения заклинаний, бот может мгновенно реагировать на такие атаки следующим образом:

```

void onIncomingCrowdControl() {
    // накладывает щит, блокирующий контроль толпы
    castSpellShield();
}

void onReceiveCrowdControl() {
    // снимает уже наложенный эффект контроля
    castCleanse();
}

```

Функция **onIncomingCrowdControl()** может **заблокировать** заклинание контроля **до** того, как оно вас настигнет. Если же это не удастся, бот может вызывать **onReceiveCrowdControl()**, чтобы **снять** уже наложенные эффекты.

## Избегание напрасной траты маны (Avoiding Wasted Mana)

Тренировщики заклинаний (Spell trainers) также довольно распространены среди ботов. Такие тренировщики ожидают, пока у игрока будет полная мана, а затем накладывают заклинания, чтобы увеличить уровень магии или характеристики персонажа. Это позволяет игрокам быстро прокачивать магические навыки, не теряя регенерацию маны просто из-за того, что её запас уже полон.

Имея возможность обнаруживать изменения маны и функцию-актор для наложения заклинаний, бот может включать следующий псевдокод для тренировщика заклинаний:

```

void onManaIncrease(int mana, int delta) {
    if (delta >= 100) // игрок использует зелья маны
        return; // если ему нужна мана, ничего не делать

    if (mana >= MAX_MANA - 10) // мана почти полная, используем заклинание
        castManaWasteSpell();
}

```

Эта функция принимает в качестве параметров количество маны игрока и увеличение маны (delta). Если увеличение маны превышает определённое значение, функция предполагает, что игрок использует зелья или другие предметы для её восполнения, и не тратит дополнительные заклинания. В противном случае, если у игрока много маны, функция применяет любое старое заклинание, чтобы получить дополнительные очки опыта.

Другие распространённые отзывчивые хаки:

**autoreload** – моментальная перезарядка боеприпасов,

**autododge** – автоматическое уклонение от летящих снарядов,

**autocombo** – автоматическая атака той же цели, что и

ближайший союзник.

Фактически, единственное ограничение для отзывчивых хаков – это количество игровых событий, которые бот может наблюдать, умноженное на число возможных ответов, которые он может отправить на каждое из них.

## **Заключительные мысли (Closing Thoughts)**

Используя хаки, манипуляции с памятью и эмуляцию клавиатуры, вы можете начать создавать свои первые отзывчивые хаки. Эти хаки станут вашей отправной точкой к автоматизации игрового процесса, но это лишь малая часть того, что возможно. Глава 11 станет кульминацией вашего путешествия в геймхакинге. Используя всё, что вы изучили до этого момента и опираясь на принципы отзывчивых хаков, вы научитесь автоматизировать сложные действия и создавать по-настоящему автономного бота.

Если вы пока не чувствуете себя достаточно уверенно, чтобы двигаться дальше, я настоятельно рекомендую пересмотреть пройденный материал и попрактиковаться в изолированной среде на своём компьютере. Реализация подобных ботов гораздо проще, чем вы можете думать, и это удивительно увлекательный процесс. Как только вы освоите создание автохилеров и других базовых отзывчивых хаков, вы будете готовы полностью автоматизировать игровой процесс.

# 11) Собираем всё воедино: написание автономных ботов



Конечная цель взлома игр — создание полностью автономного бота, способного играть в игру часами напролёт.

Такие боты могут лечиться, пить зелья, убивать монстров, собирать лут, перемещаться, продавать предметы, покупать припасы и многое другое. Создание таких мощных ботов требует объединения ваших хуков и чтения памяти с концепциями, такими как теория управления, конечные автоматы и алгоритмы поиска, которые рассмотрены в этой главе.

На протяжении этих уроков вы также узнаете о распространённых автоматизированных хаках и о том, как они должны вести себя на высоком уровне. После рассмотрения теории и кода, лежащих в основе автоматизированных хакингов, я предложу вам общий обзор двух типов ботов, которые используют такой код:

cavebots, которые могут исследовать пещеры и приносить домой добычу, и warbots, которые могут сражаться с врагами вместо вас.

К концу главы вы должны быть готовы достать свои инструменты, запустить среду разработки и приступить к созданию действительно крутых ботов.

## Теория управления и взлом игр

Теория управления — это раздел инженерии, который предоставляет способ управления поведением динамических систем. Теория управления определяет состояние *системы* с помощью *сенсоров*, после чего *контроллер* определяет набор действий, необходимых для приведения текущего состояния системы к другому, желаемому состоянию. После выполнения контроллером первого действия в наборе весь процесс — известный как *обратная связь* — повторяется (см. рисунок 11-1).

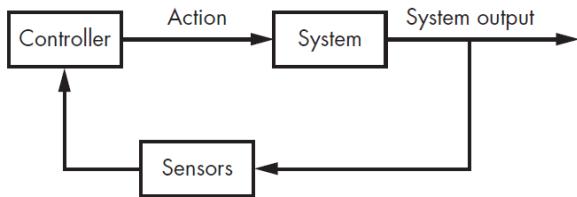


Рисунок 11-1: Контур обратной связи по теории управления

Давайте применим этот цикл обратной связи к взлому игр. Чтобы автоматизировать игровой процесс (систему), бот реализует некоторые алгоритмы (контроллер), которые понимают, как играть в игру в любом состоянии, наблюдаемом с помощью чтения памяти, сетевых хуков и так далее (сенсоров). Контроллер обычно принимает некоторые входные данные от человека, такие как маршрут движения, цели для атаки и добыча, которую нужно подобрать. Таким образом, для достижения желаемого состояния контроллер должен выполнить некоторый набор этих входных данных, возможных в текущем состоянии.

Например, если на экране нет существ и нет трупов для сбора добычи, желаемым состоянием может быть перемещение игрока к следующей точке маршрута (называемой *waypoint*) по заранее определенному пути. В этом случае контроллер передвигает игрока на один шаг ближе к точке маршрута на каждой итерации. Если игрок встречает существа, контроллер может принять решение атаковать его в первом кадре, а в следующих кадрах чередовать бегство от существа (известное как *kiting*) и применение заклинаний. После смерти существа контроллер выполняет набор действий для сбора добычи и продолжает движение к следующей точке маршрута.

Учитывая этот пример работы цикла обратной связи, может показаться, что кодирование такой системы слишком сложно. К счастью, существуют некоторые шаблоны проектирования, которые делают задачу намного проще, чем кажется.

## Конечные автоматы

Конечные автоматы – это математические модели вычислений, которые описывают, как система ведет себя в зависимости от входных данных. На рисунке 11-2 показан простой конечный автомат, который считывает последовательность двоичных цифр. Автомат начинается с начального состояния  $S_1$ . По мере итерации по цифрам во входных данных он изменяет свое состояние соответствующим образом.

В этом случае состояния  $S_1$  и  $S_2$  чередуются: при встрече единицы автомат переключается в другое состояние, а при встрече нуля остается в том же состоянии. Например, для двоичной

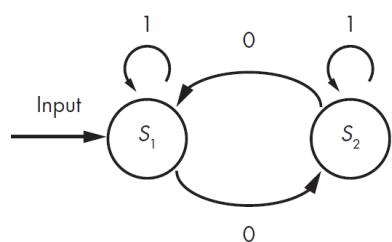


Figure 11-2: A simple state machine

последовательности **11000111** переходы состояний будут следующими:  $S_1, S_1, S_2, S_1, S_1, S_2, S_2, S_2$  и, наконец,  $S_2$ .

Небольшая адаптация классической теории конечных

## КОНЕЧНЫЕ АВТОМАТЫ И ВЗЛОМ ИГР

Конечный автомат, используемый для взлома игр, должен не только сохранять внутреннее состояние, но и реагировать (или активировать) на игровую среду в соответствии с этим состоянием. Общий игровой процесс может изменяться из-за действий бота, поведения других игроков и других непредсказуемых событий в игровом окружении.

По этой причине попытки создать конечный автомат, который бы шаг за шагом следовал за наблюдаемой игровой средой, бесполезны. Почти невозможно заранее задать переходы для каждого состояния, учитывая все возможные наблюдения, которые могут произойти между итерациями. Гораздо логичнее, если конечный автомат будет переоценивать игровую среду как новый контекст при каждом поступлении входных данных.

Чтобы реализовать этот подход, конечный автомат должен использовать саму игровую среду в качестве механизма для перехода между состояниями. Иными словами, влияние машины на окружение должно быть достаточным для того, чтобы в следующих итерациях это активировало новое состояние. Классические конечные автоматы могут работать подобным образом, но мы упростим их структуру и будем использовать их в более простом, но всё же весьма мощном формате.

Если вы знакомы с классическими конечными автоматами, этот подход может показаться вам неочевидным. Однако в следующих разделах вы увидите, как конечные автоматы можно модифицировать и комбинировать с теорией управления для достижения нужного результата.

Основное различие заключается в том, что вместо простого переключения состояний каждое состояние в автомате управления игрой выполняет игровые действия, изменяющие общее состояние игры. Таким образом, состояние, обнаруженное на следующей итерации цикла обратной связи, зависит от этих действий. В коде объект, представляющий состояние в этом автомате, может выглядеть так:

---

```
class StateDefinition {
public:
    StateDefinition(){}
    ~StateDefinition(){}
    bool condition();
    void reach();
};
```

---

Вы можете собрать объекты `StateDefinition` в автомат состояний с помощью простого определения `std::vector`, например:

```
std::vector<StateDefinition> stateMachine;
```

---

И вот, у вас есть скелет автомата состояний, готовый к приёму любых объектов `StateDefinition`, которые вы создадите. В сочетании с циклом обратной связи этот автомат состояний можно использовать для определения последовательности автоматизации.

Во-первых, вы можете создать список определений, моделирующих желаемое поведение бота, упорядоченных векторами по важности. Каждый объект `StateDefinition` может использовать данные с ваших сенсоров в качестве входных данных, передавая их в функцию `condition()`, чтобы определить, должно ли состояние активироваться. Затем вы можете создать контроллер, который проходит по списку состояний, вызывая функцию `reach()` первого состояния, для которого `condition()` возвращает `false`. Наконец, вы можете обернуть контроллер в цикл обратной связи. Если пока вам неясно, как работает этот цикл обратной связи, не беспокойтесь — я сейчас покажу, как это закодировать.

**НОТ**

*Вы можете рассматривать оператор в вашей функции `condition()` как требование для автомата перейти в следующее состояние. Если оператор истинный, это означает, что перед оценкой следующего состояния в списке не должно происходить никаких исполнительных действий, и цикл может продолжать итерацию. Если оператор ложный, это означает, что перед переходом должно произойти какое-то исполнительное действие.*

Вы найдёте весь пример кода для следующего раздела и «Исправления ошибок» на странице 230 в каталоге `GameHackingExamples/Chapter11_StateMachines` среди файлов исходного кода этой книги. Включённые проекты можно скомпилировать в Visual Studio 2010, но они также должны работать с любым другим компилятором C++. Скачайте их по адресу <https://www.nostarch.com/gamehacking/> и скомпилируйте, если хотите следовать материалу.

## Объединение теории управления и конечных автоматов

Чтобы связать состояния с помощью обратной связи, сначала необходимо предоставить каждому объекту `StateDefinition` универсальный способ доступа к датчикам и исполнительным механизмам, которые вы реализовали. Тогда класс `StateDefinition` будет выглядеть следующим образом:

---

```
class StateDefinition {
public:
    StateDefinition(){}
    ~StateDefinition(){}
    bool condition(GameSensors* sensors);
    void reach(GameSensors* sensors, GameActuators* actuators);
};
```

---

Это изменение просто модифицирует функции `condition()` и `reach()`, чтобы они принимали экземпляры классов `GameSensors` и `GameActuators` в качестве аргументов.

`GameSensors` и `GameActuators` — это классы, которые вам нужно определить; `GameSensors` будет содержать результаты чтения памяти, сетевых перехватов и других источников данных, которые ваш бот перехватывает из игры, а `GameActuators` будет представлять собой набор функций акторов, способных выполнять действия внутри игры.

Далее вам потребуется универсальный способ определения каждого отдельного состояния. Вы можете абстрагировать определение каждого состояния в отдельный класс, который наследует `StateDefinition` и реализует `condition()` и `reach()` как виртуальные функции.

Альтернативно, если исходный код должен занимать мало места (например, в книге, *подмигивание-подмигивание*), можно оставить один класс для представления каждого определения и использовать `std::function` для реализации функций `condition()` и `reach()` вне определения класса.

Следуя этому альтернативному методу, окончательная версия `StateDefinition` будет выглядеть так:

---

```
class StateDefinition {
public:
    StateDefinition(){}
    ~StateDefinition(){}
    std::function<bool(GameSensors*)> condition;
    std::function<void(GameSensors*, GameActuators*)> reach;
};
```

---

## Простой конечный автомат целителя

Следующим шагом является определение фактического поведения бота. Чтобы сохранить пример кода простым, предположим, что вы реализуете автоматического целителя.

Этот целитель имеет два метода исцеления: он использует сильное исцеление, если у игрока 50 процентов здоровья или

меньше, и слабое исцеление, если у игрока от 51 до 70 процентов здоровья.

Конечный автомат, представляющий это поведение, требует двух состояний: одно для сильного исцеления, другое для слабого исцеления. Для начала нужно определить конечный автомат как вектор с двумя объектами StateDefinition:

---

```
std::vector<StateDefinition> stateMachine(2);
```

---

Этот код создает конечный автомат, называемый stateMachine, и инициализирует его двумя пустыми объектами StateDefinition. Затем вы определяете функции condition() и reach() для этих состояний.

Состояние сильного исцеления самое важное, так как оно предотвращает гибель персонажа, поэтому оно должно быть первым в векторе, как показано в Листинге 11-1.

---

```
auto curDef = stateMachine.begin();
curDef->condition = [](GameSensors* sensors) {
❶    return sensors->getHealthPercent() > 50;
};
curDef->reach = [](GameSensors* sensors, GameActuators* actuators) {
❷    actuators->strongHeal();
};
```

---

Листинг 11-1: Код для сильного исцеляющего состояния

Этот код сначала создает итератор curDef, который указывает на первый объект StateDefinition в векторе stateMachine. Затем определяется функция condition() ❶, которая говорит: "Состояние выполняется, если процент здоровья игрока больше 50."

Если состояние не выполняется, то функция reach() объекта вызывает strongHeal() ❷, чтобы применить сильное исцеление.

Теперь, когда состояние сильного исцеления определено, можно задать слабое состояние исцеления, как показано в Листинге 11-2.

---

```
curDef++;
curDef->condition = [](GameSensors* sensors) {
❶    return sensors->getHealthPercent() > 70;
};
curDef->reach = [](GameSensors* sensors, GameActuators* actuators) {
❷    actuators->weakHeal();
};
```

---

Листинг 11-2: Код для слабого исцеления

После увеличения curDef, чтобы он указывал на второй объект StateDefinition в векторе stateMachine, этот код определяет функцию condition() объекта.

Функция ❶ определяется как: "Состояние выполняется, если

процент здоровья игрока больше 70."

Также определяется функция `reach()`, которая вызывает `actuators->weakHeal()` ②.

Когда вы завершите определение конечного автомата, необходимо реализовать контроллер. Так как поведение контроллера уже заложено в конечном автомате, нужно лишь добавить простой цикл, чтобы завершить его.

---

```
for (auto state = stateMachine.begin(); state != stateMachine.end(); state++) {  
    if (①!state->condition(&sensors)) {  
        state->reach(&sensors, &actuators);  
        break;  
    }  
}
```

---

Этот цикл контроллера проходит по конечному автоматау, выполняет функцию `reach()` первого состояния, для которого функция `condition()` возвращает `false` ①, и прерывается, если была вызвана какая-либо `reach()` функция.

Финальный шаг — реализовать петлю обратной связи (`feedback loop`) и поместить внутрь нее этот цикл контроллера, как показано в Листинге 11-3.

---

```
while (true) {  
    for (auto state = stateMachine.begin();  
         state != stateMachine.end();  
         state++) {  
        if (!state->condition(&sensors)) {  
            state->reach(&sensors, &actuators);  
            break;  
        }  
        Sleep(FEEDBACK_LOOP_TIMEOUT);  
    }  
}
```

---

*Листинг 11-3: Машина состояния окончательного выздоровления и контур обратной связи*

Этот цикл непрерывно выполняет цикл контроллера и приостанавливает выполнение на `FEEDBACK_LOOP_TIMEOUT` миллисекунд между каждой итерацией.

Вызов `Sleep()` позволяет игровому серверу получать и обрабатывать любые активации (`actuation`) от предыдущей итерации, а также дает игровому клиенту возможность получить результаты активации от сервера перед выполнением следующего цикла контроллера.

Если вы все еще немного запутались, посмотрите Рисунок 11-3, который демонстрирует работу бесконечного цикла в Листинге 11-3.

Сначала проверяется, выполнено ли условие сильного исцеления. Если оно истинно, тогда проверяется условие слабого

исцеления. Если условие сильного исцеления ложно, это означает, что уровень здоровья игрока равен или ниже 50 процентов, поэтому вызывается метод сильного исцеления. Если условие слабого исцеления также ложно, это означает, что уровень здоровья игрока находится в диапазоне от 51 до 70 процентов, поэтому вызывается метод слабого исцеления.

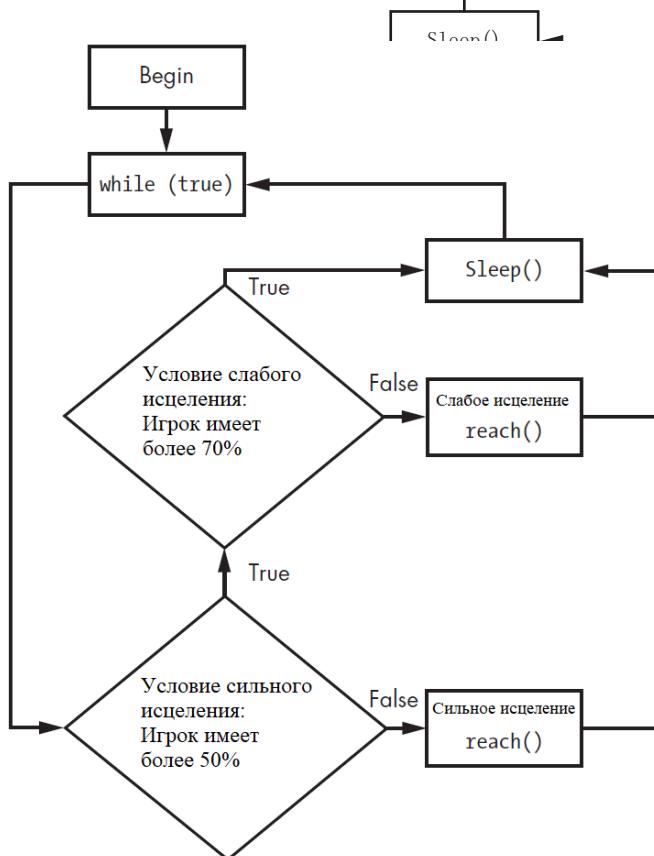


Рисунок 11-3: Блок-схема машины состояния исцеления и контура обратной связи

## Гипотетическая сложная конечная машина

Поведение, реализованное в машине состояний для лечения, является простым, поэтому включение его в такую структуру управления может показаться излишним, но это полезно, если вы хотите расширить контроллер. Например, если вы хотите объединить машину состояний для лечения с поведением "ходьба, атака, сбор добычи", которое я обсуждал в разделе "Теория управления и взлом игр" на странице 222, структура управления станет гораздо сложнее. Давайте рассмотрим основные состояния, которые вам понадобятся:

**Сильное лечение** Условие выполняется, если здоровье выше 50 процентов. Переход осуществляется путем применения заклинания сильного лечения.

**Слабое лечение** Условие выполняется, если здоровье выше 70 процентов. Переход осуществляется путем применения заклинания слабого лечения.

**Атакующее заклинание** Условие выполняется, если нет доступной цели или атакующее заклинание находится на перезарядке. Переход осуществляется путем применения атакующего заклинания на цель.

**Кайт монстра** Условие выполняется, если нет доступной цели или если дистанция до цели достаточна. (Определение "достаточно" зависит от того, насколько далеко вы хотите держаться от врагов при кайтинге). Переход осуществляется путем отступления от цели.

**Цель - монстр** Условие выполняется, если нет существ для атаки. Переход осуществляется путем атаки существа.

**Сбор добычи** Условие выполняется, если нет доступного трупа или если открытый труп не содержит добычи. Переход осуществляется путем сбора предмета из открытого трупа.

**Подход к трупу** Условие выполняется, если нет доступных трупов или если персонаж находится рядом с трупом, который можно будет открыть. Переход осуществляется путем движения к такому трупу.

**Открытие трупа** Условие выполняется, если персонаж не находится рядом с трупом, который можно открыть. Переход осуществляется путем открытия доступного трупа.

**Следовать за путевой точкой** Условие выполняется, если персонаж не может двигаться к текущей путевой точке или если он стоит на текущей путевой точке. Переход осуществляется путем движения на шаг в сторону текущей путевой точки.

**Продвижение по путевой точке** Условие выполняется, если не осталось путевых точек для следования. Переход осуществляется путем обновления текущей путевой точки до следующей в списке. Если персонаж не может достичь текущей путевой точки по какой-либо причине (например, застрял), состояние "Продвижение по путевой точке" предотвращает его застревание. Если персонаж достиг текущей путевой точки, "Продвижение по путевой точке" выбирает следующую точку для продолжения движения.

Эта конечная машина значительно сложнее, чем машина состояний, предназначенная только для лечения. Если бы я составил диаграмму этой конечной машины, в диаграмме было бы 23 объекта, а стрелки указывали бы на 33 управляющих пути. Сравните это с Рисунком 11-3, в котором всего 7 объектов и 9 управляющих путей.

Вы могли бы запрограммировать поведение лечения без использования машины состояний или цикла обратной связи, но я не могу представить, как легко сделать это для полноценного бота. Каждое из этих 10 состояний зависит не только от своего собственного условия, но и от условия каждого предыдущего состояния. Более того, жесткое кодирование логики потребовало бы либо множества вложенных операторов `if()`, либо кучи вложенных операторов `if() / return()` – и в итоге конечный результат выглядел бы как машина состояний, но без гибкости в

реальном времени.

*Гибкость в реальном времени* означает способность машины состояний к изменению. В отличие от жестко заданных условий, определения состояний в машине состояний можно удалять, заменять и добавлять динамически. Метод машины состояний позволяет вам подключать и использовать различные поведения и функции в зависимости от пользовательского ввода.

Чтобы продвинуть этот концепт еще дальше, вы могли бы предоставить доступ к сенсорам и исполнительным механизмам (актуаторам) через среду Lua, создать Lua-функции, способные добавлять и удалять состояния из машины состояний, а также модифицировать `StateDefinition` так, чтобы его функции `condition()` и `reach()` могли вызывать Lua-функции, доступные в среде Lua. Такой подход к написанию системы управления позволил бы вам кодировать ядро вашего бота (хуки, чтение памяти, выполнение действий) на C++, а саму систему управления на Lua – высокогоревом динамическом языке – что облегчило бы автоматизацию.

**НОТЕ**

Вы можете встроить Lua в свои программы, добавив несколько заголовочных файлов и связавшись с библиотекой Lua. Этот процесс несложный, но выходит за рамки этой книги, поэтому я рекомендую вам ознакомиться с главой 24 "Программирование на Lua" Роберто Иерусалимского [<http://www.lua.org/pil/24.html>] для дополнительной информации.

## Исправление ошибок

Еще один элемент теории управления, полезный для игрового хакинга, — это *исправление ошибок*. Механизм исправления ошибок в контроллере отслеживает результат выполнения действий, сравнивает его с ожидаемым результатом и корректирует будущие вычисления, чтобы привести последующие результаты к ожидаемому значению. Исправление ошибок может быть полезным при работе со *стохастическими системами*, где выходные данные, полученные из определенного входного значения, не всегда предсказуемы.

В целом игры являются стохастическими, но, к счастью для игровых хакеров, результаты действий в основном детерминированы. Возьмем, к примеру, контроллер лечения. В большинстве игр можно точно рассчитать, сколько здоровья восстанавливает заклинание, а значит, можно точно знать, когда его использовать. Но представьте, что вы создаете бота-лекаря для узкого спектра ситуаций, когда исцеление невозможно точно рассчитать; например, бот должен работать с разными персонажами на разных уровнях без пользовательского ввода.

Исправление ошибок может помочь вашему боту научиться тому, как лучше лечить игроков. В этом случае есть два способа реализовать механизм исправления ошибок, каждый из которых зависит от того, как работает система лечения.

## Корректировка для постоянного коэффициента

Если лечение выполняется с постоянным коэффициентом восстановления здоровья, вам нужно будет просто скорректировать контроллер после первого лечения. Предполагая, что ваши сенсоры могут определить, сколько здоровья было восстановлено, это добавляет всего несколько строк кода. Вы легко могли бы изменить состояние слабого лечения в листинге 11-2, сделав его таким:

---

```
curDef->condition = [](GameSensors* sensors) -> bool {
    static float healAt = 70;
    static bool hasLearned = false;
    if (!hasLearned && sensors->detectedWeakHeal()) {
        hasLearned = true;
        healAt = 100 - sensors->getWeakHealIncrease();
    }
    return sensors->getHealthPercent() > healAt;
};
```

---

Вместо жесткого кодирования значения 70 в качестве порога для слабого лечения, этот код перемещает порог в статическую переменную под названием `healAt`. Он также добавляет другую статическую переменную под названием `hasLearned`, чтобы код знал, когда обучение завершено.

При каждом вызове функции `condition()`, код проверяет два условия: является ли `hasLearned` ложным (`false`) и зафиксировали ли сенсоры событие слабого лечения. Когда эта проверка проходит, код устанавливает `hasLearned` в `true` и обновляет `healAt`, чтобы лечение происходило при идеальном или более низком проценте здоровья. Например, если слабое лечение увеличивало здоровье на 20 процентов, `healAt` будет установлен на 80 процентов вместо 70, так что каждое лечение будет восстанавливать игрока до 100 процентов здоровья.

## Реализация адаптируемой коррекции ошибок

Но что, если ваша способность к исцелению увеличивается? Если персонаж может повышать уровни, распределять очки навыков или увеличивать максимальное здоровье, количество восстанавливаемого здоровья может изменяться соответственно. Например, если вы запускаете бота на персонаже 10-го уровня и оставляете его работать до 40-го уровня, ваш код исцеления должен адаптироваться. Исцеление персонажа 40-го уровня так же, как и на 10-м уровне, привело бы либо к чрезмерному исцелению, либо к быстрой смерти от противников соответствующего уровня.

Чтобы справиться с этим сценарием, бот должен постоянно обновлять свой порог исцеления, чтобы он соответствовал наблюдаемому количеству исцеления. В Листинге 11-4 показано, как можно изменить функцию условия для сильного исцеления из Листинга 11-1, чтобы сделать это.

---

```
curDef->condition = [](GameSensors* sensors) -> bool {
    static float healAt = 50;
①    if (sensors->detectedStrongHeal()) {
        auto newHealAt = 100 - sensors->getStrongHealIncrease();
②        healAt = (healAt + newHealAt) / 2.00f;
③        sensors->clearStrongHealInfo();
    }
    return sensors->getHealthPercent() > healAt;
};
```

---

Листинг 11-4: Настройка кода состояния сильного исцеления

Как и в модифицированной функции слабого исцеления, порог исцеления был перенесён в статическую переменную, называемую `healAt`, но на этот раз логика немного отличается. Поскольку обучение должно происходить непрерывно, здесь нет переменной для отслеживания того, усвоил ли бот уже свою истинную способность к исцелению. Вместо этого код просто проверяет, видели ли сенсоры событие сильного исцеления с момента последнего вызова ①. Если да, код заменяет `healAt` на среднее значение между `healAt` и `newHealAt` и вызывает функцию для очистки сенсоров от информации, связанной с сильным исцелением ③.

Очистка сенсоров на самом деле очень важна, так как это не даёт коду постоянно обновлять `healAt` на основе обратной связи от одного и того же применения заклинания сильного исцеления. Также обратите внимание, что эта функция не обновляет `healAt` до точного значения, а вместо этого постепенно сдвигает его к наблюдаемому оптимальному значению. Такое поведение делает новую функцию идеальной для ситуаций, в которых присутствует некоторая степень случайности в том, насколько эффективно может происходить исцеление. Если вашему боту нужно быстрее сдвигаться к новому значению, вы можете изменить строку на ② следующим образом:

---

```
healAt = (healAt + newHealAt * 2) / 3.00f;
```

---

Этот код для обновления `healAt` использует среднее значение, взвешенное в сторону `newHealAt`. Однако есть несколько моментов, которые следует учитывать при использовании этого подхода.

Во-первых, что происходит при избыточном исцелении? В некоторых играх, когда вы восстанавливаете полное здоровье, ваши сенсоры могут определять только фактическое количество восстановленного здоровья. В других играх сенсоры могут фиксировать общее исцеление, независимо от того, сколько из него было избыточным.

Проще говоря, если вы применили сильное исцеление на 30%, когда у вас было 85% здоровья, ваши сенсоры увидят исцеление на 30% или на 15%?

Если сенсоры фиксируют 30%, значит, всё работает корректно.

Если сенсоры фиксируют 15%, значит, ваш код неправильно оценивает исцеление и нуждается в корректировке вниз.

Один из способов правильной настройки — уменьшать healAt, когда сенсоры обнаруживают исцеление, приводящее игрока к полному здоровью, например, следующим образом:

---

```
curDef->condition = [](GameSensors* sensors) -> bool {  
    static float healAt = 50;  
    if (sensors->detectedStrongHeal()) {  
        ❶      if (sensors->getStrongHealMaxed()) {  
            healAt--;  
        } else {  
            auto newHealAt = 100 - sensors->getStrongHealIncrease();  
            healAt = (healAt + newHealAt) / 2.00f;  
        }  
        sensors->clearStrongHealInfo();  
    }  
    return sensors->getHealthPercent() > healAt;  
};
```

---

Этот код почти такой же, как Listing 11-4, но он добавляет конструкцию `if()` для уменьшения `healAt`, если обнаружено максимальное исцеление ❶. В остальном функция должна работать так же, как в Listing 11-4.

Исцеление — это простой случай, но этот код отлично демонстрирует, как можно использовать исправление ошибок (*error correction*) для динамического улучшения поведения ботов.

Еще один более продвинутый случай использования — корректировка выстрелов по упреждению (*skillshots*) с учетом паттернов движения противников.

Каждый игрок использует определенные схемы, когда избегает упраждающих выстрелов.

- Если сенсоры бота могут измерять направление и расстояние, на которое противник движется, уклоняясь от атаки, то код контроллера может корректировать начальную точку, куда бот изначально будет стрелять.

- В этом же сценарии обучение поможет учитьывать такие факторы, как задержка на сервере, скорость передвижения персонажей и другие переменные.

При использовании исправления ошибок обратите внимание, что ваш код будет чище и более портативным, если определения состояний (*state definitions*) будут содержать внутренние данные, а не просто статические переменные.

Кроме того, чтобы избежать перегруженности в определениях состояний, я рекомендую инкапсулировать всю логику исправления ошибок в отдельных модулях, которые можно легко вызывать при необходимости.

## Поиск пути с алгоритмами поиска

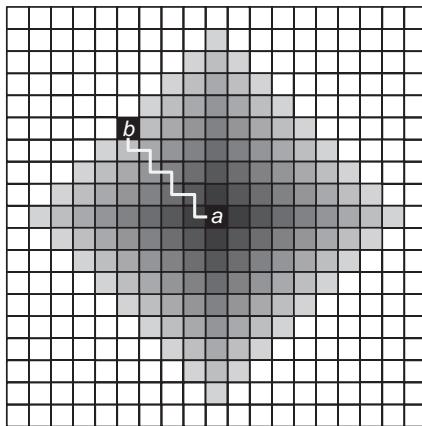
Одна из распространенных задач при написании автономного бота — это вычисление пути, по которому персонаж должен следовать из одной точки в другую. Помимо самой задачи обратного проектирования сенсоров, чтобы определить, какие координаты на игровой карте блокируют движение вперед, есть и алгоритмическая проблема расчета пути внутри карты. Вычисление пути называется **pathfinding**, и геймеры-хакеры часто используют **search algorithm**, чтобы справиться с этой задачей.

## Два распространенных метода поиска

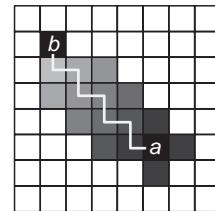
Имея сетку тайлов, начальную точку  $a$  и конечную точку  $b$ , алгоритм поиска рассчитывает путь от  $a$  до  $b$ . Алгоритм делает это, создавая *node at a*, добавляя соседние узлы к  $a$  в список тайлов, которые нужно исследовать (*frontier*), обновляя узел до наилучшего тайла в *frontier*, а затем повторяя процесс, пока узел не достигнет  $b$ . Различные алгоритмы выбирают лучший узел по-разному, используя либо стоимость (*cost*), либо эвристику (*heuristic*), либо и то и другое.

Алгоритм Дейкстры (Dijkstra's algorithm) вычисляет стоимость тайла, исходя из его расстояния от узла  $a$ , и выбирает тайл с наименьшей стоимостью. Представьте себе пустую двумерную сетку с  $a$  в центре. В поиске, следующем алгоритму Дейкстры, *frontier* будет расширяться по круговому шаблону вокруг  $a$ , пока  $b$  не окажется на краю круга (как показано на Figure 11-4).

Эвристический жадный поиск (greedy best-first search) вместо приоритезации узлов по их расстоянию от начальной точки использует эвристику для оценки расстояния от узла в *frontier* до  $b$ . Затем алгоритм выбирает узел с наименьшей оценочной дистанцией. Представьте этот алгоритм в той же сетке: *frontier* будет прямой линией, почти сразу ведущей от  $a$  к  $b$ , как показано в Figure 11-5.



*Figure 11-4: Граница алгоритма Дейкстры. Светлая плитка стоит дороже.*



*Рисунок 11-5: Граница жадного алгоритма поиска по принципу "лучший-первый". Светлая плитка стоит дороже.*

## Как препятствия нарушают поиск

Разница в поведении этих алгоритмов становится более очевидной, когда в сетку добавляются препятствия. Например, если стена разделяет *a* и *b*, алгоритм Дейкстры всегда найдет самый быстрый путь, но с огромными затратами.

Радиус кругового frontier вокруг *a* будет равен длине финального пути; назовем этот радиус *r*. Если границы сетки не обрезают frontier, можно примерно вычислить количество открытых узлов, взяв площадь круга радиусом *r*. Если путь вокруг стены составляет 50 тайлов, алгоритм откроет примерно 7 854 тайла, как показано в следующем уравнении:

$$\pi \times 50^2 = 7,854$$

В той же ситуации эвристический жадный поиск (greedy best-first search) вычислит менее оптимальный путь, но откроет значительно меньше узлов. Труднее предсказать, как будет расширяться frontier, но это сейчас не важно, поэтому мы не будем на этом останавливаться.

В конечном итоге ни один из этих алгоритмов не решает проблему поиска пути идеально. Оптимальный путь медленный, а быстрый путь не оптимален.

Чтобы быстро вычислить оптимальный путь, нужно объединить алгоритм Дейкстры с эвристическим жадным поиском. К счастью, это уже сделано, и результатом стал монстр среди алгоритмов — A-star search (часто просто называют A\*).

A\* использует сумму стоимости (*g*) и эвристики (*h*), чтобы выбрать узлы. Этот результат называется score (оценка). Проще говоря, *score* = *g* + *h*. Как и алгоритм Дейкстры, A\* может вычислить наиболее оптимальный путь от *a* до *b*, но при этом делает это относительно быстро, как и жадный эвристический поиск.

## Алгоритм поиска A\*

Теперь, когда вы знаете основы, давайте напишем код для реализации алгоритма A\* поиска.

Эта реализация будет работать в двумерной сетке. Она не будет позволять диагональное перемещение сначала, но я вскоре расскажу, как можно изменить код, чтобы он работал с диагональным движением.

Весь пример кода для этого раздела находится в каталоге GameHackingExamples/Chapter11\_SearchAlgorithms среди исходных файлов этой книги.

Включенные проекты можно скомпилировать с Visual Studio 2010, но они также должны работать с любым другим C++ компилятором.

Скачайте их по ссылке <https://www.nostarch.com/gamehacking/> и скомпилируйте, если хотите следовать за объяснениями в книге.

Если вы запустите Chapter11\_SearchAlgorithms.exe, вы сможете определить свою собственную 20×20 сетку и наблюдать, как алгоритм вычисляет путь поиска.

## Создание узла A\*

Для начала определите пустой класс AStarNode, как показано ниже:

---

```
typedef std::shared_ptr<class AStarNode> AStarNodePtr;
class AStarNode
{
public:
};
```

Этот код определяет класс AStarNode и тип std::shared\_ptr, названный AStarNodePtr, чтобы упростить создание безопасных указателей на этот класс.

Далее, в публичной области этого класса, объявите переменные-члены для координат x и y узла, стоимости (cost) и оценки (score) узла:

---

```
int x;
int y;
int g, score;
```

Дополнительно, вам нужен публичный член типа AStarNodePtr, который ссылается на родительский узел:

---

```
AStarNodePtr parent;
```

После объявления всех переменных-членов, объявите публичный конструктор, который инициализирует их при создании экземпляра, как показано ниже:

---

```
AStarNode(int x, int y, int cost, AStarNodePtr p, int score = 0)
    : x(x), y(y), g(cost), score(score), parent(p)
{}
```

Теперь, чтобы упростить создание безопасных указателей, добавьте статическую вспомогательную функцию, как показано ниже:

---

```
static AStarNodePtr makePtr(
    int x, int y, int cost,
    AStarNodePtr p,
    int score = 0)
{
    return AStarNodePtr(new AStarNode(x, y, cost, p, score));
```

Эта функция makePtr() создаёт новый экземпляр AStarNode и возвращает его, обернутым в AStarNodePtr.

Давайте повторим. Класс AStarNode имеет переменные-члены: x, y, g, score и parent. Когда класс создаётся, все эти члены инициализируются значениями, переданными в конструктор, за

исключением score, который является необязательным (так как он используется только при копировании экземпляра AStarNode) и устанавливается в 0, если не передан.

Далее, определите публичную функцию-член, чтобы вычислять эвристику, когда заданы координаты назначения:

---

```
int heuristic(const int destx, int desty) const
{
    int xd = destx - x;
    int yd = desty - y;
❶    return abs(xd) + abs(yd);
}
```

---

Эта функция возвращает Манхэттенскую эвристику расстояния ❶, которая используется для вычисления расстояния в сетках, где невозможны диагональные перемещения:

$$|\Delta x| + |\Delta y|$$

Чтобы вычислить путь, позволяющий диагональное перемещение, вам нужно изменить эту функцию, чтобы использовать Евклидову эвристику расстояния, которая выглядит так:

$$\sqrt{(\Delta x \times \Delta x) + (\Delta y \times \Delta y)}$$

Классу также нужна функция, обновляющая score. Добавьте эту функцию в публичную область класса следующим образом:

---

```
#define TILE_COST 1
void updateScore(int endx, int endy)
{
    auto h = this->heuristic(endx, endy) * TILE_COST;
    this->score = g + h;
}
```

---

Теперь score должен изменяться на  $g + h$ , когда заданы координаты назначения для вычисления  $h$ .

Подводя итог, классу узла также нужна функция, которая сможет вычислить всех его дочерних узлов. Эта функция могла бы делать это, создавая новые узлы для каждой клетки, соседней с текущим узлом. Каждый новый узел ссылается на текущий узел, как на родительский, поэтому класс должен уметь создавать AStarNodePtr на копию текущего узла. Вот как это работает:

---

```

AStarNodePtr getCopy()
{
    return AStarNode::makePtr(x, y, g, parent, score);
}
std::vector<AStarNodePtr> getChildren(int width, int height)
{
    std::vector<AStarNodePtr> ret;
    auto copy = getCopy();
    if (x > 0)
①        ret.push_back(AStarNode::makePtr(x - 1, y, g + TILE_COST, copy));
    if (y > 0)
②        ret.push_back(AStarNode::makePtr(x, y - 1, g + TILE_COST, copy));
    if (x < width - 1)
③        ret.push_back(AStarNode::makePtr(x + 1, y, g + TILE_COST, copy));
    if (y < height - 1)
④        ret.push_back(AStarNode::makePtr(x, y + 1, g + TILE_COST, copy));
    return ret;
}

```

---

Эта функция создаёт дочерние узлы в  $(x - 1, y)$  ①,  $(x, y - 1)$  ②,  $(x + 1, y)$  ③ и  $(x, y + 1)$  ④. Их родитель — узел, который вызвал `getChildren`, а их `g` равен `g` родителя плюс `TILE_COST`.

Чтобы разрешить диагональное перемещение, эта функция должна добавить дочерние узлы в  $(x - 1, y - 1)$ ,  $(x + 1, y - 1)$ ,  $(x + 1, y + 1)$ , и  $(x - 1, y + 1)$ . Кроме того, если перемещение по диагонали стоит дороже (то есть, если персонажу требуется больше времени, чтобы его выполнить), то вам также нужно сделать следующее:

1. Изменить `TILE_COST` на 10.
2. Определить константу `DIAG_TILE_COST` как `TILE_COST`, умноженный на увеличение времени. Если шаг по диагонали занимает в 1.5 раза больше времени, `DIAG_TILE_COST` будет 15.
3. Дать диагональным дочерним узлам `g`, равный `g` родителя плюс `DIAG_TILE_COST`.

Чтобы завершить `AStarNode`, объяявите операторы для сравнения приоритета и равенства двух узлов. Вы можете разместить эти объявления вне класса в глобальной области, вот так:

---

```

① bool operator<(const AStarNodePtr &a, const AStarNodePtr &b)
{
    return a.score > b.score;
}
② bool operator==(const AStarNodePtr &a, const AStarNodePtr &b)
{
    return a.x == b.x && a.y == b.y;
}

```

---

Эти операторы позволяют `std::priority_queue` сортировать узлы по **оценке** ① и `std::find` определять равенство узлов по **расположению** ②.

## Написание функции поиска A\*

Теперь, когда вы завершили класс AStarNode, вы можете запрограммировать саму функцию поиска. Начнем с объявления прототипа функции:

```
template<int WIDTH, int HEIGHT, int BLOCKING>
bool doAStarSearch(
    int map[WIDTH][HEIGHT],
    int startx, int starty,
    int endx, int endy,
    int path[WIDTH][HEIGHT])
{ }
```

Прототип принимает ширину и высоту карты игры, а также значение, обозначающее блокирующую плитку на карте, в качестве параметров шаблона. Функция doAStarSearch() также принимает саму карту (map), начальные координаты (startx и starty), координаты назначения (endx и endy), а также пустую карту (path), в которую будет записан рассчитанный путь после завершения работы функции.

### NOTE

Первые три параметра являются шаблонными параметрами, поэтому их можно передавать в виде компиляционных констант. Я сделал так в этом примере, чтобы явно объявить размер массива для параметров map и path, а также задать определенное значение для обозначения блокирующих плиток на карте. На практике, карта, загружаемая из игры, скорее всего, будет иметь динамический размер, и, возможно, вам потребуется более надежный способ передачи этих данных.

Далее, функции doAStarSearch() нужен отсортированный список, чтобы хранить фронтир (границу поиска), и контейнер для отслеживания всех созданных узлов, чтобы можно было обновлять оценку (score) и родителя существующего узла, если он открылся в качестве потомка другого родителя. Вы можете создать их следующим образом:

```
std::vector<AStarNodePtr> allNodes;
std::priority_queue<AStarNodePtr> frontier;
```

Фронтир (frontier) определен как std::priority\_queue, так как он **автоматически** сортирует узлы **на основе их оценки** (score). Контейнер узлов (allNodes) определен как std::vector.

Теперь давайте создадим первый узел:

```
auto node = AStarNode::makePtr(startx, starty, 0, nullptr);
node->updateScore(endx, endy);
allNodes.push_back(node);
```

Первый узел — это осиротевший узел без стоимости на позиции (startx, starty). Узел получает оценку (score), основанную на возвращаемом значении функции updateScore(), после чего добавляется в контейнер allNodes.

Теперь, когда узел добавлен в контейнер, пришло время написать основную часть алгоритма A\*, начиная с простого цикла:

---

```
while (true) {  
}
```

---

До особых указаний весь остальной код в этом разделе будет размещаться внутри этого цикла, в порядке, описанном далее.

Теперь первый шаг — это проверка goal state (конечного состояния). В данном случае цель — это найти путь, по которому персонаж будет следовать к следующей точке маршрута. Этот момент наступает, когда позиция узла (endx, endy).

Таким образом, для проверки goal state программа должна определить, достиг ли узел этих координат или нет. Вот как должна выглядеть эта проверка:

---

```
if (node->x == endx && node->y == endy) {  
    makeList<WIDTH, HEIGHT>(node, allNodes, path);  
    return true;  
}
```

---

Когда конечное состояние достигнуто, программа возвращает true вызывающему коду и заполняет path финальным маршрутом.

На данный момент предположим, что функция makeList() может заполнить path за вас; Я покажу вам эту функцию чуть позже.

Если конечное состояние не достигнуто, вам необходимо развернуть дочерние узлы текущего узла, а это на самом деле довольно сложный процесс:

---

```
auto children = node->getChildren(WIDTH, HEIGHT);  
for (auto c = children.begin(); c != children.end(); c++) {  
    ❶    if (map[(*c)->x][(*c)->y] == BLOCKING) continue;  
  
    auto found = std::find(allNodes.rbegin(), allNodes.rend(), *c);  
    ❷    if (found != allNodes.rend()) {  
        ❸        if (*found > *c) {  
            (*found)->g = (*c)->g;  
            (*found)->parent = (*c)->parent;  
            (*found)->updateScore(endx, endy);  
        }  
        ❹        (*c)->updateScore(endx, endy);  
        frontier.push(*c);  
        ❺        allNodes.push_back(*c);  
    }  
}
```

---

После вызова node->getChildren для генерации списка узлов, которые можно добавить в frontier, код итерируется по каждому дочернему узлу и игнорирует те, которые находятся на заблокированных плитках ❶.

Далее, для каждого дочернего узла код проверяет, не был ли уже открыт узел в тех же координатах ②. Если да и если score существующего узла больше, чем score нового дочернего узла, то существующий узел обновляется: меняется родитель, стоимость (cost) и score нового дочернего узла при помощи if()-оператора ③.

Если новый узел не имеет "брата от другого родителя", он просто добавляется в frontier ④ и в список узлов ⑤.

Также обратите внимание, что std::find использует reverse begin и reverse end итераторы allNodes вместо обычных итераторов ①.

Пример делает это, потому что новые узлы добавляются в конец вектора, и дубликаты узлов обычно располагаются ближе к концу вектора. (Этот шаг также можно выполнить непосредственно с frontier, но std::priority\_queue не поддерживает итерирование по узлам, и добавление сортировки сделало бы код слишком громоздким для печати.)

В конце концов, функция столкнётся с ситуацией, когда новых дочерних узлов для добавления в frontier не останется; следующий if()-оператор обрабатывает эту ситуацию:

---

```
if (frontier.size() == 0) return false;
❶ node = frontier.top();
❷ frontier.pop();
```

---

Этот код назначает node самым дешёвым узлом из frontier ❶, удаляет его из frontier ❷ и запускает повторение цикла.

Если frontier в итоге окажется пустым, функция возвращает false вызывающему коду, так как искать больше нечего.

## Создание списка пути

Наконец, пришло время реализовать функцию makeList():

---

```
template<int WIDTH, int HEIGHT>
void makeList(
    AStarNodePtr end,
    std::vector<AStarNodePtr> nodes,
    int path[WIDTH][HEIGHT])
{
    for (auto n = nodes.begin(); n != nodes.end(); n++)
❶        path[(*n)->x][(*n)->y] = 2;
    auto node = end;
    while (node.get() != nullptr) {
❷        path[node->x][node->y] = 1;
        node = node->parent;
    }
}
```

---

Эта функция обновляет path, добавляя как список закрытых узлов ❶, так и вычисленный путь ❷.

В этом примере значение 2 представляет закрытые узлы, а 1 представляет узлы пути. Программа вычисляет узлы в пути, следуя по родительским узлам от целевого узла, пока не достигнет начального узла, который является сиротой с nullptr в качестве

родителя.

## Когда A\* поиск особенно полезен

Обязательно поиграйтесь с примером кода и исполняемым файлом из предыдущего раздела, потому что это единственный способ действительно понять поведение A поиска\*.

В современных играх вы обычно можете отправить пакет с местоположением или даже эмулировать клик на карте в нужной точке. Но когда вы столкнётесь с ситуацией, где действительно нужно вычислить путь, вы будете рады, что изучили A\*.

На самом деле существует множество ситуаций, в которых расчет пути может быть полезен:

**Выбор целей** - Когда ваш бот выбирает цели для атаки, возможно, вам стоит проверить, может ли ваш персонаж вообще до них добраться. В противном случае, если враг изолирован в недоступной комнате, ваш бот может застрять, бесконечно пытаясь атаковать его!

**Выбор трупа** - Когда ваш бот находится в состоянии сбора добычи, он определяет, какие трупы стоит открыть. Вы можете оптимизировать этот процесс, всегда пытаясь обыскивать ближайший труп в первую очередь.

**Эмуляция движений мыши** - В редких случаях некоторые строго защищенные игры действительно связывают внутриигровые действия с движениями мыши, чтобы убедиться, что бот не запущен. В этом случае вам, возможно, придется эмулировать движения мыши. Используя модифицированную версию A\*, где экран — это карта, нет блокирующих тайлов, а стоимость узлов слегка случайна, вы можете рассчитывать движения мыши так, чтобы они выглядели естественными при их эмуляции.

**Кайтинг монстров** - Если вам когда-либо потребуется написать код для кайтинга монстров, вы можете реализовать A\* с целевым состоянием быть N единиц от всех существ. Используя тот же механизм стоимости, что показан в этой главе, можно настроить эвристику так, чтобы узлы, расположенные ближе к существам, имели большую стоимость. Кайтинг не является стандартным сценарием использования, и эвристика потребует множества корректировок, но, когда она настроена, она работает поразительно хорошо. Некоторые реализации могут кайтить любое количество монстров лучше, чем человек!

## Предсказание движений врага

Если вы пишете бота, который сражается с другими игроками, вы можете использовать A\* для предсказания их движений и соответствующего реагирования. Например, если враг начинает убегать, ваш бот может предположить, что тот направляется на свою базу, рассчитать его маршрут и использовать заклинание, чтобы заблокировать путь или даже телепортироваться в точку, где враг, скорее всего, окажется.

Это всего лишь несколько примеров использования поиска A\*, и

по мере улучшения ваших ботов вы определенно найдете еще больше вариантов. В оставшейся части главы я расскажу о некоторых популярных автоматизированных хаках, которые вы можете реализовать, используя техники, описанные в этой книге.

### ДРУГИЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ ПОИСКА A\*

A\* используется не только для вычисления маршрутов. Если добавить абстракции поверх класса AStarNode, вы можете адаптировать тот же алгоритм для любой задачи поиска. В реальности A\* — это просто взвешенная итерация по многомерному набору данных, которая продолжается до тех пор, пока не найден целевой объект. Таким образом, он может решить любую проблему, которая может быть представлена в виде многомерного набора данных.

Более продвинутые применения A\* включают игру в шахматы и шашки. Когда A\* сочетается с трехмерной эвристикой Манхэттенского расстояния и реализацией поиска в глубину, он может использоваться даже для решения Кубика Рубика.

К сожалению, я не буду углубляться в эти варианты использования. Если вы хотите стать действительно хороши в алгоритмах поиска, я рекомендую вам изучить их дополнительно в интернете.

## Распространенные и крутые автоматизированные хаки

Теперь, когда вы увидели шаблоны проектирования и алгоритмы, необходимые для создания эффективных, самообучающихся ботов, пришло время узнать о некоторых популярных автоматизированных хаках, которые выходят за рамки простого исцеления и поиска пути. Поднимемся на высоту 10 000 футов, чтобы изучить два типа ботов на высоком уровне.

## Грабеж с пещерными ботами

Обсуждая теорию управления, конечные автоматы и поисковые алгоритмы, я упомянул идею пещерного бота, который убивает существ, подбирает добычу и гуляет по пещерам. Способности пещерных ботов могут сильно различаться.

## **Депонирование золота и пополнение запасов**

Если вы хотите оставить персонажа ботить в течение нескольких дней без остановки, вам понадобятся депозитор и заправщик. Депозитор может класть добычу в ваш банк или хранилище, в то время как заправщик пополняет ваши зелья, руны и другие припасы. Эти функции можно описать шестью основными состояниями:

- **Покинуть точку появления** – Условие выполняется, если персонаж находится в зоне появления или пещере, если ему нечего депонировать и у него достаточно припасов. Переход в это состояние происходит при выходе из зоны появления или пещеры.

- **Идти в город** – Условие выполняется, если персонаж находится в зоне появления или в пещере. Переход в это состояние происходит при перемещении из зоны появления или пещеры в город.

- **Депозит** – Условие выполняется, если персонаж находится в зоне появления или пещере, либо если персонаж находится в городе и ему нечего депонировать. Переход в это состояние происходит при помещении добычи в банк или хранилище.

- **Снять деньги** – Условие выполняется, если персонаж находится в зоне появления или пещере, либо в городе и не имеет припасов для покупки или у него достаточно золота для приобретения припасов. Переход в это состояние происходит при снятии золота из банка или хранилища.

- **Купить припасы** – Условие выполняется, если персонаж находится в зоне появления или пещере, либо если у него достаточно припасов, чтобы начать охоту. Переход в это состояние происходит при покупке припасов.

- **Войти в точку появления** – Условие выполняется, если персонаж находится в зоне появления или пещере. Переход в это состояние происходит при перемещении в зону появления или пещеру.

Эти состояния будут идти перед состояниями, связанными с следованием по путевым точкам (я описываю пару таких состояний в “A Complex Hypothetical State Machine” на странице 228) векторе объектов StateDefinition. Размещение их первыми дает им приоритет над оставанием в пещере, при этом по-прежнему позволяя персонажу выбирать цели, убивать и собирать добычу с монстров по пути в город.

В зависимости от места, где идет охота, и от того, как вы хотите, чтобы бот себя вел, вы можете также задать в состояниях выбора целей правило не атаковать существ, если персонаж не находится в зоне появления или пещере. Кроме того, можно добавить еще одно состояние перед “идти в город”, которое атакует только существ, блокирующих путь персонажа к городу.

## Использование персонажа в качестве приманки

Две другие особенности cavebot, которые могут сделать вашего бота потрясающим, — это режим приманки (lure mode) и динамическая приманка (dynamic lure). Вы не стали бы реализовывать эти две функции как настоящие состояния в сложном боте, скорее, вы бы использовали их для информирования состояний бота, отвечающих за выбор целей и передвижение, чтобы помочь боту принимать решения.

Вы можете контролировать режим приманки с помощью специальных путевых точек на вашем маршруте, и код будет говорить вашим состояниям выбора целей атаковать существ только в том случае, если бот застрял, аналогично механизму, обсуждавшемуся для передвижения в город или из города. Разница в том, что режим приманки можно включать и выключать в разных зонах пещеры, позволяя вам заманивать сразу несколько групп монстров в определенные места, прежде чем атаковать их.

Это может сделать вашего бота гораздо более эффективным, поскольку определенные типы персонажей могут лучше всего справляться с уничтожением большого количества монстров одновременно.

Динамическая приманка похожа, но вместо включения и выключения в определенных точках с помощью путевых точек, вы можете автоматически включать режим приманки, когда на экране недостаточно монстров.

Например, бот с функцией динамической приманки может приказать состояниям выбора цели не атаковать ни одно существо, пока на экране не появится пять монстров. Когда на экране окажутся пять монстров, состояния выбора цели возобновят атаку и будут использовать тактику "кайтинга" до тех пор, пока все пять монстров не будут убиты, после чего бот снова переключится в режим приманки до тех пор, пока снова не появится стая подходящего размера.

Если ваш персонаж достаточно быстрый, чтобы убежать от монстров, вам придется изменить состояния передвижения бота так, чтобы он двигался медленнее, когда включен режим приманки и поблизости есть существа.

В противном случае ваш персонаж будет оставлять монстров позади, не убивая их.

Вы можете замедлить персонажа, добавив состояние перед состоянием следования по пути в вашей конечной автомате состояний, которое слегка задерживает передвижение, когда режим приманки включен, а все существа находятся слишком далеко.

## Позволение игрокам писать скрипты для пользовательского поведения

Почти каждый cavebot включает интерфейс скриптов, который позволяет игрокам добавлять собственные поведения. Вы могли бы реализовать этот интерфейс как способ указывать пользовательские путевые точки, заклинания для использования или предметы для сбора.

В более продвинутых ботах вы могли бы сделать выбор целей, сбор лута, хождение по маршрутам и приманивание существ как можно более динамическими, чтобы игроки могли добавлять уникальные функции. Если вы реализуете свою автоматизацию на Lua, трети лица смогут легко улучшать и расширять способности бота.

Сделать так, чтобы писать скрипты для вашего бота было легко, — это снимет с вас большую часть работы, так как другие программисты, играющие в эту игру, могут выпускать скрипты, добавляющие поддержку новых охотничьих зон и улучшающие автоматизацию.

Подобные скриптовые сервисы распространены в бот-сообществах, и игроки часто создают и продают профессиональные скрипты, интегрируемые с ботами.

## Автоматизация боя с варботами

Еще один класс автоматизированных ботов используется для PvP-боя (игрок против игрока).

Эти варботы (warbots или PvP-боты) имеют много функций, которые относятся к категории отзывчивых механизмов (responsive hacks) или ESP-хаков, поскольку боты сфокусированы на реагировании на входящий урон или заклинания, обнаружении скрытых врагов и предоставлении игроку информационного преимущества.

Полностью автоматизированные варботы редки, но я уже обсуждал, как можно использовать некоторые методы автоматизации, чтобы, например:

- делать целителей умнее,
- обучать ботов использовать более точные умения,
- предсказывать пути передвижения игроков, чтобы остановить их в нужный момент.

Теперь давайте рассмотрим еще несколько хаков, которые находятся на грани между отзывчивыми, ESP и автоматизированными методами.

### NOTE

В играх, полностью основанных на PvP, таких как поля сражений или стратегии в реальном времени, некоторые игроки могут просто называть такие боты "боевыми ботами" (warbots), поскольку их единственное назначение — война или PvP.

## Autowall Bots (Боты-автостены)

Если у вашего персонажа есть заклинание для создания временной стены, вы можете запрограммировать бота, который автоматически блокирует вражеских игроков, когда они входят в узкие коридоры.

Используя коррекцию ошибок, бот мог бы научиться, насколько далеко от врага нужно ставить стену.

С действительно креативной инженерией бот мог бы даже научиться, какие враги умеют перепрыгивать через стены,

отслеживая, каким врагам удается преодолеть стену, прежде чем она исчезнет.

## Autosnipe Bots (Боты-автоснайперы)

Для персонажей с дальнобойными умениями или глобальными казнящими заклинаниями можно использовать автоматизацию, чтобы определять, когда у врага заканчивается здоровье, и автоматически использовать заклинание для добивания.

Также можно использовать коррекцию ошибок, чтобы точнее определить, куда стрелять снайперским умением.

Если нельзя точно рассчитать урон, коррекция ошибок поможет боту понять, насколько сильно бьет заклинание и настроить порог срабатывания соответствующим образом.

## Autokite Bots (Боты-автокайтеры)

Если вы играете за кэрри-персонажа, который наносит основной урон с ближней дистанции, вам может понадобиться бот, который автоматически кайти́т (kite) врагов.

Используя набор состояний, схожий с теми, что использует cavebot, можно создать бота, который автоматически кайти́т противников, когда вы их атакуете.

Когда вы перестаете атаковать врага, бот прекращает кайти́нг.

Используя поиск A\*, можно улучшить механизм кайта, чтобы обходить нескольких врагов или, если вам нужно убежать во время атаки, бот мог вести механизм кайта в безопасное место, например, на базу вашей команды или в нейтральную зону.

## Заключительные мысли (Closing Thoughts)

К этому моменту вы должны быть готовы выйти и создать несколько действительно классных ботов. Не беспокойтесь, если вы всё ещё не полностью освоили техники из этой главы; лучший способ учиться — просто погрузиться в процесс и начать хакать. Используйте тысячи строк примеров кода, предоставленных в этой книге, чтобы начать без необходимости работать с нуля, и, самое главное, получайте удовольствие!

В следующей главе я расскажу о способах, с помощью которых боты могут скрываться от механизмов защиты от читов, представляющих собой программное обеспечение, которое игры используют для обнаружения и блокировки ботоводов.

## 12) Оставаться скрытым



Взлом игр - это постоянно развивающаяся практика, игра в кошки-мышки между хакерами и разработчиками игр, где каждый партития работает, чтобы подмять под себя другую. Пока люди создают ботов, игровые компании будут находить способы препятствовать развитию ботов и запрещать игроков, использующих ботов. Однако вместо того, чтобы делать свои игры изначально более сложными для взлома, игровые компании уделяют особое внимание обнаружении.

Крупнейшие игровые компании имеют очень сложные комплексы обнаружения, называемые античит-программы. В начале этой главы я расскажу о возможностях самых распространенных античит-комплексов. После того как вы узнаете, как эти комплексы обнаруживают ботов, я расскажу вам о нескольких эффективных способах их обхода.

### Известное программное обеспечение для защиты от читов (Prominent Anti-Cheat Software)

Наиболее известные античит-системы используют те же методы, что и большинство антивирусных программ, сканируя ботов и помечая их как угрозы. Некоторые античит-системы являются динамическими, то есть их внутренняя работа и возможности могут изменяться в зависимости от игры, которую они защищают. Разработчики античит-программ также отслеживают и исправляют свои системы, чтобы противостоять программам для обхода защиты, поэтому всегда проводите собственное углублённое исследование любого античит-программного обеспечения, с которым вам предстоит столкнуться.

Когда такие системы обнаруживают бота, они помечают учётную запись ботовода для блокировки. Каждые несколько

недель администраторы игр блокируют помеченные аккаунты в **волне банов (ban wave)**. Компании предпочитают банить игроков волнами, а не накладывать мгновенные блокировки, так как это более выгодно. Если игрока банят через несколько недель после начала игры, он уже знаком с игровым процессом и с большей вероятностью купит новую копию игры, чем если бы его забанили сразу после первого запуска бота.

Существуют десятки античит-систем, но я сосредоточусь на пяти пакетах, которые наиболее распространены и хорошо изучены: *PunkBuster*, *ESEA Anti-Cheat*, *Valve Anti-Cheat (VAC)*, *GameGuard* и *Warden*.

## Набор инструментов PunkBuster (The PunkBuster Toolkit)

*PunkBuster*, созданный компанией *Even Balance Inc.*, является оригинальным инструментом античит-защиты. Многие игры используют *PunkBuster*, но чаще всего он встречается в шутерах от первого лица, таких как *Medal of Honor*, *Far Cry 3* и несколько частей серии *Battlefield*.

Этот инструмент использует множество методов обнаружения, среди которых наиболее грозными являются **обнаружение по сигнатурям (Signature-Based Detection, SBD)**, создание скриншотов и проверка хешей. *PunkBuster* также известен тем, что накладывает **аппаратные бани (hardware bans)**, которые навсегда блокируют сам компьютер читера, а не только его учётную запись. Это достигается за счёт снятия отпечатков серийных номеров аппаратных компонентов и блокировки входа с устройств, соответствующих этим данным.

## Обнаружение по сигнатурям (Signature-Based Detection)

*PunkBuster* сканирует память всех процессов в системе, в которой он запущен, проверяя её на наличие байтовых паттернов, уникальных для известных читов. Эти паттерны называются **сигнатурами (signatures)**. Если *PunkBuster* обнаруживает сигнатуру, игрок помечается для блокировки. *PunkBuster* выполняет сканирование памяти от имени ядра, используя функцию Windows API **NtQueryVirtualMemory()**, а также периодически инициирует сканирование нескольких скрытых процессов.

Обнаружение по сигнатурям намеренно разработано как метод, не зависящий от контекста, но у него есть серьёзный недостаток — **ложные срабатывания (false positives)**. 23 марта 2008 года группа хакеров попыталась доказать существование этой уязвимости, заспамив публичные чаты текстовой строкой, которая была идентифицирована *PunkBuster* как сигнатура чита. Поскольку SBD слепо сканирует память на предмет совпадений с сигнатурами, все игроки, находившиеся в этих чатах, были ошибочно помечены как ботоводы.

Это вызвало блокировку тысяч честных игроков без каких-либо

оснований. Подобная ситуация повторилась в ноябре 2013 года: *PunkBuster* ошибочно забанил тысячи игроков в *Battlefield 4*. В тот раз никто не пытался доказать уязвимость системы — компания просто добавила в свою базу данных ошибочную сигнатуру.

*PunkBuster* решил обе проблемы, восстановив доступ игрокам, однако эти инциденты показывают, насколько агрессивно работает система *SBD*. С тех пор, после этих атак, *SBD* сократил количество ложных срабатываний, проверяя сигнатуры только в заранее определённых бинарных смещениях.

## Скриншоты (Screenshots)

Как ещё один метод обнаружения ботов, *PunkBuster* периодически делает скриншоты экрана игрока и отправляет их на центральный сервер. Этот метод обнаружения неудобен и гораздо слабее, чем *SBD*. Взломщики игр считают, что *PunkBuster* реализовал эту функцию, чтобы предоставить администраторам игр доказательства в спорах с ботоводами.

## Проверка хэша (Hash Validation)

Помимо использования *SBD* и скриншотов, *PunkBuster* обнаруживает ботов, создавая криптографические хэши исполняемых файлов игры на компьютере игрока и сравнивая их с хэшами, хранящимися на центральном сервере. Если хэши не совпадают, игрок помечается для блокировки. Эта проверка выполняется только для бинарных файлов на файловой системе, но не для исполняемых файлов в памяти.

## Набор инструментов ESEA Anti-Cheat (The ESEA Anti-Cheat Toolkit)

Набор инструментов *ESEA Anti-Cheat* используется *E-Sports Entertainment Association (ESEA)*, в первую очередь для её лиги *Counter-Strike: Global Offensive*. В отличие от *PunkBuster*, этот инструмент известен тем, что генерирует очень мало ложных срабатываний и при этом очень эффективно выявляет читеров.

Возможности обнаружения *ESEA Anti-Cheat* похожи на возможности *PunkBuster*, но с одним важным отличием. Алгоритм *SBD* в *ESEA Anti-Cheat* выполняется на уровне драйвера ядра, используя три различных функции Windows Kernel:

- **MmGetPhysicalMemoryRanges()**
- **ZwOpenSection()**
- **ZwMapViewOfSection()**

Эта реализация делает античит-систему почти невосприимчивой к подмене памяти (обычный метод обхода *SBD*), так как функции, которые используются для сканирования, гораздо сложнее перехватить, когда они вызываются из драйвера.

## Набор инструментов VAC (The VAC Toolkit)

VAC — это античит-система *Valve Corporation*, применяемая в её собственных играх и во многих сторонних играх, доступных на платформе *Steam*. VAC использует методы *SBD* и проверки хэшей, схожие с методами *PunkBuster*. Однако, помимо этого, он также выполняет сканирование кеша системы доменных имен (DNS) и валидацию бинарных файлов.

### Сканирование кеша DNS (DNS Cache Scans)

DNS — это протокол, который обеспечивает преобразование между доменными именами и IP-адресами, а кеш DNS хранит эту информацию на компьютере. Когда алгоритм *SBD* в VAC обнаруживает чит-программы, VAC сканирует кеш DNS игрока на предмет доменных имен, связанных с читерскими сайтами.

Неизвестно, требуется ли положительный результат сканирования кеша DNS для того, чтобы алгоритм *SBD* в VAC пометил игрока для блокировки, или же сканирование кеша DNS просто становится еще одним аргументом против уже помеченного игрока.

**Н О Т Е** Чтобы посмотреть ваш кеш DNS, введите `ipconfig /displaydns` в командной строке. Да, VAC отслеживает и это.

### Проверка бинарных файлов (Binary Validation)

VAC также использует проверку бинарных файлов, чтобы предотвратить изменение исполняемых файлов в памяти. Он сканирует такие модификации, как **IAT, jump и code hooking**, сравнивая хэши бинарного кода в памяти с хэшами того же кода в исполняемых файлах на диске. Если VAC обнаруживает несовпадение, игрок помечается для блокировки.

Этот метод обнаружения довольно мощный, однако первоначальная реализация алгоритма *Valve* была ошибочной. В июле 2010 года система проверки бинарных файлов в VAC **ошибочно забанила 12 000 игроков в Call of Duty**. Модуль проверки бинарных файлов не учёл обновление *Steam*, и игроки были забанены, так как их код в памяти не совпадал с обновлёнными файлами на диске.

### Ложные срабатывания (False Positives)

VAC также сталкивался с проблемами ложных срабатываний. В первой версии система регулярно банила честных игроков за "неисправную память" (faulty memory). Эта же версия ошибочно банила игроков, использующих Cedega — платформу, которая позволяла запускать Windows-игры в Linux. 1 апреля 2004 года *Valve* ошибочно забанила пару тысяч игроков из-за бага с дюпом на сервере (server-side glitch). Два других отдельных инцидента произошли:

- В июне 2011 года

- В феврале 2014 года
- В феврале 2014 года VAC также ошибочно забанил тысячи игроков Team Fortress 2 и Counter-Strike из-за ошибки, подробности которой Valve отказалась раскрыть.

Как и в случае с PunkBuster, эти инциденты показывают, что VAC является очень агрессивной системой.

## Набор инструментов GameGuard (The GameGuard Toolkit)

GameGuard — это античит-система, разработанная компанией INCA Internet Co. Ltd. и используемая во многих MMORPG, включая Lineage II, Cabal Online и Ragnarok Online. Помимо умеренно агрессивного SBD, GameGuard использует руткиты для превентивного предотвращения запуска вредоносного ПО.

## Руткит на уровне пользователя (User-Mode Rootkit)

GameGuard использует руткит на уровне пользователя для блокировки доступа ботов к функциям Windows API, которые они используют. Этот руткит перехватывает функции на **самом низкоуровневом входном пункте**, часто внутри недокументированных функций в **ntdll.dll**, **user32.dll** и **kernel32.dll**.

Вот наиболее важные API-функции, которые GameGuard перехватывает, и что он делает внутри каждой из них:

**NtOpenProcess()** – блокирует любые попытки *OpenProcess()* для защищаемой игры.

**NtProtectVirtualMemory()** – блокирует любые попытки *VirtualProtect()* или *VirtualProtectEx()* для игры.

**NtReadVirtualMemory()** и **NtWriteVirtualMemory()** – блокируют любые попытки *ReadProcessMemory()* и *WriteProcessMemory()* для игры.

**NtSuspendProcess()** и **NtSuspendThread()** – блокируют любые попытки приостановить GameGuard.

**NtTerminateProcess()** и **NtTerminateThread()** – блокируют любые попытки завершить GameGuard.

**PostMessage()**, **SendMessage()** и **SendInput()** – блокируют любые попытки отправки программного ввода в игру.

**SetWindowsHookEx()** – предотвращает глобальное перехватывание ввода с клавиатуры и мыши ботами.

**CreateProcessInternal()** – автоматически обнаруживает и внедряется в новые процессы.

**GetProcAddress()**, **LoadLibraryEx()** и **MapViewOfFileEx()** – предотвращает любые попытки внедрения библиотек в игру или в сам GameGuard.

## Руткит на уровне ядра (Kernel-Mode Rootkit)

*GameGuard* также использует руткит, работающий на уровне драйвера ядра, чтобы предотвращать работу ботов на этом уровне. Этот руткит имеет те же возможности, что и его аналог на уровне пользователя, и работает, перехватывая:

- `ZwProtectVirtualMemory()`
- `ZwReadVirtualMemory()`
- `ZwWriteVirtualMemory()`
- `SendInput()`

и аналогичные функции.

## Набор инструментов Warden (The Warden Toolkit)

Warden, созданный исключительно для игр Blizzard, является самым продвинутым античит-инструментом, с которым я сталкивался. Трудно сказать, что именно делает Warden, так как он загружает динамический код во время выполнения. Этот код, поставляемый в виде скомпилированного шеллкода, обычно выполняет две задачи:

- Обнаружение ботов.
- Периодическая отправка "сигнала сердцебиения" (heartbeat) на игровой сервер.

Значение, отправляемое в сигнале сердцебиения, не задано заранее, а генерируется на основе части кода обнаружения.

Если Warden не выполняет вторую задачу или отправляет неверное значение, сервер игры узнаёт, что программа была отключена или изменена. Более того, бот не может отключить код обнаружения и при этом оставить код отправки сигнала сердцебиения работающим.

### Проблема остановки (The Halting Problem)

Бот, который мог бы отключить код обнаружения Warden и при этом продолжать отправлять сигнал сердцебиения, смог бы решить проблему остановки (halting problem), которую Аллан Тьюринг доказал как нерешаемую в 1936 году. Проблема остановки — это задача определения, с помощью универсального алгоритма, завершится ли выполнение программы или будет выполняться бесконечно.

Так как Warden выполняет обе задачи с помощью одного и того же шеллкода, создание алгоритма, который мог бы отключить только одну из них, является вариантом проблемы остановки.

Алгоритм не может однозначно определить, какие части кода гарантированно выполняются, какие нет, и какие части отвечают за выполнение каждой задачи.

Warden является грозным противником, потому что у вас не только нет способа узнать, от чего именно вы скрываетесь, но и нет способа отключить этот инструмент. Даже если вам удастся

избежать обнаружения сегодня, завтра может быть использован новый метод детекции. Если вы планируете публично распространять ботов, вам в конечном итоге придётся столкнуться с одним из античит-решений, описанных в предыдущих разделах, — и вам придётся его обойти. В зависимости от следа вашего бота, типа обнаружения в игре, которую вы автоматизируете, и вашей реализации, сложность уклонения от одного из этих инструментов может варьироваться от тривиальной до чрезвычайно сложной.

## Тщательное управление следами бота (Carefully Managing a Bot's Footprint)

След бота (*footprint*) — это количество его уникальных, обнаруживаемых характеристик. Например, бот, который перехватывает (*hooks*) 100 функций, обычно легче обнаружить, чем бот, который перехватывает всего 10 функций, потому что первый вносит в код игры на порядок больше изменений, чем второй. Поскольку целевая система обнаружения должна найти только один перехват, разработчику более сложного бота приходится тратить гораздо больше времени, чтобы все его перехваты были максимально скрытыми.

Ещё одна характеристика следа бота — это то, насколько детализирован его пользовательский интерфейс. Если у известного бота есть множество диалоговых окон с определёнными заголовками, игровая компания может просто настроить античит-программу на обнаружение бота путём поиска окон с такими заголовками. Этот же принцип можно применять к именам процессов и файлам.

## Минимизация следа бота (Minimizing a Bot's Footprint)

В зависимости от того, как работает ваш бот, существует множество способов минимизировать его след. Если ваш бот сильно зависит от перехватов, например, можно избежать прямого перехвата кода игры и вместо этого сосредоточиться на перехвате функций *Windows API*. Перехват *Windows API* является удивительно распространённой практикой, поэтому разработчики не могут автоматически считать любую программу, использующую перехват *Windows API*, ботом.

Если у вашего бота хорошо проработанный пользовательский интерфейс, вы можете замаскировать его, удалив все строки из заголовков окон, кнопок и так далее. Вместо этого отображайте изображения, содержащие текст. Если вас беспокоит обнаружение конкретных имён процессов или файлов античит-программой, используйте универсальные имена файлов и заставьте бота копировать себя в новую, случайно выбранную директорию при каждом запуске.

## Сокрытие следов (Masking Your Footprint)

Минимизация следа — это предпочтительный способ избежать обнаружения, но он не является обязательным. Вы также можете обфусцировать ваш бот, что затруднит анализ его работы. Обфускация может помешать разработчикам антивирусных систем изучать ваш бот, а также другим разработчикам — воровать его функциональность. Если вы продаёте бота, обфускация предотвращает его взлом с целью обхода проверки покупки.

Один из распространённых методов обфускации — упаковка (packing). Упаковка исполняемого файла зашифровывает его и прячет внутри другого исполняемого файла. Когда запускается контейнерный исполняемый файл, упакованный исполняемый файл расшифровывается и выполняется в памяти. Если бот упакован, анализ бинарного файла с целью выяснения, что делает бот, становится невозможным, а отладка бота становится значительно сложнее.

Некоторые популярные упаковщики: UPX, Armadillo, Themida и ASPack.

## Обучение бота обнаружению отладчиков (Teaching a Bot to Detect Debuggers)

Если разработчики антивирусных систем (или другие создатели ботов) могут отлаживать бота, они могут разобраться в его работе и, следовательно, остановить его. Если кто-то активно пытается разобрать бота на части, одной упаковки исполняемого файла может быть недостаточно, чтобы избежать обнаружения.

Для защиты от этого боты часто используют методы антиотладки (anti-debugging), которые скрывают логику выполнения, изменяя поведение бота при обнаружении отладчика.

В этом разделе я быстро рассмотрю некоторые известные методы обнаружения отладчиков, а затем покажу несколько трюков по обфускации.

## Вызов CheckRemoteDebuggerPresent() (Calling CheckRemoteDebuggerPresent())

`CheckRemoteDebuggerPresent()` — это функция **Windows API**, которая может определить, **подключён ли отладчик** к текущему процессу.

Пример кода для проверки наличия отладчика может выглядеть так:

```
bool IsRemoteDebuggerPresent() {
    BOOL dbg = false;
    CheckRemoteDebuggerPresent(GetCurrentProcess(), &dbg);
    return dbg;
}
```

Эта проверка довольно проста — она вызывает `CheckRemoteDebuggerPresent()` с текущим процессом и

указателем на булево значение `dbg`. Вызов этой функции — это самый лёгкий способ обнаружить отладчик, но также и очень простой для обхода.

## Проверка обработчиков прерываний (Checking for Interrupt Handlers)

Прерывания (Interrupts) — это сигналы, которые процессор отправляет для вызова соответствующего обработчика в ядре Windows. Прерывания обычно генерируются аппаратными событиями, но их также можно создавать программно с помощью инструкции INT в ассемблере.

Ядро разрешает использование некоторых прерываний, в частности:

- 0x2D
- 0x03

Они вызывают обработчики прерываний на уровне пользователя в виде обработчиков исключений. Эти прерывания можно использовать для обнаружения отладчиков.

Когда отладчик устанавливает точку останова на инструкции, он заменяет её на инструкцию прерывания INT, например INT 0x03. Когда выполняется это прерывание, отладчик получает уведомление через обработчик исключений. В этом обработчике он обрабатывает точку останова, восстанавливает оригинальный код и позволяет приложению продолжить выполнение без сбоев.

При встрече с неизвестным прерыванием некоторые отладчики могут просто пропускать его, позволяя программе выполнять дальше без вызова каких-либо других обработчиков исключений.

Вы можете обнаружить это поведение, намеренно генерируя прерывания внутри обработчиков исключений в своём коде, как показано в Листинге 12-1.

---

```
inline bool Has2DBreakpointHandler() {
    __try { __asm INT 0x2D }
    __except (EXCEPTION_EXECUTE_HANDLER){ return false; }
    return true;
}

inline bool Has03BreakpointHandler() {
    __try { __asm INT 0x03 }
    __except (EXCEPTION_EXECUTE_HANDLER){ return false; }
    return true;
}
```

---

Листинг 12-1: Обнаружение обработчиков прерываний

Во время нормального выполнения эти прерывания вызывают обработчики исключений, которые их окружают в коде. Во время сеанса отладки некоторые отладчики могут перехватывать исключения, вызванные этими прерываниями, и просто игнорировать их, предотвращая выполнение окружающих обработчиков исключений. Таким образом, если прерывания не

вызывают ваш обработчик исключений, значит, присутствует отладчик.

## Проверка аппаратных точек останова (Checking for Hardware Breakpoints)

Отладчики также могут устанавливать точки останова с помощью отладочных регистров процессора; такие точки останова называются аппаратными точками останова (hardware breakpoints). Отладчик может установить аппаратную точку останова на инструкцию, записав её адрес в один из четырёх отладочных регистров.

Когда выполняется адрес, находящийся в отладочном регистре, отладчик получает уведомление. Чтобы обнаружить аппаратные точки останова (а значит, и присутствие отладчика), можно проверить ненулевые значения в любом из четырёх отладочных регистров следующим образом:

---

```
bool HasHardwareBreakpoints() {
    CONTEXT ctx = {0};
    ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    auto hThread = GetCurrentThread();
    if(GetThreadContext(hThread, &ctx) == 0)
        return false;
    return (ctx.Dr0 != 0 || ctx.Dr1 != 0 || ctx.Dr2 != 0 || ctx.Dr3 != 0);
}
```

---

## Вывод отладочных строк (Printing Debug Strings)

`OutputDebugString()` — это функция **Windows API**, которая может использоваться для вывода логов в консоль отладчика. Если отладчик отсутствует, функция вернёт код ошибки. Если отладчик присутствует, функция вернётся без ошибки. Вот как можно использовать эту функцию для простейшей проверки на отладчик:

---

```
inline bool CanCallOutputDebugString() {
    SetLastError(0);
    OutputDebugStringA("test");
    return (GetLastError() == 0);
}
```

---

Как и метод `CheckRemoteDebuggerPresent()`, этот метод очень прост, но также очень легко обходится отладчиком.

## Проверка обработчиков DBG\_RIPEXCEPTION (Checking for DBG\_RIPEXCEPTION Handlers)

Отладчики, как правило, содержат обработчики исключений, которые безразборно перехватывают исключения с кодом `DBG_RIPEXCEPTION` в Windows, что делает этот код очевидным способом обнаружения отладчика. Эти обработчики исключений можно обнаружить тем же способом, которым в [Листинге 12-1](#)

определяется наличие обработчиков прерываний.

---

```
#define DBG_RIPEXCEPTION 0x40010007
inline bool hasRIPEXceptionHandler() {
    __try { RaiseException(DBG_RIPEXCEPTION, 0, 0, 0); }
    __except(EXCEPTION_EXECUTE_HANDLER){ return false; }
    return true;
}
```

---

## Измерение времени выполнения критически важных функций (Timing Control-Critical Routines)

Если разработчик античит-системы отлаживает ваш бот, он, скорее всего, установит точки останова и будет выполнять пошаговую отладку через участки кода, которые критически важны для поведения бота. Вы можете обнаружить такую активность, измеряя время выполнения кода — когда кто-то проходит код пошагово, выполнение занимает намного больше времени, чем обычно.

Например, если функция только ставит перехваты (hooks), можно быть уверенным, что выполнение кода не должно занимать больше десятых доли секунды для выполнения защиты памяти. Можно проверить время выполнения для защиты памяти с помощью функции Windows API GetTickCount(), как показано ниже:

## Проверка отладочных драйверов (Checking for Debug Drivers)

Некоторые отладчики загружают драйверы режима ядра, чтобы помочь в своей работе. Можно обнаружить такие отладчики, пытаясь получить дескриптор их драйверов ядра, например так:

```

bool DebuggerDriversPresent() {
    // массив с именами устройств распространённых отладочных драйверов
    const char drivers[9][20] = {
        "\\\\.\\EXTREM", "\\\\.\\ICEEXT",
        "\\\\.\\NDBGMSG.VXD", "\\\\.\\RING0",
        "\\\\.\\SIWVID", "\\\\.\\SYSER",
        "\\\\.\\TRW", "\\\\.\\SYSERBOOT",
        "\0"
    };

    // Проходим по массиву драйверов
    for (int i = 0; drivers[i][0] != '\0'; i++) {
        // Пытаемся открыть драйвер
        auto h = CreateFileA(drivers[i], 0, 0, 0, OPEN_EXISTING, 0, 0);
        // Если получилось открыть – значит, отладчик присутствует
        if (h != INVALID_HANDLE_VALUE) {
            CloseHandle(h);
            return true;
        }
    }
    return false;
}

```

Существует несколько распространённых имён устройств драйверов режима ядра, которые следует проверять, например, `\\.\EXTREM` и другие, показанные в массиве `drivers`. Если код, получающий дескриптор, выполняется успешно, значит, в системе работает отладчик. Однако, в отличие от предыдущих методов, получение дескриптора одного из этих драйверов не всегда означает, что отладчик подключён именно к вашему боту.

## Техники защиты от отладки (Anti-Debugging Techniques)

Как только вы обнаружили отладчик, существует несколько способов обfuscировать выполнение кода. Например, можно вызвать сбой в отладчике. Следующий код вызывает крах OllyDbg версии 1.10:

---

```
OutputDebugString("%s%s%s%s");
```

---

Строка `"%s%s%s%s"` содержит форматные спецификаторы, и OllyDbg передаёт её в `printf()` без дополнительных аргументов, из-за чего отладчик завершается сбоем.

Можно поместить этот код в функцию, которая вызывается при обнаружении отладчика, но этот метод работает только против OllyDbg.

## Создание неизбежного бесконечного цикла (Causing an Unavoidable Infinite Loop)

Другой метод обfuscации — перегрузка системы, пока отладчик не будет вынужден закрыть бот и сам отладчик. Следующая функция выполняет эту задачу:

---

```
void SelfDestruct() {
    std::vector<char*> explosion;
    while (true)
        explosion.push_back(new char[10000]);
}
```

---

Этот бесконечный цикл while непрерывно добавляет элементы в explosion, пока процесс не исчерпает всю память или кто-то принудительно не завершит процесс.

## Переполнение стека (Overflowing the Stack)

Если вы хотите серьёзно запутать аналитика, можно создать цепочку функций, которая в конечном итоге вызовет переполнение стека, но не напрямую:

Скопировать

```
#include <random>

typedef void (*_recurse)(); // Определение типа указателя на функцию

// Объявление рекурсивных функций
void recurse1(); void recurse2();
void recurse3(); void recurse4();
void recurse5();

// Массив указателей на функции
_recurse recfuncs[5] = {
    &recurse1, &recurse2, &recurse3,
    &recurse4, &recurse5
};

// Определение рекурсивных функций
void recurse1() { recfuncs[rand() % 5](); }
void recurse2() { recfuncs[(rand() % 3) + 2](); }
void recurse3() {
    if (rand() % 100 < 50) recurse1();
    else recfuncs[(rand() % 3) + 1]();
}
void recurse4() { recfuncs[rand() % 2](); }
void recurse5() {
    for (int i = 0; i < 100; i++)
        if (rand() % 50 == 1)

            recfuncs[i % 5]();
    recurse5();
}

// Вызов любой из этих функций приведёт к переполнению стека
```

Проще говоря, эти функции рекурсивно вызываются случайнным образом до тех пор, пока в стеке вызовов не закончится место. Вызов переполнения стека косвенно затрудняет аналитику возможность поставить на паузу выполнение программы и изучить предыдущие вызовы, прежде чем он поймёт, что произошло.

## Вызов BSOD (Causing a BSOD)

Если вы серьёзно относитесь к обfuscации, можно даже вызвать "синий экран смерти" (BSOD) при обнаружении отладчика. Один из способов сделать это — установить процесс бота как критический с помощью функции Windows API SetProcessIsCritical(), а затем вызвать exit(). Поскольку Windows инициирует BSOD при завершении критического процесса, можно

использовать следующий код:

---

```
void BSODBaby() {
    typedef long (WINAPI *RtlSetProcessIsCritical)
        (BOOLEAN New, BOOLEAN *Old, BOOLEAN NeedScb);
    auto ntdll = LoadLibraryA("ntdll.dll");
    if (ntdll) {
        auto SetProcessIsCritical = (RtlSetProcessIsCritical)
            GetProcAddress(ntdll, "RtlSetProcessIsCritical");
        if (SetProcessIsCritical)
            SetProcessIsCritical(1, 0, 0);
    }
}

BSODBaby();
exit(1);
```

---

Или, возможно, вы настроены ещё более злобно, в таком случае можно сделать следующее:

---

```
BSODBaby();
OutputDebugString("%s%s%s%s");
recurse1();
exit(1);
```

---

Предполагая, что вы реализовали все техники, описанные в этом разделе, этот код вызовет BSOD, приведёт к сбою отладчика (если это OllyDbg v1.10), вызовет переполнение стека и завершит выполнение программы. Если один из методов не сработает или будет исправлен, аналитику всё равно придётся разбираться с оставшимися, прежде чем он сможет продолжить отладку.

## Обход детекции на основе сигнатур (Defeating Signature-Based Detection)

Даже при превосходной обfuscации вы не сможете легко победить сигнатурное обнаружение. Инженеры, которые анализируют ботов и пишут сигнатуры, обладают высокой квалификацией, и обfuscация — это, в лучшем случае, просто небольшая помеха, которая лишь немного усложняет их работу.

Чтобы полностью обойти SBD (Signature-Based Detection), нужно перехватить код обнаружения. Для этого необходимо точно знать, как работает система SBD.

Например, PunkBuster использует NtQueryVirtualMemory(), чтобы сканировать память всех запущенных процессов в поисках сигнатур.

Если вы хотите обойти это ограничение, можно внедрить код в процессы PunkBuster, перехватив вызовы NtQueryVirtualMemory().

Когда функция пытается считать память из процесса вашего бота, можно подменить возвращаемые данные. Например, так:

```

NTSTATUS onNtQueryVirtualMemory(
    HANDLE process, PVOID baseAddress,
    MEMORY_INFORMATION_CLASS memoryInformationClass,
    PVOID buffer, ULONG numberOfBytes, PULONG numberOfBytesRead)
{
    // ❶ Если сканирование направлено на этот процесс, скрываем загруженный хук DLL
    if ((process == INVALID_HANDLE_VALUE ||
        process == GetCurrentProcess()) &&
        baseAddress >= MY_HOOK_DLL_BASE &&
        baseAddress <= MY_HOOK_DLL_BASE_PLUS_SIZE)
        return STATUS_ACCESS_DENIED;

    // ❷ Если сканирование направлено на бота, обнуляем возвращаемую память
    auto ret = origNtQueryVirtualMemory(
        process,
        baseAddress,
        memoryInformationClass,
        buffer, numberOfBytes, numberOfBytesRead);

    if (GetProcessId(process) == MY_BOT_PROCESS)
        ZeroMemory(buffer, numberOfBytesRead);

    return ret;
}

```

## Обход детекции на основе сигнатур (Defeating Signature-Based Detection)

Этот перехват `onNtQueryVirtualMemory()` возвращает `STATUS_ACCESS_DENIED` ❶, если `NtQueryVirtualMemory()` пытается считать память, где загружена hook DLL, но возвращает обнулённые данные ❷, если `NtQueryVirtualMemory()` пытается считать память бота. Разница здесь не имеет особого значения — это просто один из способов скрыть процесс бота от вызова `NtQueryVirtualMemory()`. Если вы очень осторожны, можно даже заменить весь буфер случайной последовательностью байтов.

Этот метод работает только для SBD, который выполняется из пользовательского режима, например PunkBuster или VAC. SBD, который работает из режима ядра, например ESEA, или использует непредсказуемые механизмы (например, Warden), обойти сложнее.

Если ваш бот распространяется на несколько десятков или сотен пользователей, убрать все отличительные особенности сложно. Чтобы запутать аналитиков, можно делать разные версии бота для разных пользователей, используя комбинацию следующих методов:

- Компиляция бота разными компиляторами.
- Изменение настроек оптимизации компилятора.
- Переключение между `_fastcall` и `_cdecl`.
- Упаковка бинарных файлов разными упаковщиками.
- Переключение между статической и динамической линковкой библиотек.

Использование этих методов создаёт уникальный бинарный файл для каждого пользователя, но существует предел возможных вариантов. В какой-то момент этот метод перестаёт масштабироваться, и игровые компании всё равно получают сигнатуры для каждой новой версии бота.

Помимо обfuscации и мутации кода, не так много способов победить продвинутые механизмы SBD. Можно попытаться реализовать бота в виде драйвера или руткита режима ядра, чтобы скрыть его, но даже эти методы не гарантируют защиту.

#### NOTE

*Эта книга не рассматривает реализацию бота в драйвере или создание руткитов для его скрытия, так как обе темы довольно сложны. Разработка руткитов – это тема, которая уже освещена в десятках книг. Я бы порекомендовал книгу Билла Бландена "The Rootkit Arsenal: Escape and Evasion in The Dark Corners of The System" (Jones & Bartlett Learning, 2009).*

Некоторые игровые хакеры пытаются перехватывать каждую возможную функцию, связанную с чтением памяти и файловой системой, но всё равно обнаруживаются системами уровня Warden. На самом деле, я рекомендую избегать Warden и Blizzard любой ценой.

## Обход детекции через скриншоты (Defeating Screenshots)

Если вы сталкиваетесь с механизмом детекции, который использует скриншоты как дополнительное доказательство, чтобы ловить ботов, вам повезло. Обойти такую защиту легко: не позволяйте боту быть видимым.

Можно обойти этот метод, используя минимальный интерфейс, который не вносит заметных изменений в клиент игры. Если ваш бот требует HUD или другие отличительные UI-элементы, не переживайте — их тоже можно скрыть во время создания скриншота.

Некоторые версии PunkBuster, например, вызывают функцию Windows API GetSystemTimeAsFileTime() незадолго до создания скриншота. Можно перехватить этот вызов и скрыть интерфейс бота на несколько секунд, чтобы он не попал в снимок экрана:

```
void onGetSystemTimeAsFileTime(LPFILETIME systemTimeAsFileTime) {
    myBot->hideUI(2000); // hide UI for 2 seconds
    origGetSystemTimeAsFileTime(systemTimeAsFileTime);
}
```

Просто перехватите GetSystemTimeAsFileTime() с использованием техник, описанных в разделе "Hooking to Redirect Game Execution" на странице 153, напишите функцию hideUI() и вызовите её перед возобновлением выполнения.

## Обход проверки бинарных файлов (Defeating Binary Validation)

Обойти проверку бинарных файлов так же просто, как не размещать хуки внутри игровых бинарных файлов. Перехваты (jmp hooks) и хуки IAT на функции Windows API встречаются очень часто, поэтому, где только возможно, старайтесь использовать именно их, а не хуки jmp или near-call внутри бинарного файла игры.

В случаях, когда хуки в коде игры неизбежны, можно обмануть систему проверки бинарных файлов античит-программы, перехватив процесс сканирования бинарного кода и подменив данные на такие, которые ожидает увидеть античит.

Как и в случае SBD (Signature-Based Detection), проверка бинарных файлов часто использует NtQueryVirtualMemory() для сканирования памяти. Чтобы обмануть код валидации, сначала нужно перехватить вызов этой функции. Затем напишите функцию, подобную этой, чтобы подменять данные, когда вызывается NtQueryVirtualMemory():

```
NTSTATUS onNtQueryVirtualMemory(
    HANDLE process, PVOID baseAddress,
    MEMORY_INFORMATION_CLASS memoryInformationClass,
    PVOID buffer, ULONG numberofBytes, PULONG numberofBytesRead)
{
    // Вызываем оригинальную функцию
    auto ret = origNtQueryVirtualMemory(
        process, baseAddress,
        memoryInformationClass,
        buffer, numberofBytes, numberofBytesRead);

    // Здесь можно вставить код для подмены данных

    return ret;
}
```

**НОТЕ**

Этот пример предполагает, что у бота есть только один хук и что переменные с префиксом HOOK\_ уже существуют и описывают код, который заменяет хук.

В листинге 12-2 показан код для мониторинга сканирования.

```
// Проверяем, производится ли сканирование текущего процесса?  
bool currentProcess =  
    process == INVALID_HANDLE_VALUE ||  
    process == GetCurrentProcess();  
  
// Проверяем, находится ли хук в диапазоне памяти, который сканируется?  
auto endAddress = baseAddress + numberOfBytesRead - 1;  
bool containsHook =  
    (HOOK_START_ADDRESS >= baseAddress &&  
     HOOK_START_ADDRESS <= endAddress) ||  
    (HOOK_END_ADDRESS >= baseAddress &&  
     HOOK_END_ADDRESS <= endAddress);  
  
❶ if (currentProcess && containsHook) {  
    // Скрываем хук  
}
```

Листинг 12-2: Проверка сканирования памяти с крючком

Когда сканирование памяти затрагивает код, в который внедрён хук (то есть когда currentProcess и containsHook становятся true одновременно), код внутри оператора if() ❶ обновляет выходной буфер, подставляя оригинальный код.

Это означает, что нужно точно знать, где именно в сканируемом блоке находится хук, принимая во внимание, что сканируемый блок может содержать только часть изменённого кода.

Если baseAddress указывает на адрес начала сканирования, HOOK\_START\_ADDRESS указывает, где начинается модифицированный код, endAddress указывает, где заканчивается сканирование, а HOOK\_END\_ADDRESS указывает, где заканчивается модифицированный код, можно использовать простую математическую логику, чтобы определить, какие части изменённого кода присутствуют в сканируемом буфере.

Для этого используются:

writeStart — для хранения смещения изменённого кода внутри сканируемого буфера.

readStart — для хранения смещения сканируемого буфера относительно изменённого кода, в случае если сканирование начинается внутри модифицированного кода.

---

```
int readStart, writeStart;
if (HOOK_START_ADDRESS >= baseAddress) {
    readStart = 0;
    writeStart = HOOK_START_ADDRESS - baseAddress;
} else {
    readStart = baseAddress - HOOK_START_ADDRESS;
    writeStart = baseAddress;
}

int readEnd;
if (HOOK_END_ADDRESS <= endAddress)
    readEnd = HOOK_LENGTH - readStart - 1;
else
    readEnd = endAddress - HOOK_START_ADDRESS;
```

---

Как только вы знаете, сколько байтов нужно заменить, где их разместить и откуда их взять, можно выполнить подмену с помощью трёх строк кода:

---

```
char* replaceBuffer = (char*)buffer;
for ( ; readStart <= readEnd; readStart++, writeStart++)
    replaceBuffer[writeStart] = HOOK_ORIG_DATA[readStart];
```

---

Полностью собранный код выглядит следующим образом:

---

```
NTSTATUS onNtQueryVirtualMemory(
    HANDLE process, PVOID baseAddress,
    MEMORY_INFORMATION_CLASS memoryInformationClass,
    PVOID buffer, ULONG numberOfBytes, PULONG numberofBytesRead) {
    auto ret = origNtQueryVirtualMemory(
        process, baseAddress,
        memoryInformationClass,
        buffer, numberOfBytes, numberofBytesRead);
    bool currentProcess =
        process == INVALID_HANDLE_VALUE ||
        process == GetCurrentProcess();

    auto endAddress = baseAddress + numberofBytesRead - 1;
    bool containsHook =
        (HOOK_START_ADDRESS >= baseAddress &&
        HOOK_START_ADDRESS <= endAddress) ||
        (HOOK_END_ADDRESS >= baseAddress &&
        HOOK_END_ADDRESS <= endAddress);
    if (currentProcess && containsHook) {
        int readStart, writeStart;
        if (HOOK_START_ADDRESS >= baseAddress) {
            readStart = 0;
            writeStart = HOOK_START_ADDRESS - baseAddress;
        } else {
            readStart = baseAddress - HOOK_START_ADDRESS;
            writeStart = baseAddress;
        }

        int readEnd;
        if (HOOK_END_ADDRESS <= endAddress)
            readEnd = HOOK_LENGTH - readStart - 1;
        else
            readEnd = endAddress - HOOK_START_ADDRESS;

        char* replaceBuffer = (char*)buffer;
        for ( ; readStart <= readEnd; readStart++, writeStart++)
            replaceBuffer[writeStart] = HOOK_ORIG_DATA[readStart];
    }
    return ret;
}
```

---

Конечно, если у вас есть несколько хуков, которые необходимо скрыть от проверок бинарной валидации, вам нужно реализовать эту функциональность более надёжным способом, который позволит отслеживать несколько изменённых участков кода.

## Обход руткита античита (Defeating an Anti-Cheat Rootkit)

GameGuard и некоторые другие античит-программы используют руткиты в пользовательском режиме, которые не только обнаруживают ботов, но и активно мешают им запускаться.

Чтобы обойти такой тип защиты, вместо того чтобы искать нестандартные решения, можно полностью скопировать среду и работать внутри этой копии.

Например, если вам нужно записать данные в память игры, необходимо вызвать функцию WriteProcessMemory(), которая экспортируется из kernel32.dll. При вызове этой функции она напрямую вызывает NtWriteVirtualMemory() из ntdll.dll.

GameGuard перехватывает ntdll.NtWriteVirtualMemory(), чтобы запретить запись в память. Однако если NtWriteVirtualMemory() экспортируется из ntdll\_copy.dll, GameGuard не будет перехватывать эту функцию.

Это означает, что можно скопировать ntdll.dll и динамически импортировать все нужные функции, как показано ниже:

```
// Копируем и загружаем ntdll
copyFile("ntdll.dll", "ntdll_copy.dll");

auto module = LoadLibrary("ntdll_copy.dll");

// Динамически импортируем NtWriteVirtualMemory
typedef NTSTATUS (WINAPI* _NtWriteVirtualMemory)
(HANDLE, PVOID, PVOID, ULONG, PULONG);

auto myWriteVirtualMemory = (_NtWriteVirtualMemory)
GetProcAddress(module, "NtWriteVirtualMemory");

// Вызываем NtWriteVirtualMemory
myWriteVirtualMemory(process, address, data, length, &writtenlength);
```

После копирования ntdll.dll этот код импортирует NtWriteVirtualMemory() из копии под именем myWriteVirtualMemory(). После этого бот может использовать эту функцию вместо NtWriteVirtualMemory(). По сути, это тот же самый код из той же библиотеки, просто загруженный под разными именами.

Копирование функции, которую античит-программа перехватывает, работает только если вы вызываете её на самом низкоуровневом уровне. Например, если этот код копировал бы kernel32.dll и динамически импортировал WriteProcessMemory(), руткит античата всё равно заблокировал бы бота, потому что kernel32\_copy.dll всё равно использует ntdll.NtWriteVirtualMemory() при вызове WriteProcessMemory().

## Обход эвристического анализа (Defeating Heuristics)

В дополнение ко всем продвинутым методам клиентской защиты, которые мы только что обсудили, игровые компании также используют эвристические методы на стороне сервера, отслеживая поведение игрока. Эти системы учатся различать человеческое поведение и поведение бота с помощью алгоритмов

машинного обучения. Их процесс принятия решений внутренний и сложный, поэтому трудно точно определить, какие особенности игрового процесса приводят к обнаружению.

Вам не нужно знать, как работают эти алгоритмы, чтобы их обмануть; вашему боту просто нужно вести себя как человек. Вот несколько распространённых признаков, по которым ботов можно отличить от людей:

Интервалы между действиями (Intervals between actions) Многие боты выполняют действия слишком быстро или с постоянными интервалами. Боты будут казаться более похожими на людей, если между их действиями будут случайные задержки. Также у них должна быть некоторая степень рандомизации, чтобы они не повторяли одно действие с постоянной скоростью.

Повторяющиеся маршруты (Path repetition) Боты, фармящие ресурсы, автоматически посещают запрограммированные точки, чтобы убивать существ. Эти "путевые точки" часто располагаются точно, указывая на каждое местоположение как чёткую координату. Люди же, напротив, двигаются менее предсказуемо и используют более разнообразные пути к знакомым местам. Чтобы воспроизвести человеческое поведение, бот должен двигаться в случайное место в пределах заданной зоны, а не точно в заданную точку. Также, если бот рандомизирует порядок посещения точек, разнообразие его путей значительно увеличится.

Нереалистичная игра (Unrealistic play) Некоторые боты остаются в одном месте сотни или тысячи часов, но человек не может играть так долго. Заставьте вашего бота делать паузы и предупреждайте пользователя, если он играет без перерыва более 8 часов. Кроме того, если бот повторяет одно и то же действие 7 дней подряд, он почти наверняка будет обнаружен в эвристической системе.

Совершенная точность (Perfect accuracy) Боты могут попадать в голову 1000 раз подряд без единого промаха, но даже профессиональный игрок не может делать это постоянно. Это практически невозможно для человека, поэтому умный бот должен быть намеренно неточным время от времени.

Это всего несколько примеров, но в целом вы можете избежать обнаружения просто используя здравый смысл. Не заставляйте бота делать что-то, что не может человек, и не позволяйте ему выполнять одно и то же действие слишком долго.

## Заключительные мысли (Closing Thoughts)

Разработчики игр и хакеры ведут постоянную битву. Хакеры находят новые способы обхода защиты, а разработчики ищут новые методы их обнаружения. Однако если вы достаточно внимательны, знания из этой главы помогут вам обойти любую античит-систему.

# Попасть в игру

ЛУЧШИЕ РАЗВЛЕЧЕНИЯ ДЛЯ ГИКОВ™  
[www.nostarch.com](http://www.nostarch.com)



ISBN:

1

A standard linear barcode representing the ISBN number.

6 89145 76699 8