قسمت ۱۱ – تجزیه و تحلیل کاربردی بدافزارها

راهنمای جامع مهندسی معکوس، تجزیه و تحلیل بدافزارها،
باج‌افزارها، جاسوس‌افزارها، روت‌کیت‌ها و بوتکیت‌های کامپیوترای

آزمایشگاه امنیت کی‌پاد

نویسنده: میلاد کهساری الهادی

فصل دهم: دیباگ در سطح کرنل با WinDBG

دیباگر WinDBG (که عموما Windbag خوانده می‌شود) یک دیباگر رایگان از شرکت مایکروسافت است. در حالی‌که این دیباگر مانند دیباگر OllyDBG در تجزیه و تحلیل بدافزارها مشهور نیست اما ویژگی‌های بسیاری دارد که از مهم‌ترین آن‌ها می‌توان به دیباگ کرنل اشاره کرد. در این فصل روش‌های مختلفی برای دیباگ کرنل و تجزیه و تحلیل روت‌کیت‌ها مورد کاوش قرار خواهد گرفت.

دیباگر WinDBG از دیباگ حالت کاربر پشتیبانی می‌کند و بیشتر اطلاعات در این فصل در حالت کاربر و کرنل قابل اجرا است، اما در این فصل ما فقط روی دیباگ حالت کرنل متمرکز خواهیم شد، زیرا بیشتر تحلیلگران بدافزار از دیباگر OllyDBG برای دیباگ برنامه‌ها در حالت کاربر استفاده می‌کنند. همچنین از دیگر قابلیت‌های دیباگر WinDBG می‌توان به نظارت تعاملات برنامه‌ها با ویندوز و دارا بودن فایل‌های راهنمای کمکی گسترده آن که شما را در یادگیری این ابزار یاری می‌نمایند، اشاره کرد.

کد کرنل و درایورها[1]

قبل از این‌که شروع به دیباگ کدهای مخرب کرنل کنیم، نیاز داریم بدانیم چگونه کدهای سطح کرنل کار می‌کنند، چرا نویسندگان بدافزار از آن‌ها استفاده کرده و همچنین در استفاده از این نوع کدها چه چالش‌هایی وجود دارند. دیوایس درایورهای[2] ویندوز یا به طور خلاصه درایورها، به برنامه‌نویسان یا توسعه‌دهندگان برنامه‌های ویندوزی اجازه می‌دهند، کدهایشان را در کرنل سامانه‌عامل ویندوز اجرا کنند.

تجزیه و تحلیل درایورها بسیار دشوار است، زیرا آن‌ها در حافظه بارگذاری می‌شوند و در آنجا باقی می‌مانند و به درخواست‌های برنامه‌های کاربردی از آنجا پاسخ می‌دهند. این موضوع بعدها پیچیده‌تر می‌شود، زیرا برنامه‌های کاربردی با درایورهای کرنل به صورت مستقیم تعامل ندارند. به جای آن، برنامه‌های کاربردی به Device Objectها دسترسی دارند که درخواست‌ها را به دیوایس‌های مشخصی ارسال می‌کنند. به این نکته توجه داشته باشید، لزوماً دیوایس‌ها به مولفه‌های سخت‌افزاری فیزیکی نیاز ندارند؛ یک درایور می‌تواند یک دیوایس (بدون یک مولفه فیزیکی) ایجاد کند یا از بین ببرد که از فضای کاربر قابل دسترس باشند.

---

[1] Drivers and Kernel Code
[2] Device Drivers

به عنوان مثال، یک درایو USB Flash را در نظر بگیرید. در این مثال، یک درایور روی سامانه عامل می‌تواند درایوهای USB Flash را کنترل کند، اما یک برنامه کاربردی نمی‌تواند به صورت مستقیم به آن درخواستی (Request) ارسال کند، بجای آن به یک آبجکت مشخص از دیوایس درخواست می‌دهد. به عبارت دیگر، هنگامی‌که کاربر USB Flash را به رایانه متصل می‌کند، ویندوز به صورت خودکار یک درایو (مانند F:\) برای آن ایجاد می‌کند که برنامه کاربردی می‌تواند درخواست‌های خود را از طریق آن به درایورهای USB Flash ارسال کند. شایان ذکر است، یک درایور مشابه ممکن است درخواست های یک برنامه برای یک USB Flash دیگر را به واسطه یک آبجکت درایو دیگر مانند G:\ کنترل کند.

همچنین برای این‌که سامانه عامل به درستی کار کند، همانند کتابخانه‌های پیوندی پویا که درون فرایندها بارگذاری می‌شوند، درایورها باید درون کرنل سامانه‌عامل بارگذاری شوند. هنگامی‌که یک درایور برای اولین بار بارگذاری می‌شود، مشابه تابع DLLMain در کتابخانه‌های پیوندی پویا، تابع DriverEntry درایور فراخوانی می‌شود.

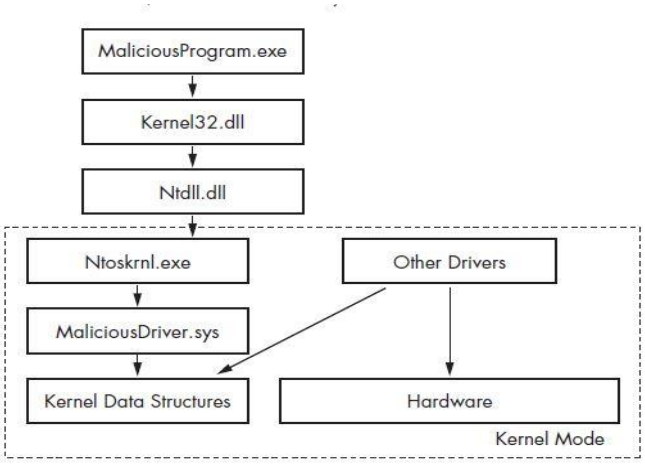
قابل ذکر است، برخلاف کتابخانه‌های پیوندی پویا که توابع خود را از طریق جدول خروجی یا Export Table نمایش می‌دهند، درایورها باید آدرس توابع callback را ثبت کنند که هنگام درخواست یک سرویس توسط یک مولفه نرم‌افزاری فراخوانی شوند. این فرایند ثبت در روتین DriverEntry درایورها رخ می‌دهد. ویندوز یک استراکچر Driver Object ایجاد می‌کند که به روتین DriverEntry عبور داده می‌شود. روتین DriverEntry مسئول پر کردن این استراکچر با توابع callback خودش است. سپس روتین DriverEntry یک دیوایس ایجاد می‌کند که می‌تواند از فضای کاربر در دسترس قرار گیرد و برنامه‌های فضای کاربر با ارسال درخواست به آن با درایور تعامل برقرار کنند.

نکته: در برنامه‌نویسی، یک تابع callback یک تابع است که به عنوان یک پارامتر به توابع دیگر عبور داده می‌شود که انتظار می رود در زمان مناسب اجرا شود. زبان‌های برنامه‌نویسی توابع callback را به روش‌های گوناگونی از قبیل سابروتین‌ها، عبارت‌های لامبادا، بلاک‌ها یا توابع اشاره‌گر پیاده‌سازی و اجرا می‌کنند.

به عنوان مثال، در این قسمت یک درخواست خواندن[1] از یک برنامه در فضای کاربر را در نظر بگیرید. این درخواست ابتدا به یک درایور عبور داده می‌شود که سخت‌افزاری را مدیریت می‌کند که روی آن داده‌ها (اطلاعات) به منظور خواندن ذخیره شده‌اند. برنامه در فضای کاربر باید ابتدا یک شناسه هندل[2] از این دیوایس به دست آورده و سپس تابع ReadFile را به همراه آن شناسه هندل فراخوانی کند. در ادامه کرنل درخواست ایجاد شده توسط تابع ReadFile را پردازش خواهد کرد و سپس تابع callback درایوری را که مسئول کنترل درخواست‌های ورودی و خروجی دیوایس است، اجرا می‌کند.

شایان ذکر است، رایج‌ترین درخواست برای یک مولفه مخرب که در کرنل رخ می‌دهد، درخواست DeviceIoControl می‌باشد که یک درخواست عمومی از سمت یک ماژول حالت کاربر به یک دیوایس تحت مدیریت یک درایور است. برنامه در حالت کاربر یک بافر از داده‌ها با طول دلخواه به عنوان ورودی ارسال کرده و سپس یک بافر با طول دلخواه را به عنوان خروجی دریافت می‌کند.

رهگیری فراخوانی‌های یک برنامه حالت کاربر به یک درایور در حالت کرنل بسیار دشوار است، زیرا تمامی کدهای سامانه‌عامل از این فراخوانی‌ها پشتیانی می‌کنند. تصویر ۱ نشان می‌دهد که چگونه یک درخواست از برنامه سطح کاربر به یک درایور سطح کرنل دسترسی می‌یابد. ارسال درخواست از یک برنامه سطح کاربر آغاز شده و در نهایت به کرنل می‌رسد. برخی از درخواست‌های ارسالی به درایورها سخت‌افزار را کنترل می‌کنند و برخی دیگر تنها وضعیت داخلی کرنل را تحت تاثیر قرار می‌دهند.



تصویر ۱: چگونه یک فراخوانی سطح کاربر توسط کرنل کنترل می‌شود

[1] Read Request
[2] Handle ID

درایورهای آلوده معمولا سخت‌افزار را کنترل نمی‌کنند بلکه با مولفه‌های اصلی کرنل ویندوز (ntoskrnl.exe و hal.dll) تعامل می‌کنند. مولفه ntoskrnl.exe شامل کدهایی برای قابلیت‌های اصلی سامانه‌عامل و مولفه hal.dll شامل کدهایی برای تعامل با سخت‌افزار است. بدافزارها اغلب توابعی را از این دو فایل برای دستکاری فضای کرنل فراخوانی می‌کنند.

آماده سازی محیط برای دیباگ کرنل

دیباگ در حالت کرنل بسیار پیچیده تر از دیباگ در حالت کاربر است، زیرا هنگام دیباگ کرنل، سامانه‌عامل متوقف شده و به موجب آن اجرای یک دیباگر غیر ممکن می‌شود. بنابراین رایج‌ترین راه ممکن برای دیباگ کرنل استفاده از نرم‌افزار VMware است. برخلاف دیباگ در سطح کاربر، دیباگ کرنل نیازمند آماده‌سازی شرایط خاصی می‌باشد که در ادامه آن را مورد بررسی قرار خواهیم داد.

پیکربندی فایل Boot.ini

به منظور دیباگ کرنل، ابتدا باید یک ماشین‌مجازی را به منظور دیباگ کرنل سامانه‌عامل ایجاد کرده و سپس یک پورت سریال مجازی بین ماشین‌مجازی (ماشینی که قرار است دیباگ شود) و ماشین میزبان (ماشینی که قرار است روی آن دیباگر اجرا شود) در VMware راه‌اندازی کنید.

در نهایت دیباگر WinDBG را در ماشین میزبان اجرا کرده و مورد استفاده قرار دهید. همچنین برای آماده سازی ماشین‌مجازی، باید در فایل C:\boot.ini ویندوز تغییراتی ایجاد کنید (برای مشاهده و ویرایش این فایل در سامانه‌عامل ویندوز، در گام اول مطمئن شوید تنظیمات پوشه‌های سامانه روی نمایش تمامی فایل‌های سامانه‌ای تنظیم شده باشد.).

همچنین پیش از آغاز عملیات ویرایش boot.ini یک snapshot از ماشین‌مجازی بگیرید تا در صورتی که در ادامه راه اشتباهی کردید، بتوانید با استفاده از snapshot به حالت اولیه سامانه‌عامل بازگردید. لیست ۱ محتویات فایل boot.ini را در حالتی که دارای یک خط اضافه برای فعال کردن دیباگ کرنل می‌باشد، نشان می‌دهد.

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
❶ multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
/noexecute=optin /fastdetect
❷ multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional with Kernel
Debugging" /noexecute=optin /fastdetect /debug /debugport=COM1 /baudrate=115200
```

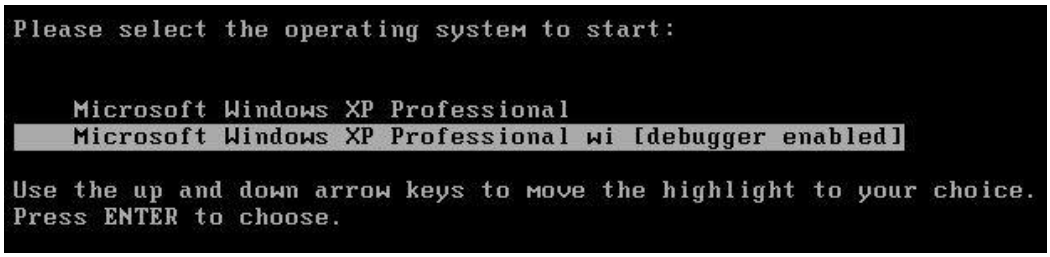
لیست ۱: یک فایل نمونه boot.ini که برای عملیات دیباگ کرنل ویرایش شده است.

در لیست ۱ (شماره ۱) سامانه‌عاملی که باید برای عملیت دیباگ بارگذاری شود، مشخص شده است. سپس در خطوط بعدی (شماره ۲) گزینه دیباگ کرنل سامانه‌عامل توسط ما فعال شده است. قابل ذکر است، نسخه تغییر نیافته boot.ini تنها حاوی خطوط شماره ۱ می‌باشد و شما باید خطوط آورده شده در قسمت شماره ۲ لیست ۱ را خود به این فایل اضافه کنید تا قابلیت دیباگ کرنل سامانه‌عامل برقرار شود.

همانطور که در قسمت شماره ۲ لیست ۱ مشاهده می‌کنید، آیتم /debug حالت دیباگ کرنل سامانه‌عامل را فعال می‌کند، گزینه /debugport=COM1 به سامانه‌عامل می‌گوید کدام پورت ماشین دیباگ شونده[1] را به ماشین دیباگ کننده[2] متصل کند و در نهایت گزینه baudrate=115200 سرعت ارتباط را مشخص می‌کند (Baud Rate یعنی تعداد سیگنالی که در واحد زمان می‌توان ارسال کرد).

در مثالی که ما در این قسمت آورده‌ایم، از پورت COM مجازی که توسط VMware ایجاد شده است، استفاده می‌کنیم. همچنین شما باید گزینه Boot ویندوز (شماره ۲) را که در boot.ini اضافه کرده‌اید با نامی متفاوت از نام اصلی ویندوز تعریف کنید. در این مثال ما آن را Windows XP Professional with Kernel Debugging معرفی کرده‌ایم. در مرحله بعدی ماشین‌مجازی را روشن کرده تا سامانه‌عامل شروع به راه‌اندازی شود، در قسمت Boot گزینه راه‌اندازی با عنوان ذکر شده (Windows XP Professional with Kernel Debugging) را خواهید دید که به منظور دیباگ کرنل سامانه‌عامل باید آن را انتخاب کنید. چون در آن قابلیت دیباگ کرنل سامانه‌عامل فعال شده است.

[1] Debugee Machine
[2] Debugger Machine

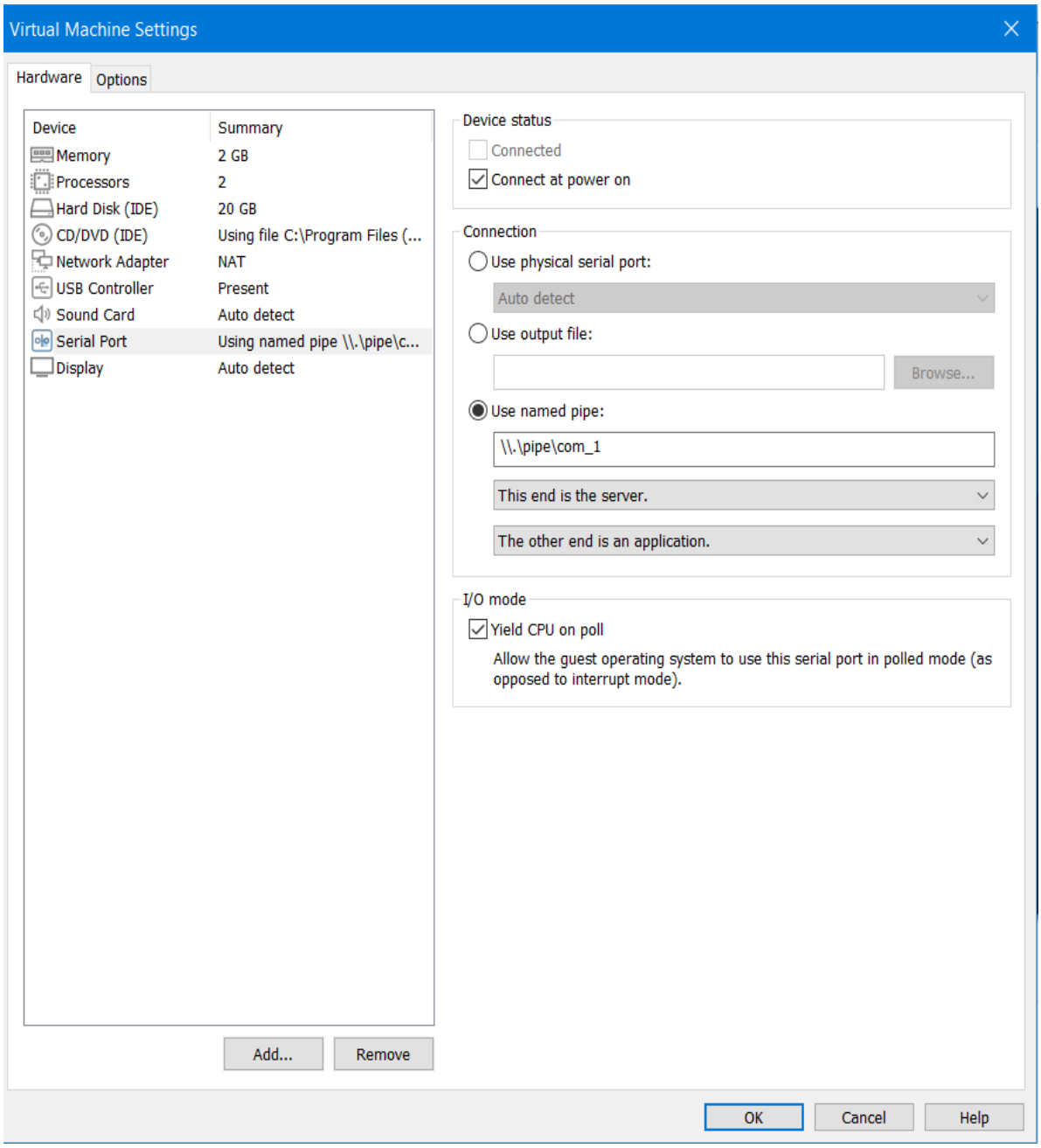


تصویر ۲: انتخاب گزینه دیباگ ویندوز

نکته: راه‌اندازی سامانه‌عامل با وضعیت دیباگ به این معنا نیست که شما حتما باید دیباگری را به آن پیوست کنید. سامانه‌عامل زمانی‌که دیباگری به آن وصل نشود به صورت عادی کار می‌کند و اگر به آن دیباگر WinDBG را متصل سازید به حالت دیباگ می‌رود.

سپس نرم‌افزار VMware را برای ایجاد یک ارتباط مجازی بین سامانه‌عامل میزبان و ماشین‌مجازی تنظیم می‌کنیم. برای این کار یک پورت سریال در بخش "Named Pipe" به ماشین میزبان اضافه می‌کنیم. مراحل زیر را برای افزودن این سخت‌افزار مجازی انجام دهید.

۱. در نرم‌افزار VMWare روی منوی VM کلیک کرده و گزینه Settings را انتخاب کنید تا پنجره تنظیمات VMware باز شود.
۲. در پنجره تنظیمات روی دکمه Add کلیک کرده و گزینه Serial Port را انتخاب کنید.
۳. در پنجره‌ای که نوع پورت سریال را درخواست می‌کند گزینه "Output to Named Pipe" را انتخاب کنید.
۴. در پنجره بعدی مقدار \\.\pipe\com_1 را به عنوان نام سوکت وارد کرده و از منوهای بازشو گزینه‌های This end is the server و The other end is an application را انتخاب کرده و روی Finish کلیک کنید.



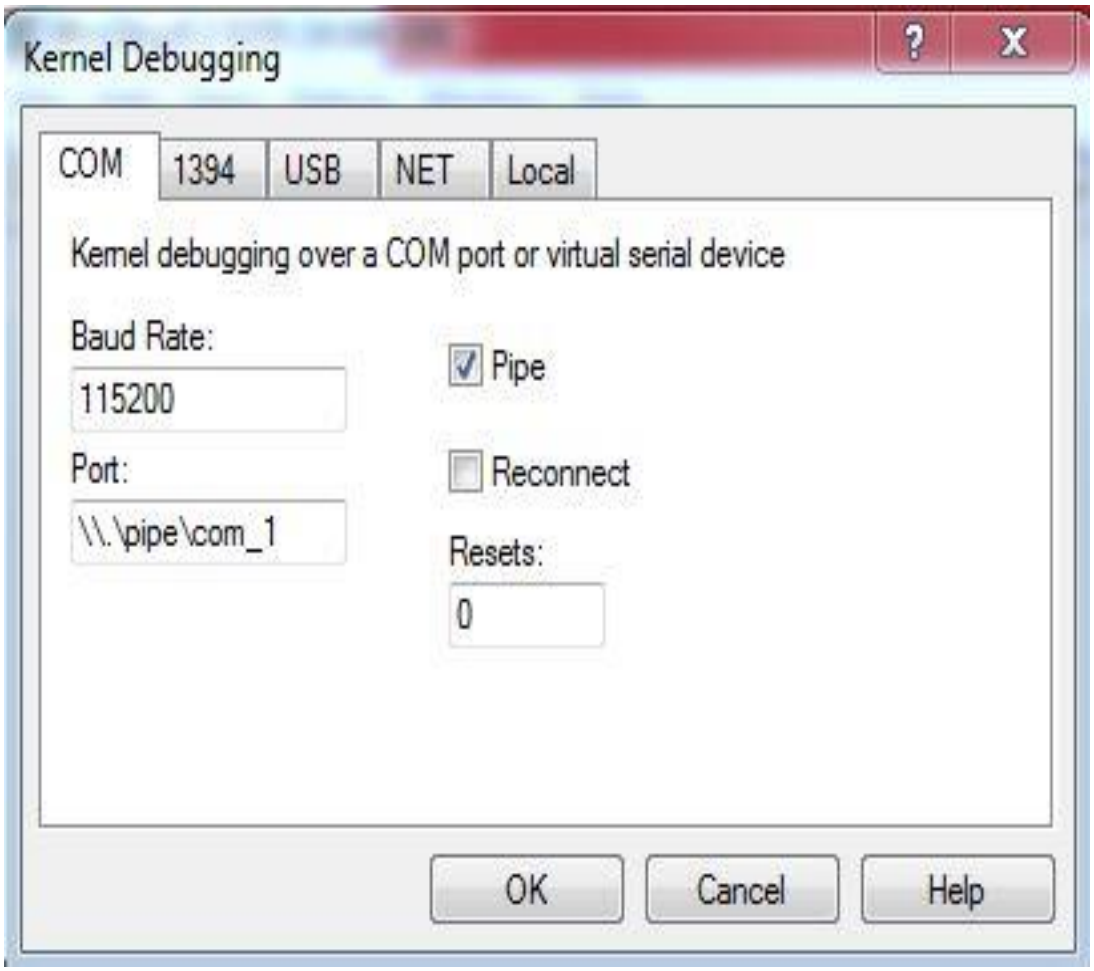
تصویر ۳: تنظیمات محیط **VMWare** برای دیباگ کرنل

۵. زمانی‌که افزودن پورت سریال را به پایان رساندید، ماشین‌مجازی یک پورت سریال را به لیست سخت‌افزارهای مجازی اضافه می‌کند. دقت کنید، زمانی‌که پورت مجازی سریال را در لیست سخت‌افزارهای مجازی انتخاب می‌کنید باید کادر علامت "Yield CPU on poll" فعال شده باشد (مانند تصویر ۳).

پس از تنظیم ماشین‌مجازی آن را راه‌اندازی کنید. سپس مراحل زیر را در ماشین میزبان طی کنید تا بتوانید ابزار WinDBG را برای ارتباط با ماشین‌مجازی و آغاز عملیات دیباگ کرنل پیکربندی کنید.

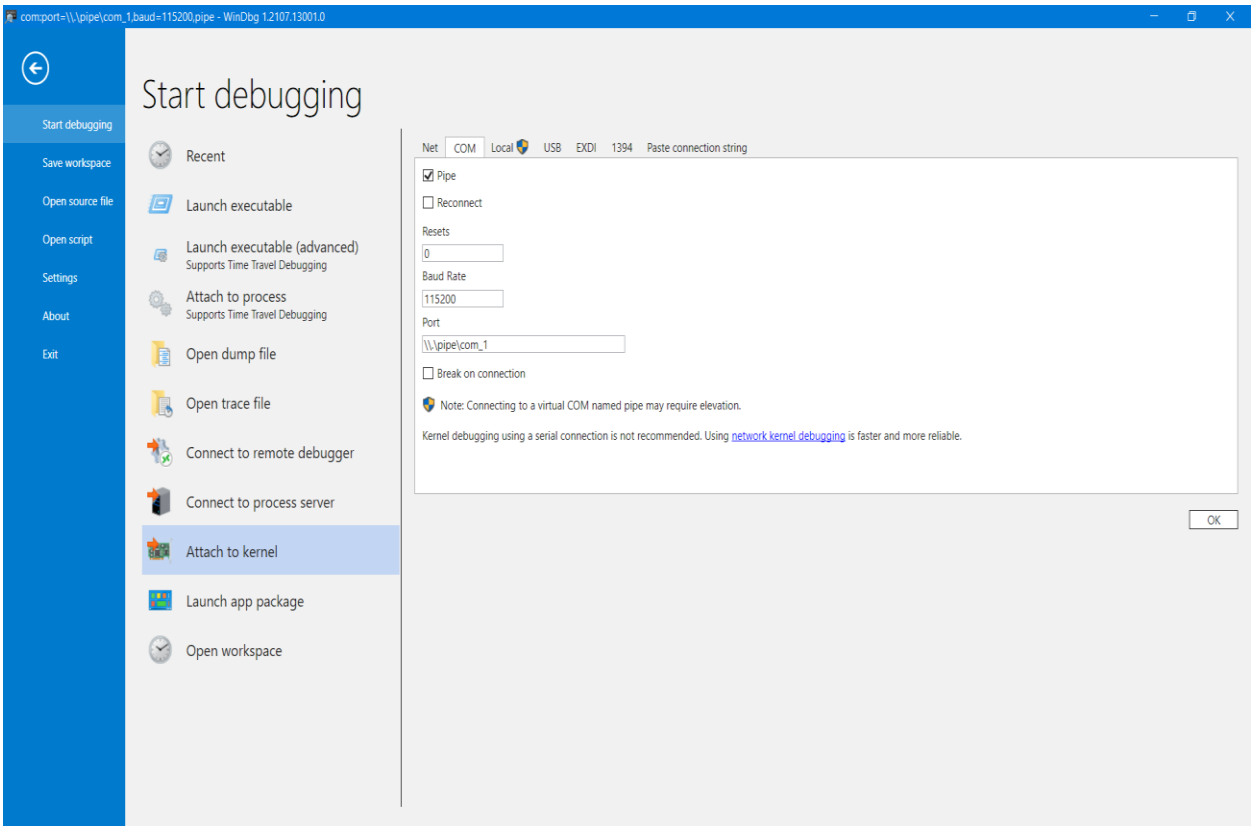
۱. ابتدا ابزار WinDBG را اجرا کنید.

۲. از منوی File گزینه Kernel Debug را انتخاب کرده و در نهایت زبانه COM را انتخاب کنید. در قسمت Baud Rate مقدار آن را روی مقدار ۱۱۵۲۰۰ تنظیم کنید (اگر دقت کرده باشید؛ این مقدار را پیش از این در ماشین‌مجازی در فایل boot.ini تنظیم کرده بودیم) در نهایت اطمینان حاصل کنید که کادر علامت Pipe در این صفحه فعال شده باشد، سپس روی دکمه OK کلیک کنید. پنجره شما باید مانند تصویر ۴ باشد.



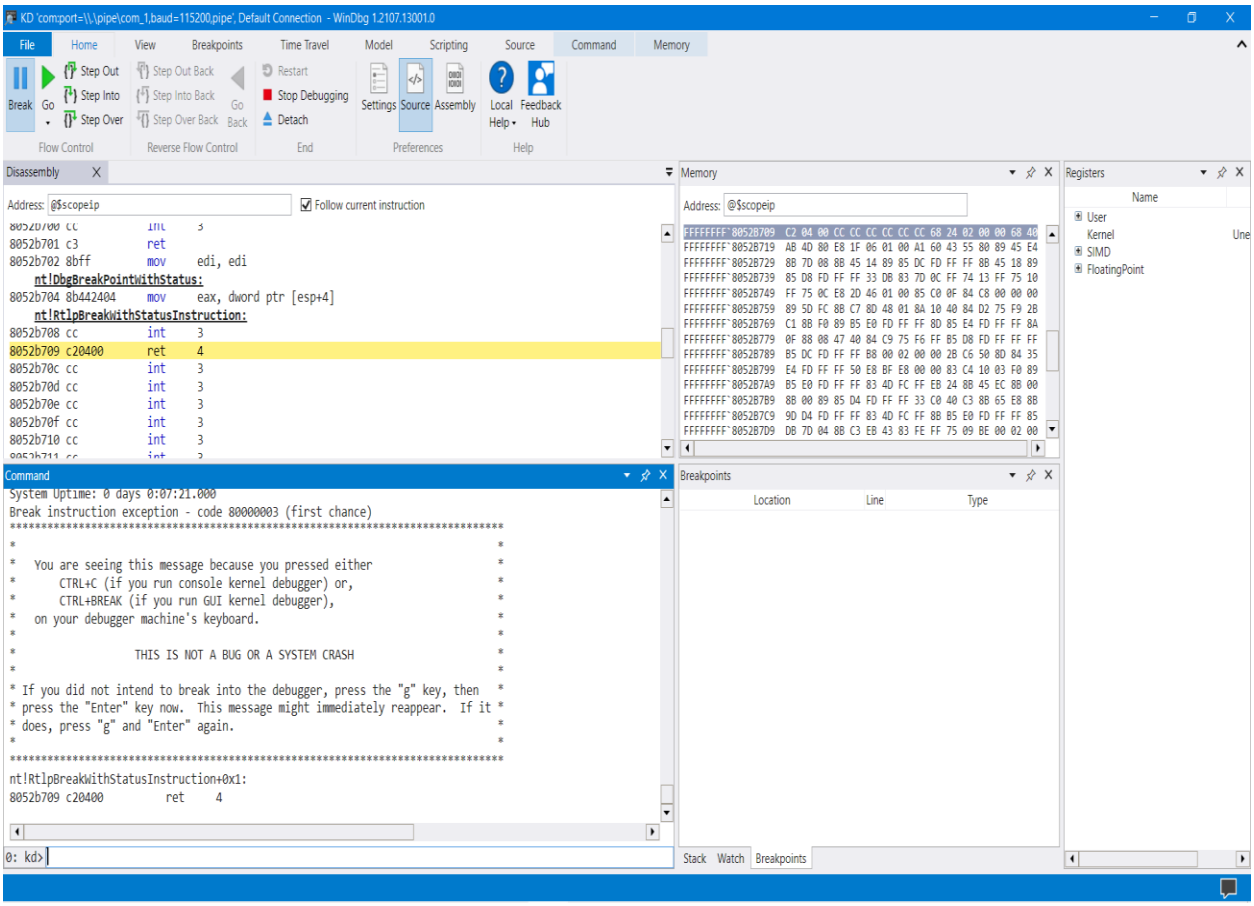
تصویر ۴: آغاز عملیات دیباگ کرنل در ابزار WinDBG

شایان ذکر است، اگر از نسخه Windbg Preview استفاده می‌کنید، می‌توانید از قسمت تنظیمات و انتخاب گزینه Attach to Kernel این اطلاعات را در زبانه COM وارد کنید (تصویر ۵)، و در نهایت با کلیک بر روی OK به ماشین مجازی برای دیباگ کرنل متصل شوید. نسخه Windbg Preview دارای یک سری تفاوت‌ها در ظاهر و همچنین ویژگی‌ها عملیاتی این دیباگر است از قبیل Time Travel Debugging، که به مرور در این قسمت یا قسمت‌های آینده درباره آن بحث خواهیم کرد.



تصویر ۵: محیط دیباگر **Windbg Preview** برای دیباگ کرنل

در صورتی که ماشین‌مجازی در حال اجرا باشد، دیباگر در عرض چند ثانیه به ماشین‌مجازی وصل خواهد شد. اما اگر ماشین‌مجازی را فعال نکرده باشید، دیباگر تا زمان راه‌اندازی سامانه‌عامل مجازی منتظر می‌ماند و سپس به آن متصل می‌شود. در تصویر ۶، محیط دیباگر Windbg Preview بعد از اتصال به ماشین مجازی برای دیباگ کرنل را مشاهده می‌کنید.



تصویر ٦: محیط دیباگر **Windbg Preview** هنگام دیباگ کرنل

زمانی‌که دیباگر به ماشین‌مجازی متصل شد، وضعیت خروجی "Verbose" را فعال کنید تا بتوانید شکل بهتری از وضعیت موجود داشته باشید. با فعال بودن خروجی Verbose هر زمان که درایوری بارگذاری یا از حالت بارگذاری خارج شود، متوجه خواهید شد. این ویژگی در بعضی موارد می‌تواند به شما برای تشخیص یک درایور مخرب کمک کند.

استفاده از WinDBG

ابزار WinDBG از یک رابط کاربری تحت خط فرمان برای بیشتر قابلیت‌های خودش استفاده می‌کند. ما در اینجا مهم‌ترین فرامین آن را مورد بررسی قرار خواهیم داد. همچنین شما می‌توانید لیست کاملی از فرامین این ابزار را در منوی Help آن مشاهده کنید.

خواندن حافظه[1]

پنجره حافظه دیباگر WinDBG از قابلیت مشاهده مستقیم حافظه به وسیله خط فرمان برخوردار است. فرمان d برای خواندن آدرس‌های مختلف حافظه مانند بخش مرتبط با داده‌های برنامه یا پشته برنامه است که به صورت زیر به کار می‌رود.

dx addressToRead

در فرمان بالا، حرف x یکی از راه‌های درخواست از WinDBG به منظور نمایش داده‌ها در فرمت‌های گوناگون است. جدول ۱ رایج‌ترین راه‌های نمایش داده‌ها را نشان می‌دهد.

جدول ۱: گزینه‌های خواندن WinDBG

گزینه	تشریح
da	این گزینه داده‌های درون حافظه را به صورت متن ASCII نمایش می‌دهد.
du	این گزینه داده‌های درون حافظه را به صورت UNCODE نمایش می‌دهد.
dd	این گزینه داده‌های درون حافظه را به صورت یک DWORD نمایش می‌دهد.

به عنوان مثال برای نمایش یک رشته در آدرس 0x401020 باید از فرمان da 0x401020 استفاده کنید. در این موقعیت به این دلیل از گزینه a استفاده می‌کنیم، زیرا گزینه a داده‌های درون حافظه را به صورت متن ASCII نمایش می‌دهد. همچنین ما می‌توانیم از فرمان e برای تغییر محتویات حافظه (نه فقط نمایش آن) استفاده کنیم که ساختار آن در زیر آورده شده است.

ex addressToWrite dataToWrite

فرمان ex در اینجا نیز مشابه فرمان dx است. علاوه بر این، می‌توانید با رجوع به راهنمای این برنامه جزییات بسیار بیشتری درباره این فرمان به‌دست آورید.

[1] Reading from Memory

استفاده از عملوندهای محاسباتی[1]

علاوه بر نکات بالا، می‌توانید عملیات‌های مختلفی را روی حافظه یا رجیسترها با استفاده از عملوندهای محاسباتی مانند جمع (+)، تفریق (-)، ضرب (*) و تقسیم (/) انجام دهید. گزینه‌های خط فرمان در زمان ایجاد یک عبارت شرطی برای درج یک نقطه توقف می‌توانند به عنوان یک میانبر بسیار مفید باشند.

فرمان dwo برای دی‌رفرنس[2] ساختن یک اشاره‌گر ۳۲ بیتی و مشاهده مقدار آن استفاده می‌شود. به عنوان مثال، اگر در ابتدای یک تابع نقطه توقفی قرار بدهید و اولین پارامتر ورودی آن تابع یک رشته متنی عریض باشد، می‌توانید این رشته را با استفاده از فرمان زیر مشاهده کنید.

```
du dwo (esp+4)
```

گزینه esp+4 آدرس اولین پارامتر ورودی به تابع است (به منظور درک این نکته، شما باید برنامه‌نویسی اسمبلی را به خوبی یاد داشته باشید و قسمت‌های گذشته را به خوبی درک کرده باشید). عملوند dwo محل اشاره‌گر به آن رشته را تشخیص داده و پارامتر du به دیباگر WinDBG می‌گوید که رشته متنی مورد نظر در آن آدرس را نمایش دهد.

تنظیم نقطه توقف[3]

در دیباگر WinDBG از فرمان bp برای ایجاد یک نقطه توقف ساده استفاده می‌شود. همچنین شما می‌توانید فرامینی را مشخص سازید تا هنگامی‌که توقفی رخ داد، قبل از این‌که کنترل برنامه به کاربر ارائه شود آن فرامین به صورت خودکار اجرا شوند. این ویژگی با فرمان go (g) مورد استفاده قرار می‌گیرد، به این صورت که نقطه توقف یک عملیات خاص را انجام داده و سپس بدون انتظار برای دریافت دستور ادامه اجرا از سوی فرد دیباگ کننده به اجرا ادامه می‌دهد. به عنوان مثال، فرمان زیر هر زمان که تابع GetProcAddress فراخوانی شود، پارامتر دوم ورودی به تابع را نمایش می‌دهد، بدون این‌که واقعا اجرای برنامه را متوقف سازد.

```
bp GetProcAddress "da dwo(esp+8); g"
```

[1] Using Arithmetic Operators
[2] Dereference
[3] Setting Breakpoints

این مثال نام تمامی توابع درخواست داده شده در هر فراخوانی GetProcAddress را چاپ خواهد کرد. این ویژگی بسیار مفید است زیرا نقط توقف در این شرایط با سرعت بیشتری در مقایسه با ارائه کنترل به کاربر اجرا می‌شوند. همچنین شایان ذکر است، فرامین رشته‌ای با پشتیبانی از عبارات شرطی مانند عبارات if. و حلقه‌های while. می‌توانند بسیار پیچیده‌تر باشند، به همین دلیل دیباگر WinDBG به منظور تسهیل روند شرط‌گذاری از اسکریپت‌هایی استفاده می‌کند که این فرامین را مورد پشتیبانی قرار می‌دهند.

نکته: فرامین گاهی اوقات تلاش می‌کنند به آدرس‌هایی از حافظه دسترسی پیدا کنند که وجود ندارند. به عنوان مثال، دومین پارامتر به تابع GetProcAddress می‌تواند یک رشته و یا یک عدد ترتیبی[1] باشد. در صورتی که مقدار ورودی یک عدد باشد، دیباگر WinDBG تلاش خواهد کرد داده موجود در آدرسی را که آن عدد به آن اشاره می‌کند، به‌دست آورد. شایان ذکر است، در صورتی که آدرس موجود نباشد، اگر خوش شانس باشید دیباگر WinDBG خراب نخواهد شد یا به عبارت دیگر Crash نخواهد کرد و تنها مقدار ؟؟؟؟ را نمایش می‌دهد.

در قسمت زیر یک نمونه اسکریپت WinDBG نمایش داده شده است که از آن می‌توان در دیباگر WinDBG به منظور بررسی روتین KdDebuggerEnabled استفاده کرد. البته در این کتاب قصد نداریم چگونگی نوشتن این نوع اسکریپت‌ها را آموزش دهیم، اما به منظور آشنایی با اسکریپت‌های WinDBG نمونه زیر در این قسمت آورده شده است.

```
$$
$$ A sample windbg script for Control of KdDebuggerEnabled
$$ Written by Milad Kahsari Alhadi
$$ http://www.aiooo.ir
$$

r $t0 = nt!KdDebuggerEnabled;

.if (poi(@$t0) == 1)
{
```

---

[1] Ordinal Number

```
    .printf "\\nThe debugger is enabled...\\n";

    $$
    $$ Here I'm displaying the KdDebuggerEnabled byte.
    $$
    .printf /D "      nt!KdDebuggerEnabled <b>";
    db nt!KdDebuggerEnabled nt!KdDebuggerEnabled
    .printf /D "</b>\\n";

    $$
    $$ Here with this command I'm printing a link command!
    $$
    .printf /D "<link cmd=\"eb @$t0 0x00\"> Show me your talent!</link>\\n";
}
.else
{
    .printf /D "\\nThe debugger is hidden. \\n";

    $$
    $$ Here with this command I'm printing a link command!
    $$
    .printf /D "      nt!KdDebuggerEnabled <b>";
    db nt!KdDebuggerEnabled nt!KdDebuggerEnabled
    .printf /D "</b>\\n";

    $$
    $$ Here with this command I'm printing a link command!
    $$
    .printf /D "<link cmd=\"eb @$t0 0x01\">Oh Please reset the debug flag!</link>\\n";
}
```

لیست ۲: اسکریپت نمونه برای Windbg

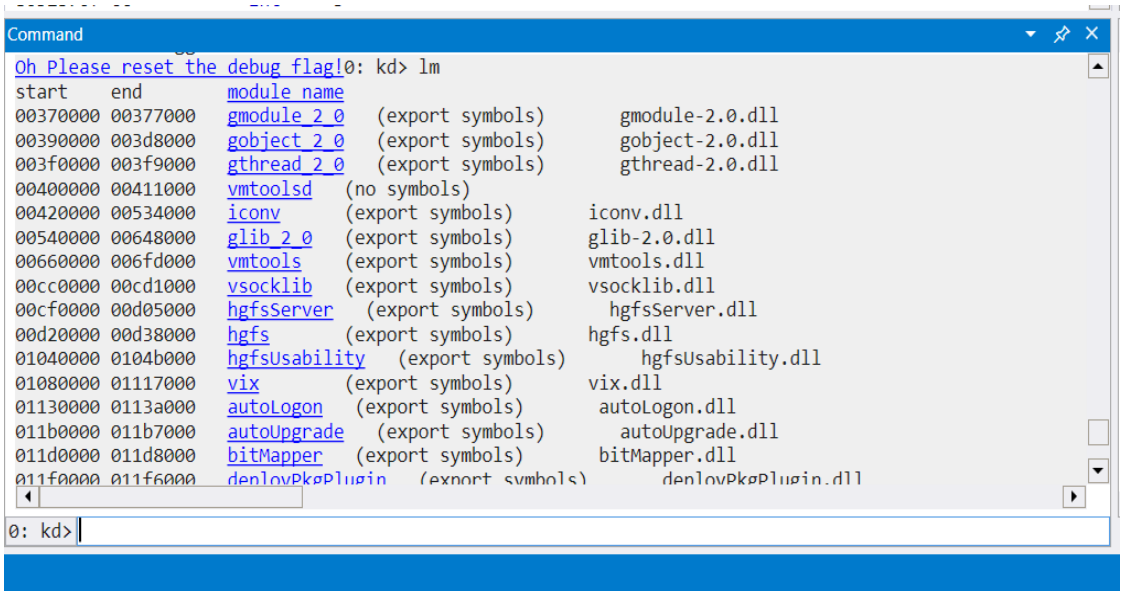
سپس این اسکریپت را با یک نام نمونه (به عنوان مثال، clightning.wds) ذخیره کنید. سپس وارد محیط دیباگر WinDBG شده و با اجرای فرمان زیر، این اسکریپت را در محیط دیباگر WinDBG اجرا کنید و خروجی آن را مورد بررسی قرار بدهید.

```
kd> $$><c:\clightning.wds

The debugger is hidden.
```

nt!KdDebuggerEnabled 8054c6c1 01.
Oh Please reset the debug flag!

لیست کردن ماژول‌ها

دیباگر WinDBG مانند دیباگر OllyDBG ویژگی برای نمایش نقشه، لایه‌ها و بخش‌های حافظه و همچنین ماژول‌های بارگذاری شده دارا نمی‌باشد. به جای آن، دیباگر WinDBG دارای فرمان lm است که تمام ماژول‌های بارگذاری شده در پروسه از جمله کدهای اجرایی، کتابخانه‌های پیوندی پویا در سطح کاربر و درایورهای سطح کرنل را نشان می‌دهد. همچنین آدرس‌های آغازین و انتهایی هر ماژول نیز با این فرمان نمایش داده خواهد شد. در تصویر ۷، خروجی دستور lm در محیط دیباگر Windbg Preview را مشاهده می‌کنید که رابط کاربری جامع‌تری نسبت به Windbg دارد.



تصویر ۷: خروجی دستور lm در محیط Windbg

سمبول‌های مایکروسافت[1]

سمبول‌های دیباگ مایکروسافت اطلاعات محدودی برای درک بهتر کدهای دیزاسمبلی ارائه می‌دهند. سمبول‌هایی که مایکروسافت در اختیار ما قرار می‌دهد شامل اسامی توابع و متغیرهای آن‌ها است.

[1] Microsoft Symbols

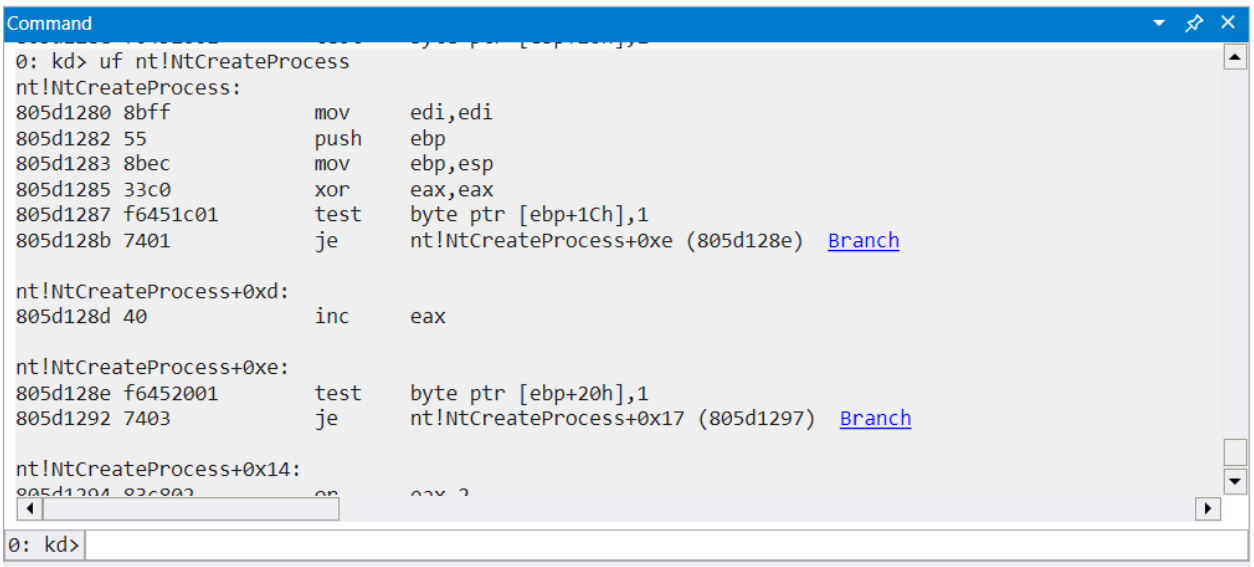
یک سمبول در اینجا به مفهوم یک نام برای یک آدرس از حافظه است. به عنوان مثال بدون داشتن اطلاعات سمبول‌های مایکروسافت، یک تابع در آدرس 8050f1a2 برچسب نخواهد خورد (به این معنا که نامی برای آن آدرس در دیباگر مشخص نخواهد شد) در صورتی‌که اگر تنظیمات لازم برای دریافت اطلاعات سمبول‌های مایکروسافت موجود باشد، آنگاه دیباگر WinDBG می‌تواند بگوید که در آدرس f1a28050 تابع MmCreateProcessAddressSpace وجود دارد اما اگر سمبول‌های مایکروسافت وجود نداشته باشند، ما فقط یک آدرس حافظه را مشاهده خواهیم کرد که از آن نمی‌توانیم چیز زیادی متوجه شویم. ولی با سمبول‌های مایکروسافت می‌توانیم متوجه شویم که تابع MmCreateProcessAddressSpace در آدرس 8050f1a2 به منظور ایجاد یک فضای آدرس برای یک پروسه استفاده می‌شود. همچنین از نام سمبول‌ها می‌توانید برای شناسایی یک تابع و یا داده‌هایی در حافظه استفاده کنید.

جستجو سمبول‌ها

ساختار مورد نیاز برای ارجاع به یک سمبول در WinDBG به صورت زیر است :

moduleName!symbolName

در هر موقعیت دارای یک آدرس این Syntax می‌تواند مورد استفاده کرد. پارامتر moduleName نام یک ماژول در فرمت exe، dll یا sys است که توابع استفاده شده در پروسه را شامل می‌شوند، البته بدون پسوند است و پارامتر symbolName نامی است که در آن آدرس معرفی شده است. یک حالت خاص نیز در مورد این موضوع وجود دارد و آن فایلntoskrnl.exe است که نام ماژول بجای ntoskrnl باید تنها nt باشد. به عنوان مثال، اگر می‌خواهید به دیزاسمبلی تابع NtCreateProcess نگاه کنید، باید از دستور "u" (به معنای unassemble) با پارامتر nt!NtCreateProcess به شکل (u nt!NtCreateProcess) استفاده کنید. شایان ذکر است، برای اینکه کل بدنه تابع دیزاسمبل و در خروجی به نمایش داده شود، از دستور uf باید استفاده کنیم. به عنوان مثال، در تصویر ۸، خروجی دیزاسمبلی تابع NtCreateProcess به صورت کامل نمایش داده شده است.



تصویر ۸: دیزاسمبلی تابع **NtCreateProcess**

فرمان "bu" به شما اجازه می‌دهد تا از سمبول برای ایجاد یک نقطه توقف در کدی که هنوز بارگذاری نشده است، استفاده کنید (به این نوع نقطه توقف، نقطه توقف تخیری[1] گفته می‌شود.) نقطه توقف تخیری نقطه توقفی است که هنگام بارگذاری یک ماژول از پیش تعیین ایجاد می‌شود. به عنوان مثال، فرمان bu newModule!exportedFunction به WinDBG می‌گوید در تابع exportedFunction به محض بارگذاری یک ماژول با نام newModule یک نقطه توقف ایجاد کند. در صورتی که چنین ماژولی بارگذاری شود، این اتفاق خواهد افتاد در غیر این صورت برنامه به روند عادی خود ادامه خواهد داد.

ترکیب فرمان bu با فرمان $iment(driverName) که نقطه ورودی یک درایور را مشخص می‌کند، هنگام بررسی ماژول‌های کرنل می‌تواند مفید باشد. فرمان bu $iment(driverName) یک نقطه توقف روی نقطه ورودی یک درایور پیش از آنکه کدی از آن شانسی برای اجرا داشته باشند ایجاد خواهد کرد. به این نکته توجه کنید، به منظور ایجاد یک نقطه توقف در نقطه ورودی یک درایور با استفاده از این فرمان، باید به جای عبارت driverName، نام درایوری که قصد دارید روی آن عملیات مذکور را انجام دهید، وارد کنید. به عنوان مثال به منظور قرار دادن یک نقطه توقف در نقطه ورودی درایور Hardware Policy Driver باید دستور bu $iment(hwpolicy) را اجرا کنید.

[1] Deferred Breakpoint

در دیباگر WinDBG همچنین یک فرمان x وجود دارد که به شما اجازه می‌دهد نام یک تابع یا سمبول خاص را با استفاده از ویژگی wildcard جستجو کنید. wildcard به این معنا است که اگر شما دنبال جستجوی تابعی هستید که فقط بخشی از نام آن را می‌دانید، می‌توانید با افزودن دو ستاره به ابتدا و انتهای آن نام به سادگی آن را جستجو کنید، به این ویژگی wildcard می‌گویند که در ادامه مورد بررسی قرار خواهیم داد. به عنوان مثال اگر به دنبال یک تابع در کرنل هستید که کارش ایجاد پروسه است، می‌توانید با جستجوی کلمه CreateProcess در فایل ntoskrnl.exe به دنبال آن بگردید. فرمان مربوط به صورت x nt!*CreateProcess* خواهد بود که توابع اکسپورت شده و ایمپورت شده‌ای که شامل کلمه CreateProcess هستند را در خروجی به شما نمایش خواهد داد. خروجی زیر برای این فرمان می‌باشد.

```
0:003> x nt!*CreateProcess*
805c736a nt!NtCreateProcessEx = <no type information>
805c7420 nt!NtCreateProcess = <no type information>
805c6a8c nt!PspCreateProcess = <no type information>
804fe144 nt!ZwCreateProcess = <no type information>
804fe158 nt!ZwCreateProcessEx = <no type information>
8055a300 nt!PspCreateProcessNotifyRoutineCount = <no type information>
805c5e0a nt!PsSetCreateProcessNotifyRoutine = <no type information>
8050f1a2 nt!MmCreateProcessAddressSpace = <no type information>
8055a2e0 nt!PspCreateProcessNotifyRoutine = <no type information>
```

لیست ۳: خروجی دستور x

فرمان مفید دیگر ln است که نزدیکترین سمبول‌ها را به یک آدرس حافظه مشخص شده نشان می‌دهد. از این ویژگی می‌توان برای شناسایی این‌که یک اشاره‌گر به چه تابعی اشاره می‌کند، استفاده کرد. به عنوان مثال، فرض کنید یک فراخوانی تابع در آدرس 0x805717aa مشاهده کردیم و قصد داریم متوجه شویم دلیل استفاده از این کد در آن آدرس چیست. برای این کار می‌توان از فرمان زیر استفاده کرد.

```
0:002> ln 805717aa
kd> ln ntreadfile
❶ (805717aa)   nt!NtReadFile   |  (80571d38)   nt!NtReadFileScatter
Exact matches:
❷     nt!NtReadFile = <no type information>
```

لیست ۴: خروجی دستور ln در Windbg

در لیست ۴ (شماره ۱) دو سمبول تقریبا منطبق و خط آخر (شماره ۲) یک سمبول کاملا منطبق بر آدرس مورد نظر را نشان می‌دهد. درصورتی‌که گزینه‌ای کاملا منطبق وجود نداشته باشد، تنها خط اول نمایش داده می‌شود.

مشاهده اطلاعات استراکچر[1]

سمبول‌های مایکروسافت همچنین شامل اطلاعات نوع داده[2] بسیاری از استراکچرها هستند از جمله مواردی که توسط مایکروسافت مستندسازی نشده‌اند. این ویژگی برای تحلیلگر بدافزار می‌تواند مفید باشد، زیرا بدافزارها اصولا سعی در دستکاری استراکچرهای مستندسازی نشده دارند. لیست ۵ خطوط اولیه یک استراکچر برای آبجکت درایور[3] را نشان می‌دهد که اطلاعات درباره یک درایور کرنل را ذخیره می‌کند.

```
0:000> dt nt!_DRIVER_OBJECT
kd> dt nt!_DRIVER_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 Flags            : Uint4B
1  +0x00c DriverStart      : Ptr32 Void
   +0x010 DriverSize       : Uint4B
   +0x014 DriverSection    : Ptr32 Void
   +0x018 DriverExtension  : Ptr32 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING
   +0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
   +0x028 FastIoDispatch   : Ptr32 _FAST_IO_DISPATCH
   +0x02c DriverInit       : Ptr32     long
   +0x030 DriverStartIo    : Ptr32     void
   +0x034 DriverUnload     : Ptr32     void
   +0x038 MajorFunction    : [28] Ptr32     long
```

لیست ۵: مشاهده نوع اطلاعات برای یک ساختار

نام استراکچرها در تشخیص نوع داده‌ای که در آن‌ها ذخیره شده است، یک امتیاز به ما ارائه می‌دهد. به عنوان مثال، در آفست 0x00c (شماره ۱) یک اشاره‌گر وجود دارد که مشخص می‌کند یک درایور در کجای حافظه بارگذاری می‌شود.

دیباگر WinDBG به شما اجازه می‌دهد تا داده‌های موجود در یک استراکچر را شفاف مشاهده کنید. اجازه دهید در این قسمت فرض کنیم که یک آبجکت از یک درایور در آفست 828b2648 قرار دارد و ما قصد داریم هر مقدار مرتبط با این ساختار را نمایش دهیم. لیست ۵ نحوه این کار را نشان می‌دهد.

[1] Viewing Structure Information
[2] Type
[3] Driver Object Structure

```
kd> dt nt!_DRIVER_OBJECT 828b2648
   +0x000 Type             :  4
   +0x002 Size             :  168
   +0x004 DeviceObject     :  0x828b0a30 _DEVICE_OBJECT
   +0x008 Flags            :  0x12
   +0x00c DriverStart      :  0xf7adb000
   +0x010 DriverSize       :  0x1080
   +0x014 DriverSection    :  0x82ad8d78
   +0x018 DriverExtension  :  0x828b26f0 _DRIVER_EXTENSION
   +0x01c DriverName       :  _UNICODE_STRING "\Driver\Beep"
   +0x024 HardwareDatabase :  0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\
HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   :  (null)
   +0x02c DriverInit       : ❶ 0xf7adb66c     long  Beep!DriverEntry+0
   +0x030 DriverStartIo    :  0xf7adb51a      void  Beep!BeepStartIo+0
   +0x034 DriverUnload     :  0xf7adb620      void  Beep!BeepUnload+0
   +0x038 MajorFunction    :  [28] 0xf7adb46a      long  Beep!BeepOpen+0
```

لیست ۶: پوشش اطلاعات در یک ساختار

این درایور، درایور Beep است که برای ایجاد صدای بوق هنگام رخ دادن اشتباهی در سامانه‌عامل مورد استفاده قرار می‌گیرد. در این قسمت ما می‌توانیم مشاهده کنیم که تابع راه‌اندازی اولیه هنگامی‌که درایور بارگذاری می‌شود در آدرس 0xF7ADB66C (شماره ۱) قرار دارد.

اگر این درایور مخرب باشد، باید بررسی کنیم چه کدی در آن آدرس قرار دارد، زیرا آن کد همیشه در ابتدای بارگذاری درایور اجرا می‌شود. قابل ذکر است، تابع راه‌اندازی اولیه تنها تابعی است که هر زمان که درایوری را اجرا می‌کنیم (اجرا به معنای بارگذاری شدن است) فراخوانی می‌شود. از همین روی، بدافزارها گاهی اوقات تمامی محموله مخرب خودشان را در این تابع قرار می‌دهند.

پیکربندی سمبول‌های ویندوز

سمبول‌های دیباگ مایکروسافت برای ویرایش‌های مختلف از یک فایل متفاوت هستند و همچنین می‌توانند با هر بسته به‌روزرسانی یا هاتفیکسی (Hotfix) در سامانه‌عامل تغییر کنند اما اگر پیکربندی WinDBG به درستی انجام شود، می‌تواند به صورت خودکار از سرورهای مایکروسافت اطلاعات سمبول‌های درست را برای فایلی که در حال دیباگ است، دریافت کند. به هر صورت، به منظور استفاده از سمبول‌های دیباگ مایکروسافت در دیباگر WinDBG می‌توانید به منوی File رفته و سپس گزینه Symbol FIle Path را انتخاب کنید. سپس در پنجره جدیدی که باز می‌شود، آدرس آورده شده در قسمت زیر را وارد کنید.

```
SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
```

در فرمان بالا، دستور SRV یک سرور را پیکربندی می‌کند، مسیر c:\ symbols یک مخزن محلی برای ذخیره اطلاعات سمبول‌های دیباگ مایکروسافت و URL آدرس سرور ثابت سمبول‌های مایکروسافت را مشخص می‌کند.

شایان ذکر است، اگر شما ماشینی را دیباگ می‌کنید که بطور پیوسته به اینترنت متصل نمی‌باشد، می‌توانید سمبول‌ها را به صورت دستی از سایت مایکروسافت دانلود کرده و مسیر آن‌ها روی سامانه را به دیباگر WinDBG معرفی کنید تا از آن‌ها در فرایند دیباگ استفاده کند.

البته به این نکته توجه کنید، هنگام دانلود سمبول‌های دیباگ مشخصاً شما باید سمبول‌های دیباگ مربوط به نوع سامانه‌عامل، نسخه سرویس پک و معماری مشخصی را که استفاده می‌کنید، انتخاب کنید.

دیباگ کرنل در عمل

در این بخش، ما برنامه‌ای را آزمایش می‌کنیم که روی فایل‌ها از فضای کرنل اطلاعات می‌نویسد. برای نویسندگان بدافزار، سود نوشتن اطلاعات روی فایل‌ها از محیط کرنل این است که کشف و شناسایی بدافزار بسیار دشوار و سخت خواهد شد. البته به این نکته توجه داشته باشد، این روش مخفی‌ترین راه برای نوشتن اطلاعات روی یک فایل نیست، ولی می‌توان با استفاده از آن برخی از ابزارهای امنیتی مشخص را دور زد و همچنین تحلیلگر بدافزار را گمراه کرد.

شایان ذکر است، توابع معمولی Win32 به راحتی از فضای کرنل قابل دسترسی نیستند که این موضوع یک چالش برای نویسندگان بدافزار ایجاد می‌کند. همچنین از آنجایی‌که توابع CreateFile و WriteFile در فضای کرنل سامانه‌عامل وجود ندارند، بدافزارها به منظور انجام عملیات نوشتن و ایجاد فایل از فضای کرنل معادل این توابع یعنی NtCreateFile و NtWriteFile را مورد استفاده قرار می‌دهند.

مشاهده کد فضای کاربر

در این مثال، یک مولفه فضای کاربر یک درایور ایجاد می‌کند که فایل‌هایی را در کرنل می‌خواند و می‌نویسد. ابتدا ما به کد فضای کاربر در دیزاسمبلر IDA Pro به منظور بررسی اینکه چه توابعی برای تعامل با درایور فراخوانی شده‌اند، نگاهی خواهیم انداخت. همانطور که در لیست ۷ مشاهده می‌کنید.

```
04001B3D  push    esi             ; lpPassword
04001B3E  push    esi             ; lpServiceStartName
04001B3F  push    esi             ; lpDependencies
04001B40  push    esi             ; lpdwTagId
04001B41  push    esi             ; lpLoadOrderGroup
04001B42  push    [ebp+lpBinaryPathName] ; lpBinaryPathName
04001B45  push    1               ; dwErrorControl
04001B47  push    3               ; dwStartType
04001B49  push    ❶1              ; dwServiceType
04001B4B  push    0F01FFh         ; dwDesiredAccess
04001B50  push    [ebp+lpDisplayName] ; lpDisplayName
04001B53  push    [ebp+lpDisplayName] ; lpServiceName
04001B56  push    [ebp+hSCManager] ; hSCManager
04001B59  call    ds:__imp__CreateServiceA@52
```

لیست ۷: ایجاد یک سرویس برای بارگذاری یک درایور سطح کرنل

در روتین‌های مدیریت سرویس مشاهده می‌کنیم که یک درایور با استفاده از تابع CreateService ایجاد شده است. به این نکته توجه، در لیست ۷ پارامتر dwServiceType (شماره ۱) برابر با مقدار 0x01 است که این مقدار نشان‌دهنده این است که این درایور یک درایور سطح کرنل است.

سپس در لیست ۸ مشاهده می‌کنیم که فایلی ایجاد شده است تا با استفاده از فراخوانی CreateFileA (شماره ۱) هندل یک دیوایس را دریافت کند. نام فایل وارد شده به پشته (شماره ۲) در ثبات EDI ذخیره می‌شود (چیزی که اینجا نشان داده نمی‌شود، این است که ثبات EDI با مقدار \\.\FileWriterDevice بارگذاری می‌شود. FileWriterDevice نام آبجکتی است که توسط درایور برای دسترسی برنامه سطح کاربر ایجاد شده است.)

```
04001893                xor     eax, eax
04001895                push    eax             ; hTemplateFile
04001896                push    80h             ; dwFlagsAndAttributes
0400189B                push    2               ; dwCreationDisposition
0400189D                push    eax             ; lpSecurityAttributes
0400189E                push    eax             ; dwShareMode
0400189F                push    ebx             ; dwDesiredAccess
040018A0              ❷push     edi             ; lpFileName
040018A1              ❶call     esi ; CreateFileA
```

لیست ۸: به‌دست آوردن کنترل اشیاء سخت‌افزار

هنگامی‌که بدافزار هندل یک سخت‌افزار را در اختیار گرفت، با استفاده از تابع DeviceIoControl (شماره ۱) داده‌ها مورد نظر خود را به درایور ارسال می‌کند که در لیست ۹ نمایش داده شده است.

```
04001910  push    0               ; lpOverlapped
04001912  sub     eax, ecx
04001914  lea     ecx, [ebp+BytesReturned]
0400191A  push    ecx             ; lpBytesReturned
0400191B  push    64h             ; nOutBufferSize
0400191D  push    edi             ; lpOutBuffer
0400191E  inc     eax
0400191F  push    eax             ; nInBufferSize
04001920  push    esi             ; lpInBuffer
04001921  push    9C402408h       ; dwIoControlCode
04001926  push    [ebp+hObject]   ; hDevice
0400192C  call    ds:DeviceIoControl❶
```

لیست ۹: استفاده از **DeviceIoControl** برای ارتباط بین حالات کاربر و حالت کرنل

مشاهده کدهای سطح کرنل

در این نقطه، ما توجه خود را به سمت مشاهده کدهای سطح کرنل متمرکز می‌کنیم. در این قسمت ما به صورت پویا کدی را که پس از فراخوانی DeviceIoControl اجرا می‌شود، با دیباگ کرنل تحلیل می‌کنیم.

اولین مرحله در این قسمت، شناسایی درایور در کرنل است. اگر دیباگر WinDBG با فعال بودن گزینه Verbose در حال اجرا باشد، هر زمانی‌که یک ماژول کرنل بارگذاری می‌شود، هشداری را دریافت خواهید کرد. ماژول‌های کرنل اغلب بارگذاری (Load) و از حالت بارگذاری خارج (Unload) نمی‌شوند. بنابراین وقتی در حال تحلیل بدافزار هستید و ناگهان یک ماژول کرنل بارگذاری می‌شود، می‌تواند یک نشانه مشکوک از یک ماژول خطرناک باشد.

نکته: در هنگامی‌که از VMware برای دیباگ کرنل استفاده می‌کنید، KMixer.sys مرتب بارگذاری یا از حالت بارگذاری خارج می‌شود. این یک رفتار معمولی بوده و نشانه یک فعالیت مشکوک مرتبط با بدافزار نیست.

به هر صورت، در مثال آورده شده در این قسمت، ما هنگام تجزیه و تحلیل کد، مشاهده خواهیم کرد که درایور FileWriter.sys در پنجره Kernel Debugging دیباگر WinDBG بارگذاری شده است. احتمالا این درایور یک درایور مخرب باشد.

```
ModLoad: f7b0d000 f7b0e780   FileWriter.sys
```

برای این‌که متوجه شویم چه کدی در درایور مخرب فراخوانی شده است، باید آبجکت درایور را پیدا کنیم. از آنجایی که ما نام درایور را می‌دانیم، می‌توانیم درایور آن را با استفاده از فرمان !drvobj پیدا کنیم. لیست ۱۰ نمونه خروجی از فرمان drvobj را نشان می‌دهد.

```
kd> !drvobj FileWriter
Driver object (❶827e3698) is for:
Loading symbols for f7b0d000   FileWriter.sys ->   FileWriter.sys
*** ERROR: Module load completed but symbols could not be loaded for FileWriter.sys
 \Driver\FileWriter
Driver Extension List: (id , addr)

Device Object list:
826eb030
```

لیست ۱۰: مشاهده آبجکت‌ها برای درایوری که بارگذاری شده است

> **نکته:** گاهی اوقات، آبجکت یک درایور نام متفاوتی از خود درایور خواهد داشت یا گاهی‌اوقات فرمان !drvobj جواب نمی‌دهد. به عنوان یک جایگزین می‌توانید آبجکت یک درایور را با استفاده از فرمان !object \Driver پیدا کنید. این فرمان تمامی آبجکت‌ها در فضای‌نام \Driver را لیست می‌کند که یکی از Root Namespaces است که در فصول قبلی مورد بررسی قرار خواهد گرفت.

همانطور که در لیست ۱۰ مشاهده می‌کنید، آبجکت درایور در آدرس 0x827e3698 (شماره ۱) ذخیره شده است. زمانی‌که ما آدرس یک آبجکت درایور را به‌دست آوردیم، می‌توانیم با استفاده از فرمان dt به ساختار آن نگاه کنیم که در لیست ۱۱ خروجی این فرمان نمایش داده شده است.

```
kd>dt nt!_DRIVER_OBJECT 0x827e3698
nt!_DRIVER_OBJECT
   +0x000 Type             : 4
   +0x002 Size             : 168
   +0x004 DeviceObject     : 0x826eb030 _DEVICE_OBJECT
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xf7b0d000
   +0x010 DriverSize       : 0x1780
   +0x014 DriverSection    : 0x828006a8
   +0x018 DriverExtension  : 0x827e3740 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\FileWriter"
   +0x024 HardwareDatabase : 0x8066ecd8 _UNICODE_STRING "\REGISTRY\MACHINE\
                             HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xf7b0dfcd     long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0xf7b0da2a     void  +0
   +0x038 MajorFunction    : [28] 0xf7b0da06     long  +0
```

لیست ۱۱: مشاهده یک آبجکت درایور در سطح کرنل

نقطه ورودی تابع MajorFunction در این ساختار یک اشاره‌گر به اولین آیتم از جدول MajorFunction است. جدول MajorFunction به ما می‌گوید که چه چیزی در هنگام فراخوانی یک درایور مخرب از حالت کاربر فراخوانی می‌شود.

این جدول در هر ایندکس توابع متفاوتی دارد. هر ایندکس یک نوع متفاوتی از درخواست‌ها را نمایش می‌دهد. شایان ذکر است، ایندکس‌ها در فایل wdm.h با IRP_MJ_ شروع می‌شوند. به عنوان مثال، هنگامی‌که ما قصد داریم متوجه شویم هنگامی‌که یک برنامه در فضای کاربر DeviceIoControl را فراخوانی می‌کند، کدام آفست در جدول فراخوانی می‌شود، باید به ایندکس IRP_MJ_DEVICE_CONTROL نگاه کنیم.

در این حالت، IRP_MJ_DEVICE_CONTROL یک مقدار 0xe دارد و جدول MajorFunction در آفست 0x038 از شروع آبجکت درایور آغاز می‌شود. برای شناسایی تابعی که درخواست DeviceIoControl را کنترل می‌کند از فرمان زیر استفاده کنید.

```
dd 827e3698+0x38+e*4 L1
```

در دستور بالا، آدرس 0x038 آفست شروع جدول است و آفست 0xe ایندکس IRP_MJ_DEVICE_CONTROL است که با مقدار عددی چهار ضرب می‌شود. قابل ذکر است، از آنجایی‌که هر اشاره‌گر ۴ بایت است، ایندکس IRP_MJ_DEVICE_CONTROL با عدد ۴ ضرب شده است. در پایان پارامتر L1 مشخص می‌کند که ما فقط می‌خواهیم یک خروجی DWORD مشاهده کنیم.

همانطور که در لیست ۱۲ نمایش داده شده است، فرمان قبلی نشان می‌دهد که تابع فراخوانی شده در فضای کرنل در آدرس 0xf7b0da66 است. همچنین ما می‌توانیم با استفاده از فرمان u بررسی کنیم آیا دستورالعمل‌های در آن آدرس معتبر هستند یا خیر. در این مثال که مشاهده می‌کنید دستورالعمل‌های موجود معتبر هستند، اما اگر معتبر نباشند، معنی آن این است که ما در محاسبه آدرس اشتباه کرده‌ایم.

```
kd> dd 827e3698+0x38+e*4 L1
827e3708  f7b0da66
kd> u f7b0da66
FileWriter+0xa66:
f7b0da66 6a68            push    68h
f7b0da68 6838d9b0f7      push    offset FileWriter+0x938 (f7b0d938)
f7b0da6d e822faffff      call    FileWriter+0x494 (f7b0d494)
```

لیست ۱۲: محل قرارگیری تابع برای IRP_MJ_DEVICE_CONTROL در یک آبجکت درایور

حالا که ما آدرس را داریم، می‌توانیم درایور کرنل را در دیزاسمبلر IDA Pro بارگذاری کنیم یا یک نقطه توقف روی آن تابع تنظیم کرده و تجزیه و تحلیل را درون دیباگر WinDBG ادامه دهیم. معمولا شروع تجزیه و تحلیل تابع ابتدا با استفاده از دیزاسمبلر IDA Pro و سپس انجام تجزیه و تحلیل‌های بیشتر در دیباگر WinDBG ساده‌تر است. به هر حال، هنگام بررسی خروجی درایور مخرب در دیزاسمبلر IDA Pro ما کد آورده شده در لیست ۱۳ را کشف خواهیم کرد که توابع ZwCreateFile و ZwWriteFile را برای نوشتن در یک فایل از فضای کرنل فراخوانی می‌کند.

```
F7B0DCB1  push    offset aDosdevicesCSec ; "\\DosDevices\\C:\\secretfile.txt"
F7B0DCB6  lea     eax, [ebp-54h]
F7B0DCB9  push    eax             ; DestinationString
F7B0DCBA  call    ❶ds:RtlInitUnicodeString
F7B0DCC0  mov     dword ptr [ebp-74h], 18h
F7B0DCC7  mov     [ebp-70h], ebx
F7B0DCCA  mov     dword ptr [ebp-68h], 200h
F7B0DCD1  lea     eax, [ebp-54h]
F7B0DCD4  mov     [ebp-6Ch], eax
F7B0DCD7  mov     [ebp-64h], ebx
F7B0DCDA  mov     [ebp-60h], ebx
F7B0DCDD  push    ebx             ; EaLength
F7B0DCDE  push    ebx             ; EaBuffer
F7B0DCDF  push    40h             ; CreateOptions
F7B0DCE1  push    5               ; CreateDisposition
F7B0DCE3  push    ebx             ; ShareAccess
F7B0DCE4  push    80h             ; FileAttributes
F7B0DCE9  push    ebx             ; AllocationSize
F7B0DCEA  lea     eax, [ebp-5Ch]
F7B0DCED  push    eax             ; IoStatusBlock
F7B0DCEE  lea     eax, [ebp-74h]
F7B0DCF1  push    eax             ; ObjectAttributes
F7B0DCF2  push    1F01FFh         ; DesiredAccess
F7B0DCF7  push    offset FileHandle ; FileHandle
F7B0DCFC  call    ds:ZwCreateFile
F7B0DD02  push    ebx             ; Key
F7B0DD03  lea     eax, [ebp-4Ch]
F7B0DD06  push    eax             ; ByteOffset
F7B0DD07  push    dword ptr [ebp-24h] ; Length
F7B0DD0A  push    esi             ; Buffer
F7B0DD0B  lea     eax, [ebp-5Ch]
F7B0DD0E  push    eax             ; IoStatusBlock
F7B0DD0F  push    ebx             ; ApcContext
F7B0DD10  push    ebx             ; ApcRoutine
F7B0DD11  push    ebx             ; Event
F7B0DD12  push    FileHandle      ; FileHandle
F7B0DD18  call    ds:ZwWriteFile
```

لیست ۱۳: لیست کد برای تابع IRP_MJ_DEVICE_CONTROL

کرنل ویندوز از یک ساختمان UNICODE_STRING (UNICODE_STRING structure) استفاده می‌کند که از رشته کاراکتر گسترده (Wide Character Strings) در فضای کاربر متفاوت است. تابع RtlInitUnicodeString (شماره ۱) برای ایجاد رشته‌های کرنل استفاده شده است. پارامتر دوم این تابع یک رشته کاراکتر گسترده خاتمه یافته به یک کاراکتر NULL[1] از UNICODE_STRING ایجاد شده است.

[1] NULL-terminated wide character string

نام فایل تابع ZwCreateFile که مشاهده می‌کنید *\DosDevices\C:\secretfile.txt* است. همچنین قابل ذکر است، برای ایجاد یک فایل از درون کرنل، باید یک نام کامل (مانند \DosDevices\C:\secretfile.txt) مشخص کنید که دیوایس روت را مشخص می‌کند. برای بیشتر دیوایس‌ها، پوشه روت DosDevices است.

قابل ذکر است، رابط برنامه‌نویسی DEVICEIOCONTROL تنها تابعی نیست که می‌تواند از فضای کاربر به درایورهای کرنل داده ارسال کند. توابع CreateFile، ReadFile، WriteFile و دیگر توابع هم می‌توانند این کار را انجام دهند. به عنوان مثال، اگر یک برنامه سطح کاربر تابع ReadFile را به همراه هندل یک دیوایس فراخوانی کند، تابع IRP_MJ_READ فراخوانی می‌شود. در این مثال، ما توانستیم تابع DeviceIoControl را با افزودن مقدار 0xe*4 به ابتدای جدول Major Function شناسایی کنیم، زیرا مقدار IRP_MJ_DEVICE_CONTROL برابر با 0xe است.

شناسایی آبجکت‌های درایور[1]

در مثال قبلی، هنگامی‌که بدافزار را اجرا کردیم، مشاهده کردیم که یک درایور در فضای کرنل بارگذاری شد و همچنین فرض کردیم آن درایور، آلوده بوده است. گاهی اوقات پیدا کردن درایور درایور[2] خیلی دشوار می‌شود ولی با این حال ابزارهایی وجود دارند که می‌توانند به منظور شناسایی درایور درایور به ما کمک کنند. به منظور فهمیدن نحوه عملکرد این ابزارها، به یاد آورید که برنامه‌ها با دیوایس‌ها (نه درایورها) در تعامل هستند. از برنامه‌های فضای کاربر می‌توانید برای شناسایی درایور دیوایس و از درایور دیوایس می‌توانید برای شناسایی درایور درایور استفاده کنید. شما همچنین می‌توانید از فرمان !devobj با استفاده از نام دیوایس مشخص شده در فراخوانی CreateFile برای دریافت اطلاعات درایور دستگاه استفاده کنید.

[1] Finding Driver Objects
[2] Device Object

```
kd> !devobj FileWriterDevice
Device object (826eb030) is for:
 Rootkit \Driver\FileWriter DriverObject 827e3698
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e13deedc DevExt 00000000 DevObjExt 828eb0e8
ExtensionFlags (0000000000)
Device queue is not busy.
```

لیست ۱۴: خروجی دستور **devobj**

دیوایس آبجکت یک اشاره‌گر به آبجکت درایور ارائه می‌دهد و زمانی‌که آدرس برای دیوایس آبجکت را دارید، می‌توانید جدول تابع مهم را پیدا کنید. پس از این‌که درایور مخرب را شناسایی کردید، شما هنوز باید برنامه‌ای که از آن استفاده می‌کند را کشف و شناسایی کنید. یکی از خروجی های فرمان !devobj که ما اجرا کردیم یک کنترل برای دیوایس آبجکت است. شما می‌توانید از آن کنترل با فرمان !devhandles لیست همه برنامه‌های کاربردی فضای کاربر که یک کنترل به آن دستگاه دارند را به‌دست آورید. این فرمان از طریق هر جدول کنترل برای هر فرایند تکرار می‌شود که مدت زمان زیادی را می‌گیرد. در زیر خروجی مختصر شده فرمان !devhandles به ما نشان می‌دهد که برنامه FileWriterApp.exe از درایور مخرب استفاده می‌کند.

```
kd>!devhandles 826eb030
...
Checking handle table for process 0x829001f0
Handle table at e1d09000 with 32 Entries in use

Checking handle table for process 0x8258d548
Handle table at e1cfa000 with 114 Entries in use

Checking handle table for process 0x82752da0
Handle table at e1045000 with 18 Entries in use
PROCESS 82752da0  SessionId: 0  Cid: 0410    Peb: 7ffd5000  ParentCid: 075c
    DirBase: 09180240  ObjectTable: e1da0180  HandleCount:  18.
    Image: FileWriterApp.exe

07b8: Object: 826eb0e8  GrantedAccess: 0012019f
```

لیست ۱۵: خروجی دستور **devobj**

حال که ما می‌دانیم کدام برنامه کاربردی آلوده شده است، می‌توانیم آن را در فضای کاربر پیدا کنیم و با استفاده از روش‌های مورد بحث قرار گرفته در طول این کتاب آن را مورد تحلیل قرار بدهیم. در این قسمت

ما روش‌های ساده تجزیه و تحلیل درایور های مخرب کرنل را مورد بررسی قرار دادیم، در قسمت بعد، به روش‌های تجزیه و تحلیل روت‌کیت‌ها که عموما به عنوان یک درایور کرنل پیاده‌سازی می‌شوند خواهیم پرداخت.

روت‌کیت‌ها[1]

روت‌کیت‌ها در عملکرد داخلی سامانه‌عامل برای پنهان ساختن خودشان تغییرات ایجاد می‌کنند. این تغییرات می‌تواند فایل‌ها، پروسه‌ها، ارتباطات شبکه و دیگر منابع برنامه‌های در حال اجرا را مخفی کنند و کشف عملیات‌های مخرب آن‌ها را برای تحلیلگران بدافزارها، مدیران و ضدویروس‌ها دشوار کنند.

اکثر روت‌کیت‌ها در انجام عملیات خود به نحوی تغییری در کرنل ایجاد می‌کنند. گرچه روت‌کیت‌ها می‌توانند یک آرایه از روش‌های متفرقه را به کار گیرند، ولی در عمل، یک روش نسب به بقیه بیشتر مورر استفاده قرار می‌گیرد که هوک System Service Descriptor Table نام دارد. این روش چندین سال قدمت دارد و شناسایی ارتباط آن با دیگر روش روت‌کیت‌ها آسان است. به هر حال، این روش هنوز توسط بدافزارها مورد استفاده قرار می‌گیرد، زیرا بسیار قابل فهم، انعطاف پذیر می‌باشد و پیاده‌سازی آن ساده است.

جدول توصیف کننده سرویس‌های سامانه یا به عبارتی System Service Descriptor Table (SSDT)، توسط مایکروسافت به منظور مشاهده فراخوانی توابع درون کرنل مورد استفاده قرار می‌گیرد. جدول توصیف کننده سرویس‌های سامانه (SSDT) معمولا توسط هیچ برنامه یا درایور جانبی مورد دسترس قرار نمی‌گیرد. از فصل هشت به یاد دارید که کد کرنل فقط از فضای کاربر و از طریق دستورالعمل‌های SYSCALL، SYSENTER یا INT 0x2E قابل دسترس است. با این حال، نسخه‌های جدید ویندوز از دستورالعمل SYSENTER استفاده می‌کنند که از کد تابع ذخیره شده در ثبات EAX دستورالعمل دریافت می‌کند. لیست ۱۶ کدی از nt.dll را نشان می‌دهد که تابع NtCreateFile را پیاده‌سازی می‌کند و باید انتقال از فضای کاربر به فضای کرنل که در هر بار فراخوانی NtCreateFile رخ می‌دهد را کنترل کند.

[1] Rootkits

```
7C90D682 ❶mov     eax, 25h        ; NtCreateFile
7C90D687  mov     edx, 7FFE0300h
7C90D68C  call    dword ptr [edx]
7C90D68E  retn    2Ch
```

لیست ۱۶: کد تابع **NtCreateFile**

فراخوانی dword ptr[edx] به دستورالعمل‌های زیر خواهد رفت.

```
7c90eb8b 8bd4  mov     edx,esp
7c90eb8d 0f34  sysenter
```

در لیست ۱۶ ثبات EAX ابتدا با مقدار 0x25 تنظیم شده است (شماره ۱)، سپس اشاره‌گر پشته در ثبات EDX ذخیره شده و در نهایت دستورالعمل sysenter فراخوانی شده است. مقدار ثبات EAX شماره تابع برای NtCreateFile است که به عنوان یک ایندکس هنگامی‌که کد وارد کرنل می‌شود، درون SSDT مورد استفاده قرار می‌گیرد. مخصوصاً، آدرس در آفست 0x25 (شماره ۱) در SSDT در حالت کرنل فراخوانی خواهد شد. در لیست ۱۷ مقدار محدودی از موجودیت‌های در SSDT به همراه موجودیت NtCreateFile در آفست ۲۵ نمایش داده شده است.

```
SSDT[0x22] = 805b28bc (NtCreateaDirectoryObject)
SSDT[0x23] = 80603be0 (NtCreateEvent)
SSDT[0x24] = 8060be48 (NtCreateEventPair)
❶ SSDT[0x25] = 8056d3ca (NtCreateFile)
SSDT[0x26] = 8056bc5c (NtCreateIoCompletion)
SSDT[0x27] = 805ca3ca (NtCreateJobObject)
```

لیست ۱۷: چندین ورودی جدول **SSDT** که**NtCreateFile** را نمایش می‌دهند.

هنگامی‌که یک روت‌کیت یکی از این توابع را هوک کند، مقدار جدول SSDT را تغییر خواهد داد، بنابراین بجای تابع در نظر گرفته شده، کد روت‌کیت در کرنل فراخوانی خواهد شد. در مثال قبلی، ورودی در آفست 0x25 باید تعویض شود تا آنکه به تابع درون درایور مخرب اشاره شود.

این تعویض می‌تواند تابع اجرایی را تغییر دهد، زیرا باز کردن و بررسی فایل مخرب غیر ممکن است. معمولا این موضوع در روت‌کیت‌ها با فراخوانی NtCreateFile و فیلتر نتیجه مبتنی بر تنظیمات روت‌کیت پیاده‌سازی می‌شود. یک روت‌کیت که NtCreateFile را هوک می‌کند فقط از قابل مشاهده بودن فایل در

یک دایرکتوری ممانعت نخواهد کرد. با این حال، شما در آزمایشگاه‌های این فصل، یک روت‌کیت خیلی واقعی خواهید دید که فایل‌ها را از دایرکتوری مخفی می‌کند.

تجزیه و تحلیل روت‌کیت در عمل

در این قسمت، مثالی از یک روت‌کیت که SSDT را هوک می‌کند را مورد بررسی قرار خواهیم داد. در این قسمت یک سامانه آلوده فرضی را تجزیه و تحلیل خواهیم کرد که فکر می‌کنیم در آن ممکن است یک درایور مخرب نصب شده باشد. اولین و همچنین واضح‌ترین راه به منظور بررسی هوک SSDT بررسی SSDT است. SSDT می‌تواند درون دیباگر WinDBG در آفست nt!KeServiceDescriptorTable مورد مشاهده قرار گیرد. تمامی آفست‌های تابع در SSDT باید به توابع درون محدوده ماژول NT اشاره کنند.

بنابراین اولین چیزی که ما باید انجام بدهیم به‌دست آوردن آن محدوده است. در مثال ما، ntoskrnl.exe از آدرس 804d7000 شروع می‌شود و به آدرس 806cd580 پایان پیدا می‌کند. اگر یک روت‌کیت یکی از این توابع را هوک کند، تابع به درون ماژول NT اشاره نخواهد کرد. هنگامی‌که ما SSDT را بررسی می‌کنیم، مشاهده می‌کنیم که یک تابع وجود دارد که ظاهر آن مناسب با فرضیات ما نیست. در لیست ۱۸ نسخه مختصر شده SSDT آورده شده است.

```
kd> lm m nt
...
8050122c  805c9928 805c98d8 8060aea6 805aa334
8050123c  8060a4be 8059cbbc 805a4786 805cb406
8050124c  804feed0 8060b5c4 8056ae64 805343f2
8050125c  80603b90 805b09c0 805e9694 80618a56
8050126c  805edb86 80598e34 80618caa 805986e6
8050127c  805401f0 80636c9c 805b28bc 80603be0
8050128c  8060be48 ❶f7ad94a4 8056bc5c 805ca3ca
8050129c  805ca102 80618e86 8056d4d8 8060c240
805012ac  8056d404 8059fba6 80599202 805c5f8e
```

لیست ۱۸: یک جدول SSDT ساده با یک ورودی بازنویسی شده توسط یک روت‌کیت را نشان می‌دهد.

مقدار در آفست 0x25 این جدول (شماره ۱) به یک تابع که خارج از ماژول ntoskrnl است، اشاره می‌کند. بنابراین یک روتیکت آن تابع را هوک کرده است. تابع هوک شده در این قسمت NtCreateFile است. ما می‌توانیم بدون نصب روتیکت و مشاهده این‌که کدام تابع در آفست قرار داده شده است با بررسی SSDT

روی سامانه بفهمیم کدام تابع هوک شده است. ما با لیست کردن ماژول‌های باز با استفاده از دستور lm که در لیست ۱۹ نمایش داده شده است، متوجه شدیم کدام ماژول شامل آدرس هوک شده است. در کرنل، ماژول‌های لیست شده همه درایور هستند. با این حال، ما درایوری که شامل آدرس 0xf7ad94a4 است را پیدا کرده و مشاهده کردیم که آن درون درایور روت‌کیت فراخوانی شده است.

```
kd>lm
...
f7ac7000 f7ac8580   intelide   (deferred)
f7ac9000 f7aca700   dmload     (deferred)
f7ad9000 f7ada680   Rootkit    (deferred)
f7aed000 f7aee280   vmmouse    (deferred)
...
```

لیست ۱۹: استفاده از فرمان lm به منظور پیدا کردن درایوری که شامل یک آدرس خاص است.

هنگامی‌که درایور را شناسایی کردیم در ادامه شروع به تجزیه و تحلیل آن می‌کنیم. در این فرایند ما به دو چیز نگاه خواهیم انداخت: قسمت کدی که هوک را نصب می‌کند و تابعی که هوک را اجرا می‌کند. ساده‌ترین راه برای شناسایی تابعی که هوک را نصب می‌کنید، جستجو در دیزاسمبلر IDA Pro برای داده‌های ارجاع داده شده به تابع هوک است. لیست ۲۰ کد اسمبلی است که SSDT را هوک کرده است.

```
00010D0D  push    offset aNtcreatefile ; "NtCreateFile"
00010D12  lea     eax, [ebp+NtCreateFileName]
00010D15  push    eax             ; DestinationString
00010D16  mov     edi, ds:RtlInitUnicodeString
00010D1C  call  ❶edi ; RtlInitUnicodeString
00010D1E  push    offset aKeservicedescr ; "KeServiceDescriptorTable"
00010D23  lea     eax, [ebp+KeServiceDescriptorTableString]
00010D26  push    eax             ; DestinationString
00010D27  call  ❷edi ; RtlInitUnicodeString
00010D29  lea     eax, [ebp+NtCreateFileName]
00010D2C  push    eax             ; SystemRoutineName
00010D2D  mov     edi, ds:MmGetSystemRoutineAddress
00010D33  call  ❸edi ; MmGetSystemRoutineAddress
00010D35  mov     ebx, eax
00010D37  lea     eax, [ebp+KeServiceDescriptorTableString]
00010D3A  push    eax             ; SystemRoutineName
00010D3B  call    edi ; MmGetSystemRoutineAddress
00010D3D  mov     ecx, [eax]
00010D3F  xor     edx, edx
00010D41                  ; CODE XREF: sub_10CE7+68 j
00010D41  add   ❹ecx, 4
00010D44  cmp     [ecx], ebx
00010D46  jz      short loc_10D51
00010D48  inc     edx
00010D49  cmp     edx, 11Ch
00010D4F  jl    ❺short loc_10D41
00010D51                  ; CODE XREF: sub_10CE7+5F j
00010D51  mov     dword_10A0C, ecx
00010D57  mov     dword_10A08, ebx
00010D5D  mov   ❻dword ptr [ecx], offset sub_104A4
```

لیست ۲۰: کد روت‌کیت که یک هوک در **SSDT** نصب می‌کند.

این کد تابع NtCreateFile را هوک می‌کند. دو تابع اول فراخونی شده در قسمت شماره ۱ و شماره ۲ برای NtCreateFile و KeServiceDescriptorTable رشته‌ایجاد می‌کنند که در شناسایی آدرس توابع صادراتی توسط ntoskrnl.exe مورد استفاده قرار می‌گیرند و همچنین این مقادیر می‌توانند توسط درایورهای کرنل مشابه مقادیر دیگر به خود وارد یا Import کنند.

قابل ذکر است، شما نمی‌توانید GetProcAddress را از حالت کرنل بارگذاری کنید، اما رابط MmGetSystemRoutineAddress معادل این رابط در کرنل است، گرچه آن کمی از

GetProcAddress متفاوت است ولی با این حال می‌تواند آدرس توابع صادراتی را فقط از hal و ntoskrnl ماژول‌های کرنل دریافت کند.

اولین فراخوانی MmGetSystemRoutineAddress (شماره ۳) آدرس تابع NtCreateFile را نمایش می‌دهد که توسط بدافزار به منظور مشخص ساختن آدرس SSDT بازنویسی شده مورد استفاده قرار می‌گیرد. فراخوانی دوم MmGetSystemRoutineAddress به ما آدرس SSDT خودش را ارائه می‌دهد.

در قسمت بعد از شماره ۴ تا شماره ۵ یک حلقه وجود دارد که SSDT را تکرار می‌کند تا آن یک مقدارکه با آدرس NtCreateFile منطبق باشد را پیدا کند که با هوک تابع بازنویسی خواهد شد. هوک با استفاده از آخرین دستورالعمل در این لیست (شماره ۶) نصب شده است، در جایی که آدرس سابروتین به محل یک حافظه کپی شده است.

تابع هوک یک سری عملیات‌های ساده را انجام می‌دهد. به عنوان مثال، این تابع برخی از درخواست‌ها را فیلتر می‌کند در حالی‌که به دیگران اجازه می‌دهد به NtCreateFile اصلی عبور داده شوند. لیست ۲۱ تابع هوک را نمایش می‌دهد.

```
000104A4  mov     edi, edi
000104A6  push    ebp
000104A7  mov     ebp, esp
000104A9  push    [ebp+arg_8]
000104AC  call    ❶sub_10486
000104B1  test    eax, eax
000104B3  jz      short loc_104BB
000104B5  pop     ebp
000104B6  jmp     NtCreateFile
000104BB ----------------------------
000104BB                  ; CODE XREF: sub_104A4+F j
000104BB  mov     eax, 0C0000034h
000104C0  pop     ebp
000104C1  retn    2Ch
```

لیست ۲۱: لیست تابع هوک روت‌کیت

تابع هوک به تابع NtCreateFile اصلی برای برخی درخواست‌ها پرش می‌کند و برای برخی دیگر به 0xC0000034 باز می‌گردد. مقدار 0xC0000034 معادل

STATUS_OBJECT_NAME_NOT_FOUND است. فراخوانی (شماره ۱) شامل کدی است که ObjectAttributes فایل را بررسی می‌کند که برنامه حالت کاربر برای باز کردن آن، تلاش می‌کند.

شایان ذکر است، اگر تابع NtCreateFile فایل را باز کند تابع هوک یک مقدار غیر صفر و اگر فایل را باز نکند یک مقدار صفر بازگشت خواهد داد. همچنین به این نکته توجه داشته باشید، اگر تابع هوک یک مقدار صفر بازگشت دهد، برنامه حالت کاربر یک پیام خطا (the file does not exist) دریافت خواهد کرد. این موضوع از به‌دست آوردن کنترل به فایل‌های خاص توسط یک برنامه حالت کاربر ممانعت می‌کند در حالی‌که با دیگر فراخوانی‌های NtCreateFile تداخلی ندارد.

وقفه‌ها[1]

در فرهنگ رایانه، وقفه یا اینترراپت (Interrupt) یک سیگنال به ریزپردازنده است که به توجه و پاسخ سریع CPU نیاز دارد. هنگامی‌که یک وقفه رخ می‌دهد، پردازنده عملیات جاری خود را متوقف می‌کند تا به درخواست وقفه رسیدگی کند. ریزپردازنده‌های خانواده ۸۰۸۶ به وقفه‌های تولید شده به وسیله سخت‌افزار و نرم‌افزار پاسخ می‌دهند که به ترتیب به آن‌ها وقفه‌های سخت‌افزاری، و وقفه‌های نرم‌افزاری گفته می‌شود

وقفه‌ها گاهی اوقات توسط روت‌کیت‌ها یا درایورها برای مداخله در رویدادهای سامانه یا اجرای کدهای مخرب مورد استفاده قرار می‌گیرند. یک درایور تابع IoConnectInterrupt را به منظور ثبت یک کنترل‌کننده وقفه فراخوانی می‌کند و سپس یک روتین سرویس وقفه[2] مشخص می‌کند که ویندوز هر بار وقفه تولید شود آن را فراخوانی کند. جدول توصیف‌گر وقفه[3] اطلاعات ISR را ذخیره می‌کند که شما می‌توانید آن اطلاعات را با استفاده از فرمان !idt مشاهده کنید. لیست ۲۲ یک IDT معمولی را نمایش می‌دهد.

[1] Interrupts
[2] Interrupt Service Routine
[3] Interrupt Descriptor Table

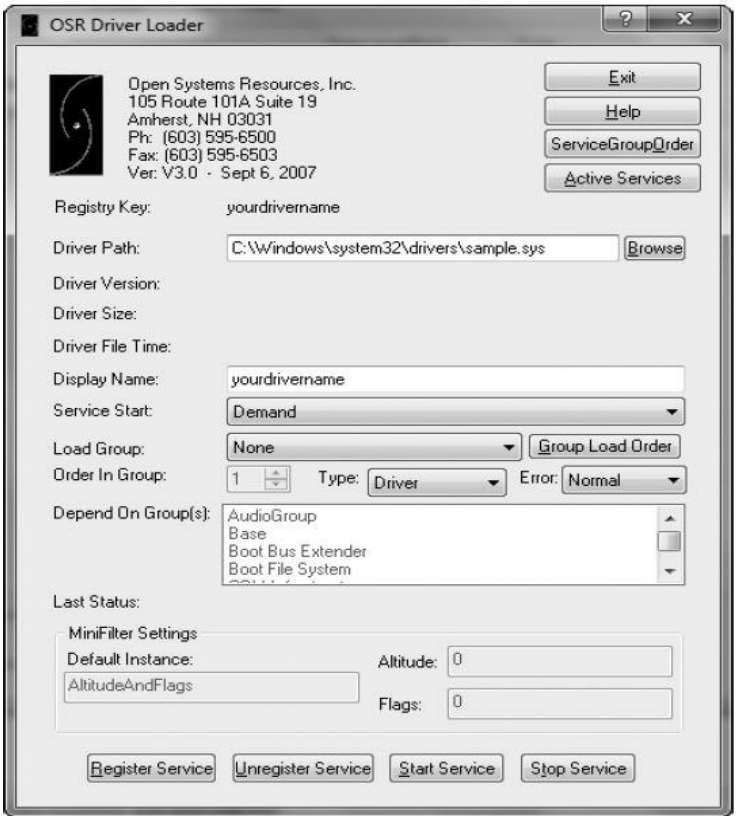
```
kd> !idt

37:   806cf728 hal!PicSpuriousService37
3d:   806d0b70 hal!HalpApcInterrupt
41:   806d09cc hal!HalpDispatchInterrupt
50:   806cf800 hal!HalpApicRebootService
62:   8298b7e4 atapi!IdePortInterrupt (KINTERRUPT 8298b7a8)
63:   826ef044 NDIS!ndisMIsr (KINTERRUPT 826ef008)
73:   826b9044 portcls!CKsShellRequestor::`vector deleting destructor'+0x26
      (KINTERRUPT 826b9008)
            USBPORT!USBPORT_InterruptService (KINTERRUPT 826df008)
82:   82970dd4 atapi!IdePortInterrupt (KINTERRUPT 82970d98)
83:   829e8044 SCSIPORT!ScsiPortInterrupt (KINTERRUPT 829e8008)
93:   826c315c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 826c3120)
a3:   826c2044 i8042prt!I8042MouseInterruptService (KINTERRUPT 826c2008)
b1:   829e5434 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 829e53f8)
b2:   826f115c serial!SerialCIsrSw (KINTERRUPT 826f1120)
c1:   806cf984 hal!HalpBroadcastCallService
d1:   806ced34 hal!HalpClockInterrupt
e1:   806cff0c hal!HalpIpiHandler
e3:   806cfc70 hal!HalpLocalApicErrorService
fd:   806d0464 hal!HalpProfileInterrupt
fe:   806d0604 hal!HalpPerfInterrupt
```

لیست ۲۲: یک جدول توصیف‌گر وقفه ساده

وقفه‌ها بدون نام، بدون نشانه یا درایورهای مشکوک می‌توانند نشان‌دهنده یک روت‌کیت یا دیگر نرم‌افزارهای مخرب باشند.

بارگذاری درایورها

در طول این فصل، ما فرض کردیم که بدافزار تجزیه و تحلیل شده شامل یک کامپوننت فضای کاربر است که با آن بارگذاری می‌شود. اگر شما یک درایور مخرب دارید اما هیچ برنامه فضای کاربری آن را نصب نکرده است، می‌توانید آن درایور را با استفاده از یک بارگذار از قبیل OSR Driver Loader به صورت دستی بارگذاری کنید، همانند تصویر ۹ که این موضوع را نمایش می‌دهد. استفاده از این بارگذار درایور بسیار ساده و رایگان است ولی نیاز به ثبت نام دارد. هنگامی‌که ابزار OSR Driver Loader نصب شد، به سادگی آن را اجرا کنید و درایوری را که می خواهید بارگذاری کنید را مشخص سازید و سپس روی دکمه‌های Register Service و Start Service برای آغاز به کار درایور کلیک کنید.



تصویر ۹: پنجره ابزار **OSR Driver Loader**

دیباگ کرنل ویندوز ویستا، هفت و نسخه‌های ٦٤ بیتی

چندین تغییر مهم در نسخه‌های جدید ویندوز ایجاد شده‌اند که روی فرایند دیباگ کرنل و تاثیرات بدافزار روی کرنل اثر گذاشته‌اند. هنوز هدف بیشتر بدافزارها ماشین‌های ۳۲ بیتی است که روی آن‌ها ویندوز XP در حال اجرا می‌باشد، ولی از آن‌جایی که هر روز بر استفاده‌کنندگان و محبوب ویندوز ۷ و معماری ۶۴ بیتی افزوده می‌شود به زودی هدف نویسندگان بدافزارها قرار خواهند گرفت.

مهم‌ترین تغییر در نسخه ویندوز ویستا به بعد این است که دیگر فایل boot.ini به منظور مشخص ساختن گزینه بوت سامانه‌عامل وجود ندارد. از همین فصل، به یاد آورید که ما برای فعال‌سازی قابلیت دیباگ کرنل ویندوز XP از فایل boot.ini استفاده می‌کردیم ولی از سامانه‌عامل ویستا به بعد ویندوز از یک فایل به نام BCDEditبه منظور ویرایش پیکربندی اطلاعات بوت سامانه استفاده می‌کند، بنابراین شما باید از فایل BCDEdit برای فعال‌سازی قابلیت دیباگ کرنل در نسخه‌های جدید سامانه‌عامل ویندوز استفاده کنید.

بزرگترین تغییر امنیتی ایجاد شده در سامانه‌عامل‌های مایکروسافت معرفی ویژگی PatchGuard است که در معماری x64 پیاده‌سازی شده است. این مکانیزم محافظتی از هرگونه‌ایجاد تغییر در کرنل سامانه‌عامل توسط برنامه‌های جانبی جلوگیری می‌کند. به عنوان مثال این مکانیزم از ایجاد تغییر در خود جدول سرویس‌های سامانه، تغییر IDT و دیگر روش‌های اصلاح (Patching) کرنل جلوگیری می‌کند.

هنگامی‌که این ویژگی معرفی شد بحث برانگیز بود، زیرا ویژگی ایجاد تغییر در کرنل توسط برنامه‌های غیر مخرب هم مورد استفاده قرار می‌گرفت. به عنوان مثال، ضدویروس‌ها، دیوارهای‌آتش و دیگر محصولات امنیتی از مکانیزم ایجاد تغییر در هسته سامانه‌عامل برای شناسایی فعالیت‌های مخرب استفاده می‌کنند.

همچنین باید به این نکته توجه داشته باشید، از آن‌جایی که دیباگر هنگام درج یک نقطه توقف در کد تغییرات ایجاد می‌کند، ممکن است مکانیزم محافظت از اصلاح (Patching) کد کرنل در فرایند دیباگ سامانه‌عامل در سامانه‌های ۶۴ بیتی تداخل ایجاد کند. بنابراین اگر یک دیباگر در هنگام بارگذاری سامانه‌عامل به آن پیوست شود، مکانیزم ممانعت از اصلاح کرنل اجرا نخواهد شد اما اگر پس از بارگذاری سامانه‌عامل یک دیباگر هسته را به سامانه‌عامل پیوست کنید، ویژگی Patch Guard باعث می‌شود سامانه‌عامل Crash کند.

علاوه بر همه این‌ها، در سامانه‌های ۶۴ بیتی جدید مایکروسافت که از ویستا به بعد انتشار پیدا کردند، مکانیزمی مبنی بر ثبت درایورهای جانبی به سامانه‌عامل افزوده شده است. بدین معنی که شما نمی‌توانید در سامانه‌عامل‌های جدید خانواده مایکروسافت به سادگی درایور جانبی بارگذاری و سپس اجرا کنید. مگر این‌که درایور را قبلا به صورت دیجیتالی در سامانه ثبت کرده باشید. به همین دلیل، بدافزارهایی که بتوانند در سطح هسته فعالیت کنند در سامانه‌های ۶۴ بیتی وجود ندارند. با این حال، احتمال این است که در آینده این مکانیزم دور زده شود. به هر صورت اگر نیاز داشتید که درایورها را بدون ثبت بارگذاری کنید و اجرا کنید، می‌توانید فایل BCDEdit را ویرایش کرده و در آن گزینه nointegritychecks را غیرفعال کنید تا ویژگی نیاز به ثبت درایورها متعاقباً در سامانه غیرفعال شود.

نتیجه‌گیری

دیباگر WinDBG یک دیباگر رایگان از شرکت مایکروسافت است که برای تجزیه و تحلیل بدافزارهای پیشرفته و پیچیده بسیار مفید است. این دیباگر ویژگی‌های بسیاری از قبیل توانایی دیباگ کرنل سامانه‌عامل را ارائه می‌دهد که در دیباگر OllyDBG وجود ندارند. قابل ذکر است، بدافزارهایی که از کرنل سامانه‌عامل

استفاده می‌کنند، خیلی زیاد رایج نیستند ولی با این حال وجود دارند و تحلیلگران بدافزار باید با نحوه کنترل و عملکرد آن‌ها آشنایی داشته باشند.

در این فصل، ما نشان دادیم که چگونه درایورهای سامانه‌عامل کار می‌کنند، چگونه از دیباگر WinDBG برای تجزیه و تحلیل آن‌ها باید استفاده کرد، چگونه باید فهمید کدام کد کرنل اجرا می‌شود، هنگامی‌که یک برنامه در فضای کاربر درخواستی ایجاد می‌کند و همچنین چگونه روت‌کیت‌ها را مورد تجزیه و تحلیل قرار بدهیم. در فصل‌های بعدی، بحث خود را از ابزارهای تجزیه و تحلیل به چگونگی تحلیل عملیات‌های بدافزار در سامانه‌های محلی و شبکه متمرکز خواهیم کرد.