

گزارش و راهنمای پروژه AntiAntiDisassmbler Plugin For OllyDBG 2.01(Beta 4)

# طریقه استفاده از این پلاگین

همانطور که در مقاله "تکنیک های AntiDisassembling" گفته ایم، روش های زیادی برای جلوگیری از دیس اسمبل شدن درست برنامه، و در نتیجه عدم درک صحیح آن وجود دارد. ما در این پروژه، یک پلاگین برای OllyDBG ارائه داده ایم، که می‌تواند این تکنیک ها را تشخیص داده، و با پیدا کردن Junk Byteها می تواند این روش ها را خنثی نماید. این پلاگین برای نسخه 2.01 از دیباگر OllyDBG یا نسخه های بعدی تهیه شده است. جهت استفاده از آن ابتدا باید آن را نصب نمایید. برای این کار، فایل AntiAntiDisassembler.dll را به درون پوشه Plugins دیباگر OllyDBG‌ کپی کرده، و سپس فایل Patterns.txt را از داخل پوشه plugin برداشته و در دایرکتوری که OllyDBG.exe‌ قرار دارد، کپی نمایید. توجه کنید که مسیر پوشه Plugins دیباگر OllyDBG‌ از بخش menu->options->options->directories->plugins قابل تنظیم است.

بعد از نصب پلاگین، OllyDBG‌ را باز کرده، و از منوها گزینه File-> Open را انتخاب کنید، و سپس فایل مورد نظر خود را به برنامه بدهید. برای نمونه 8 مثال در آدرس ذیل موجود است

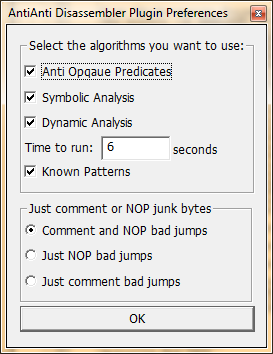
AntiDisassembling Examples

بعد از باز شدن برنامه در درون OllyDBG‌ باید مشخص کنید که در چه بخشی از برنامه می خواهید از پلاگین AntiAntiDisassembling‌ استفاده کنید. برای این کار، باید نقطه آغاز، و نقطه پایان را تعیین کنید. بهتر است محدوده ای که انتخاب می کنید، محدوده یک تابع در برنامه باشد. برای تعیین نقطه آغاز و پایان می توانید در منوها گزینه Plugins 🡪 AntiAntiDisassembler را انتخاب کرده، و سپس یکی از گزینه های Set start point‌ یا Set end point‌ را انتخاب نمایید. با انجام این کار دیالوگی باز می شود که شما می توانید در آن نقطه آغاز یا نقطه پایان را تنظیم نمایید.

نکته: راه ساده تر برای تنظیم نقطه آغاز و پایان کلیک راست کردن در پنجره CPU، انتخاب گزینه AntiAntiDisassembler و انتخاب یکی از گزینه های Set as start point یا Set as end point‌ می باشد. به این ترتیب خط انتخاب شده در پنجره CPU به عنوان نقطه آغاز، یا پایان در نظر گرفته می شود.

پس از تعیین نقطه آغاز یا پایان، می توانید برای شروع عملیات AntiAntiDisassembling، از منوها به ترتیب گزینه های Plugins 🡪 AntiAntiDisassembler 🡪 Start AntiAntiDisassembling‌ را انتخاب نمایید.

توجه کنید که قبل از شروع عملیات، می توانید تغییراتی در تنظیمات پلاگین اعمال کنید. تنظیمات نرم افزار از منوی Plugins 🡪 AntiAntiDisassembler 🡪 Preferences‌ قابل دسترسی است. دیالوگ تنظیمات نرم افزار در شکل یک نشان داده شده است.



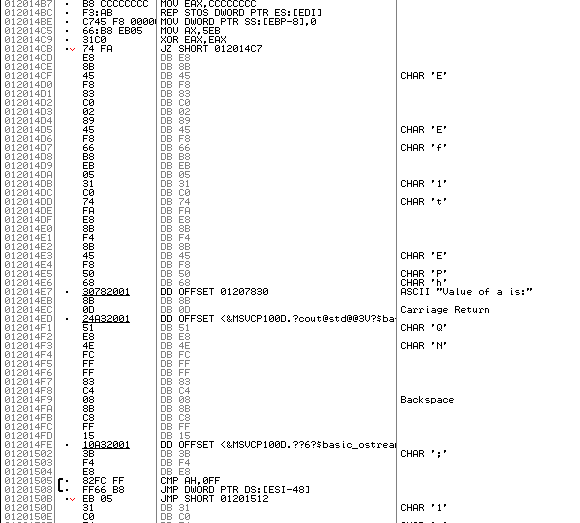
شکل 1: دیالوگ تنظیمات نرم افزار

همانطور که در شکل یک می بینید دیالوگ تنظیمات به دو بخش تنظیم شده است. در بخش اول، می توانید تعیین کنید که از چه الگوریتم هایی برای انجام عملیات AntiAntiDisassembling استفاده شود. این پلاگین در واقع از چهار الگوریتم متفاوت استفاده می کند، که شما می توانید هر کدام از آن ها را فعال، یا غیرفعال کنید. در بخش دوم، می توانید تعیین کنید که چه اتفاقی بر سر Junk Byteهای کشف شده باید بیفتد. اگر فقط می خواهید در پنجره CPU در کنار آن بایت، یک عبارت مشابه "I think this is a junk byte!"‌ نمایش داده شود، آنگاه گزینه "Just comment bad jumps" را انتخاب کنید. اگر می‌خواهید Junk Byteها با دستور NOP جایگزین شوند، گزینه "Just NOP bad jumps" را انتخاب کنید. و اگر می‌خواهید هر دو کار با هم انجام شود، گزینه "Comment and NOP bad jumps"‌ را انتخاب نمایید.

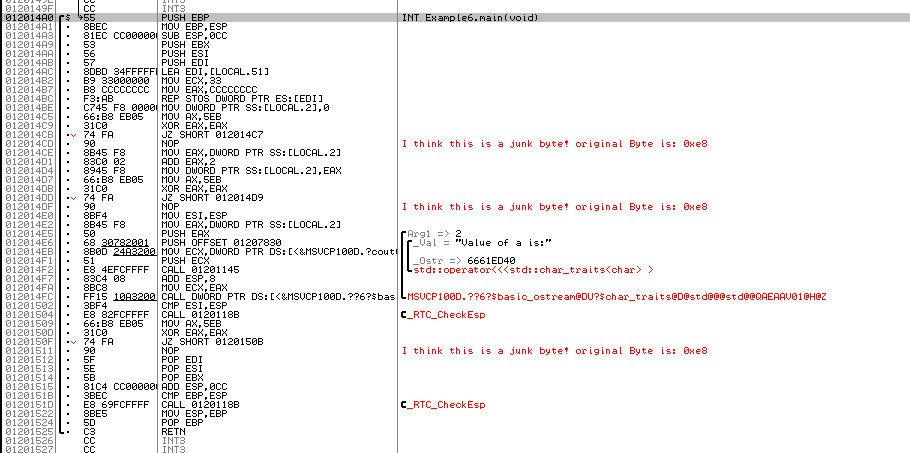
نکته: اگر می بینید که کدها در هنگام فعال بودن قابلیت Analysis دیباگر OllyDBG‌ بسیار ناخوانا هستند، توصیه می شود ابتدا قابلیت Analysis را از کدها حذف کرده، و سپس عملیات AntiAntiDisassembling‌ را آغاز نمایید. برای غیرفعال کردن قابلیت Analysis‌ در پنجره CPU کلیک راست کرده، به ترتیب گزینه های Analysis 🡪 Remove analysis from module را انتخاب نمایید.

نکته: هر کدام از الگوریتم ها که اجرا می شوند، وقتی یک Junk Byte‌ پیدا می کنند، یا اینکه بایتی که قبلا به عنوان Junk Byte‌ در نظر گرفته شده بود، را حذف می کنند، در پنجره Log دیباگر OllyDBG‌ پیغام هایی نمایش می دهند، بررسی این پیغام‌ها می تواند شما را در پیدا کردن بهترین تنظیمات جهت عملیات AntiAntiDisassembling درست یاری نماید.

در شکل دو، نمونه کدی ارائه شده است که از تکنیک Impossible Disassembly جهت جلوگیری از دیس اسمبل شدن درست کد استفاده کرده است. همانطور که می بینید در صورت فعال بودن قابلیت Analysis دیباگر، کلیه کدها توسط دیباگر به عنوان Data در نظر گرفته شده است. در شکل سه نیز همان کد بعد از استفاده از پلاگین طراحی شده نشان داده شده است. همانطور که می بینید Junk Byteها به خوبی تشخیص داده شده، و دیباگر دیگر در دیس اسمبل کردن آن دچار مشکل نخواهد بود.



شکل 2: خروجی OllyDBG برای برنامه ای که از تکنیک Impossible Disassembly استفاده می کند.



شکل 3: پس از استفاده از پلاگین ما، برنامه به درستی دیس اسمبل شده است.

# طریقه کارکرد داخلی این پلاگین

همانطور که گفته ایم این برنامه از چهار الگوریتم متفاوت استفاده می کند. هر کدام از این الگوریتم ها مزایا و معایب خاص خود را دارند، و ممکن است در برخی موارد بسیار خوب، و در برخی موارد دیگر ضعیف عمل کنند. این الگوریتم ها با استفاده از بخش Preferences پلاگین قابل فعال سازی یا غیرفعال سازی خواهند بود. ترتیب اجرای این الگوریتم ها به گونه ای انتخاب شده است که بهترین خروجی حاصل گردد. در اینجا توضیحاتی راجع به ساختار داخلی هر کدام از این الگوریتم ها ارائه می شود. کد این الگوریتم ها در درون فایل های جداگانه قرار گرفته است، و به سادگی از سایر بخش های پلاگین قابل جدا شدن می باشد. البته بخشی از کدهای مربوط به الگوریتم Dynamic Analysis‌ به ناچار در درون فایل اصلی پلاگین یعنی فایل AntiAntiDisassembler.cpp‌ قرار گرفته است.

* الگوریتم Anti Opaque Predicates: این الگوریتم همانطور که از اسمش مشخص است برای جلوگیری از روش Opaque Predicates و البته روش Junk Bytes ارائه شده است. فرض اصلی این الگوریتم این است که هیچ دستور پرشی نباید وجود داشته باشد که مقصد آن به وسط یک دستور باشد. اگر چنین پرشی وجود داشته باشد(که آن را Bad Jump می‌نامیم)، یعنی این وسط یک یا چند Junk Byte وجود دارد.

بنابراین این الگوریتم یک بار کل برنامه را دیساسمبل می کند، و تمامی پرش هایی که مقصدشان وسط یک دستور است را پیدا می کند، و سپس مقصدها را در یک لیست پیوندی به نام bad\_jumps قرار می دهد، سپس با فراخوانی تابع linklist\_sortElements این لیست پیوندی را مرتب کرده، و کوچکترین عضو این لیست پیوندی را می گیرد، و بایت قبلی آن را به عنوان Junk Byte در نظر می گیرد. البته همیشه این کار را نمی کند، اگر بایت قبلی، قبلا جزء Junk Byteها بوده است، بایت قبل تر آن در نظر گرفته می شود. آنقدر این کار تکرار می شود تا دیگر هیچ دستور پرشی پیدا نشود که به وسط یک دستور اشاره کند. توجه کنید که این برنامه چنانچه به حلقه برخورد کند، یا اینکه بایتی کوچکتر از قبلی را به عنوان Junk Byte پیدا کند، آنگاه برگشت به عقب می کند، یعنی بایت هایی که قبلا به عنوان Junk Byte در نظر می گرفته را از لیست خودش حذف می کند، و بایت های جدید را به عنوان Junk Byte‌ در نظر می گیرد.

* الگوریتم Symbolic Analysis: این الگوریتم در واقع کدهای برنامه را به صورت Symbolic اجرا کرده، و روال اجرای برنامه را به دست می آورد. یعنی می فهمد که کدام بایت ها از برنامه اجرا می شوند، و کدام بایت ها هرگز اجرا نمی شوند. این الگوریتم تمامی پرش های شرطی را دو بار چک می کند، یعنی یک بار بررسی می کند که اگر این پرش رخ دهد، برنامه به کجا می رود، و بار دیگر چک می کند که اگر پرش رخ ندهد برنامه به کجا می رود. البته این الگوریتم دچار مشکلاتی نیز می باشد، به عنوان مثال چنانچه پرش به صورت نسبی یا غیرمستقیم باشد، این الگوریتم نمی تواند ادامه مسیر را مشخص کند. مثلا دستوراتی مثل JMP EAX‌ یا JMP DWROD PTR:[EAX+EAX] را در نظر بگیرید. در این گونه موارد که الگوریتم توان پیش بینی ندارد، بیخیال شده، و از ادامه اجرای الگوریتم جلوگیری می شود. البته این الگوریتم سعی می کند بایت هایی که حتی اندکی حدس می‌زند که Junk Byte‌ است را به عنوان Junk Byte‌ در نظر بگیرد. در واقع از قصد میزان تشخیص اشتباه این الگوریتم بالا قرار داده شده است، تا در الگوریتم هایی که بعد از این الگوریتم اجرا می شوند، این بایت هایی که احتمال می رود Junk Byte باشند، با دقت بیشتری بررسی شده، و Junk Byte‌ بودن یا نبودن آن ها به طور دقیق تر تعیین گردد.
* الگوریتم Dynamic Analysis: این الگوریتم در ابتدا یک لیست پیوندی از تمامی بایت های موجود در محدوده ایجاد می‌کند. (این لیست پیوندی all\_bytes نام دارد) سپس کدهای برنامه را به طور واقعی، و در درون دیباگر اجرا می کند، سپس تمامی بایت هایی که اجرا شده اند، را از لیست all\_bytes حذف می کند، چرا که بایتی که اجرا شده است، نمی تواند Junk Byte‌ باشد. (البته ممکن است بایت های بی ارزش باشند، که اجرای آن ها تاثیری ندارد، ولی ما این بایت ها را در الگوریتم های بعدی پیدا می کنیم.) سپس در بین بایت هایی که اجرا نشده اند، سعی می شود Junk Byteها پیدا شود. اما مشکلی که در اینجا وجود دارد این است که ممکن است بایت هایی وجود داشته باشند که Junk Byte نباشند، ولی در روال اجرا نیز نباشند. واضح است که همیشه تمامی دستورات یک روال با هم اجرا نمی شود. بنابراین این الگوریتم بایت هایی را به عنوان Junk Byte در نظر می گیرد که با دستوری که اجرا شده است، تداخل دارند. یعنی اگر دستوری موجود باشد که بخشی از آن اجرا شده، و بخشی از آن اجرا نشده باشد، یعنی به احتمال زیاد بایت هایی از آن دستور که اجرا نشده است، همگی یا بخشی از آن Junk Byte است.

نکته مهمی که در این الگوریتم وجود دارد این است که این الگوریتم اگر ببیند بایتی قبلا به عنوان Junk Byte در نظر گرفته شده است، ولی طی این الگوریتم مشخص شده باشد که Junk Byte نیست، آنگاه آن بایت از لیست Junk Byteها حذف خواهد شد.

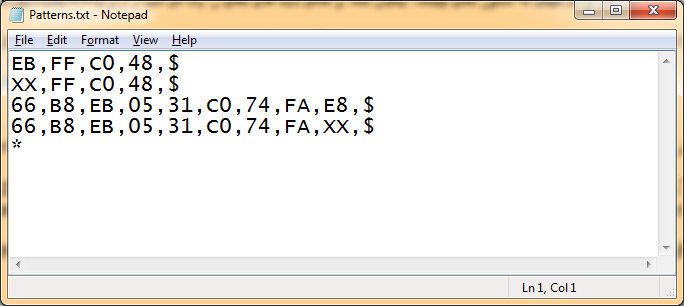
نکته: این الگوریتم زمانی بهترین جواب را می دهد که در هنگام اجرای آن، خطایی در درون دیباگر رخ ندهد، در غیر این صورت شما باید به صورت دستی، و با فشردن کلیدهای CTRL + F9 اجرای برنامه را ادامه دهید.

نکته: شما می توانید مدت زمانی که برنامه اجرا می شود را تعیین نمایید. برای این کار، از بخش Preferences‌ پلاگین استفاده نمایید. این الگوریتم در مدت زمان تعیین شده برنامه را اجرا می کند، و پس از اتمام آن زمان، برنامه را متوقف کرده، و راجع به Junk Byteها تصمیم گیری می کند. لذا انتخاب منطقی این زمان بسیار حائز اهمیت است.

* الگوریتم Known Patterns: اکثر طراحان پکرها، و پروتکتورها از روش های AntiDisassembling‌ از پیش ساخته شده استفاده می کنند. این الگوریتم در واقع در درون کدهای برنامه می گردد، و تکنیک ها یا بایت هایی که در واقع برای روش های AntiDisassembling‌ هستند را پیدا کرده، و آن ها را تصحیح می نماید. برای این کار، این الگوریتم از فایلی به نام Patterns.txt استفاده می کند. محتوای این فایل در شکل چهار به نمایش گذاشته شده است. همانطور که می بینید در این فایل نوشته شده است که اگر دنباله بایت های EB FF C0 48‌ در کدها موجود باشد، آنگاه این دنباله، یک دنباله برای استفاده از تکنیک های AntiAntiDisassembling‌ است، و باید بایت اول آن به عنوان Junk Byte‌ در نظر گرفته شود. (همانطور که می بینید در خط بعدی، بایت اول با XX‌ نمایش داده شده است.)

این الگوریتم ها به ترتیب اجرا می شوند. یعنی ابتدا Anti Opaque Predicates اجرا می شود، سپس Symbolic Analysis، و سپس Dynamic Analysis و Known Patterns اجرا می گردند. البته هر کدام که فعال نباشند، اجرا نمی شوند، و پلاگین مستقیم به سراغ الگوریتم بعدی آن می رود.

کدهای این پلاگین به خوبی Commentگذاری شده اند، و شما می توانید با بررسی آن ها، اطلاعات بیشتری کسب نمایید.



شکل 4: محتوای فایل Patterns.txt

# طریقه نوشتن پلاگین برای OllyDBG

در این مرحله آموزش نوشتن یک پلاگین در ollyDBG آموزش داده می شود. موارد لازم برای انجام این مورد با ویژوال C++ را می توان از سایت [OllyDBG](http://ollydbg.de/) دریافت کرد. در این صفحه یک نمونه پلاگین برای این مورد وجود دارد. بعد از دریافت آن و باز کردن فایل plug201h.zip، داخل آن یک پوشه Visual C وجود دارد. در داخل این پوشه یک فایل Bookmark.vcproj وجود دارد که با باز کردن آن محیط visual stadio برای نوشتن کد خود اماده می شود.

در خود خود یک سری از رویداد های مهم وجود دارد که در زیر آن را مشاهده می کنید که باید فایل dll ای که به عنوان پلاگین آن را می نوسیسد باید تحت یک تابع وجود داشته باشد. لازم به توضیح است که در صورت نیاز از این توابع استفاده می شود.

لازم به توضیح است که تمام کارهای مربوط به کاراکتر ها و مانند ان بر مبنای Unicode می باشد و برای اجرای دستوری مانند کپی کردن یک جمله در یک حافظه می باسیت از تابع StrcopyW استفاده شود.

[ODBG2\_Pluginanalyse](http://ollydbg.de/Help/ODBG2_Pluginanalyse.htm)

در صورت نیاز از این تابع استفاده می شود. و زمانی که محیط آنالیز این نرم افزار کارش به اتمام رسید این فراخوانی می شود. در این قسمت در ادامه کار آنالیز می توان کد پلاگین شما فراخوانی شود.

تابع بصورت

void ODBG2\_Pluginanalyse(t\_module \*pmod);

می باشد ورودی آن نیز از نوع وروی و خروجی است و از نوع t\_module، این یک ساختاری است که توضیحاتی در مورد ماژول های آنالیزی می دهد.  
[ODBG2\_Pluginclose](http://ollydbg.de/Help/ODBG2_Pluginclose.htm)

در صورت نیاز از این تابع استفاده می شود. زمانی که کاربر بخواهد ollydbg را خاموش کند این تابع فراخوانی میشود. این تابع مقدار 0 را بر می گرداند اگر بصورت امن خاموش شود و مقدار غیر صفر اگر برگرداند جلوی خاموش شدن این نرم افزار را می گیرد. اگر شما بخواهید داده ای را برای مرحله های روشن شدن بعدی اجرا کند مورد استفاده قرار می گیرد.

این تابع بصورت

int ODBG2\_Pluginclose(void);

فراخوانی می شود و پارامتر ورودی نیز ندارد.  
[ODBG2\_Plugindestroy](http://ollydbg.de/Help/ODBG2_Plugindestroy.htm)

در صورت نیاز از این تابع استفاده می شود. این تابع زمانی اجرا می شود که کاربر بخواهد این پلاگین را در زمان اجرا پاک کند. شما در این تابع باید منابع خود را آزاد کنید تا این کار با موفقیت انجام گردد.

این تابع بصورت زیر مورد استفاده قرار می گیرد و هیچ پارامتر ورودی نیز دریافت نمی کند و خروجی نیز ندارد.

void ODBG2\_Plugindestroy(void)

[ODBG2\_Plugindump](http://ollydbg.de/Help/ODBG2_Plugindump.htm)

در صورت نیاز از این تابع استفاده می شود. این تابع زمانی از طرف ollydbg فراخونی می شود که این نرم افزار بخواهد عملیات dump را بر روی فایل exe انجام دهد.

این تابع بصورت

int ODBG2\_Plugindump(t\_dump \*pd,wchar\_t \*s,uchar \*mask,int n,int \*select,ulong addr,int column);

فراخوانی میشود و پارامتر هایی که دریافت می کند شامل pd که از نوع t\_dump است، این نوع متغیر اطلاعات مربوط به dump window را در اختیار برنامه نویس قرار می دهد. متغیر s از نوع ورودی و خروجی است و ساختار آن از نوع wchar\_t می باشد که برای ساختار جمله بصورت Unicode مورد استفاده قرار می گیرد. متغیر mask از نوع ورودی و خروجی می باشد، در این نوع متغیر آرایه بوده به طول n byte که برای عملیات mask مورد استفاده قرار می گیرد. متغیر n اندازه آرایه های s و mask می باشد و فقط از نوع ورودی است. select از نوع ورودی و خروجی است، اشاره گری است برای  DRAW\_xxx مورد استفاده قرار می گیرد. Addr نیز از نوع ورودی است و برای نمایش آدرس اول dump مورد استفاده قرار می گیرد. و متغیر column نیز از نوع ورودی است و DF\_FILLCACHE را می فرستد.

[ODBG2\_Pluginexception](http://ollydbg.de/Help/ODBG2_Pluginexception.htm)

در صورت نیاز از این تابع استفاده می شود. زمانی این تابع مورد فراخوانی قرار می گیرد که در زمان دیباگ کردن خطایی رخ داده باشد و exception ای  EXCEPTION\_DEBUG\_EVENT رخ داده باشد

تابع آن بصورت زیر مورد فرارخوانی قرار می گیرد.

int ODBG2\_Pluginexception(t\_run \*prun,const t\_disasm \*da,t\_thread \*pthr,t\_reg \*preg,wchar\_t \*message);

متغیر های این تابع نیز prun یک اشاره گر است از نوع ورودی فقط از نوع ساختار t\_run توضیحاتی در مورد رویداد های دیباگ کردن مورد استفاده قرار می گیرد. متغیر da نیز از نوع ورودی است و اطلاعاتی در مورد قسمت های disassemble شده را بدست می آورد.  
[ODBG2\_Plugininit](http://ollydbg.de/Help/ODBG2_Plugininit.htm)

در صورت نیاز از این تابع استفاده می شود. پلاگين از اين تابع مي تواند براي مقداردهي اوليه استفاده کند. اگر اين تابع موفق باشد، بايد مقدار صفر برگرداند. در غير اين صورت بايد مقدار منفي يک برگرداند.

int ODBG2\_Plugininit(void);

[ODBG2\_Pluginmenu](http://ollydbg.de/Help/ODBG2_Pluginmenu.htm)

در صورت نیاز از این تابع استفاده می شود. این تابع زمانی که ollydbg بخواهد منو را باز کند از پلاگین شما می پرسد که آیا شما برای این قسمت پلاگینی دارید یا خیر. در صورت وجود باید منو خود را به آن بدهید. تابع بصورت زیر نیز باید نوشته شود.

t\_menu \*ODBG2\_Pluginmenu(wchar\_t \*type);

ورودی این تابع از نوع ورودی است و بصورت جمله ای Unicode که نوع منو را برای برنامه نویس مشخص می کند که برنامه نویس منو مخصوص آن را برای نمایش به نرم افزار بدهد. انواع این را نیز می توان در فایل plugin.h دریافت کرد.  
[ODBG2\_Pluginquery](http://ollydbg.de/Help/ODBG2_Pluginquery.htm)

در صورت نیاز از این تابع استفاده می شود. اين تابع براي دريافت اطلاعاتي در مورد پلاگين، اين که چه نسخه اي از اولي ديباگر را پشتيباني مي کند، توسط اولي ديباگر مورد استفاده قرا مي گيرد.

int ODBG2\_Pluginquery(int ollydbgversion,unsigned long \*features,wchar\_t pluginname[SHORTNAME],wchar\_t pluginversion[SHORTNAME]);

متغیر ollydbgversion مقدار ورژن را برای محاسبه ارسال می کند مقدار ورژن برابر می شود با  VERSIONHI\*100+VERSIONLO. متغیر pluginname نام پلاگینی را که نوشته اید را برای ollydbg ارسال می کند و از نوع خروجی است. و pluginversion نیز ورژن پلاگین را باز می گرداند.  
[ODBG2\_Pluginreset](http://ollydbg.de/Help/ODBG2_Pluginreset.htm)

اين تابع هنگامي فراخواني مي شود که برنامه تحت ديباگ ري استارت شده باشد. این تابع نیز در صورت نیاز مورد استفاده قرار می گیرد.

void ODBG2\_Pluginreset(void);

[ODBG2\_Pluginsaveudd](http://ollydbg.de/Help/ODBG2_Pluginsaveudd.htm)

این تابع نیز در صورت نیاز مورد استفاده قرار می گیرد. این تابع برای ذخیره اطلاعات مورد نیاز در زمان خاموش شدن مورد استفاده قرار می گیرد تا در زمان دوباره اجرا شدن آن را دوباره از فایل بخواند.

void ODBG2\_Pluginsaveudd(t\_uddsave \*psave,t\_module \*pmod,int ismainmodule);

ورودی های این تابع همه از نوع ورودی هستند و خروجی نیز ندارد. ماژول برای ذخیره اطلاعات مورد استفاده قرار می گیرد.

[ODBG2\_Plugintempbreakpoint](http://ollydbg.de/Help/ODBG2_Plugintempbreakpoint.htm)

اين تابع در هنگام رسيدن به بريکپوينت هاي موقتي فراخواني مي شود.

void ODBG2\_Plugintempbreakpoint(ulong addr,const t\_disasm \*da,t\_thread \*pthr,t\_reg \*preg);

ورودی های این تابع همه از نوع ورودی هستند و preg از نوع ورودی و خروجی است. متغیر addr آدرس نقطه ای که بریکپوین قرار داده شده است را شامل می شود. متغیر da نیز اطلاعاتی در مورد دیس اسمبلر می دهد.pthr نیز اطلاعاتی در مورد نخ های این برنامه که در حال بیباگ است می دهدو preg نیز اطلاعاتی در مورد وضعیت رجیستر ها را در اختیار برنامه نویس قرار می دهد. خروجی نیز این تابع ندارد.  
[ODBG2\_Pluginuddrecord](http://ollydbg.de/Help/ODBG2_Pluginuddrecord.htm)

اين تابع در هنگام بارگذاري فايل يودي دي فراخواني مي شود.

void ODBG2\_Pluginuddrecord(t\_module \*pmod,int ismainmodule,ulong tag,ulong size,void \*data);

تا اینجای توضیحات توضیحاتی در مورد رویداد ها و نوع های ورودی آن ها داده شده است در این قسمت توضیحاتی در مورد توابع نیز داده خواهد شد که در صورت نیاز می توانید از آنها استفاده کنید.

[**Addtolist**](http://ollydbg.de/Help/Addtolist.htm)

**void Addtolist(ulong addr,int color,wchar\_t \*format,...);**

**پیامی را که در زمان دیباگ نیاز به نمایش دارید را می توان با این تابع دست پیدا کرد و آنرا در قسمت log ها نمایش می دهد. ورودی های این تابع addr از نوع ورودی است و آدرس حافظه اختصاص داده شده را می دهد. متغیر format نیز یک جمله ای که باید نمایش دهد را می گیرد.**

[**Readmemory**](http://ollydbg.de/Help/Readmemory.htm)

**ulong Readmemory(void \*buf,ulong addr,ulong size,int mode);**

**این تابع مقدار حافظه را از پردازه ای که در حال دیباک شدن است می خواند. پارامتر های این تابع نیز mode وضعیت حافظه را ورودی دریافت می کند. متغیر buf بافری برای دریافت حافظه مورد استفاده قرار می گیرد. متغیر addr آدرس شروع را برای دریافت مورد استفاده قرار می دهد. Size نیز اندازه حافظه درخواستی را از سیستم درخواست می کند. مقدار خروجی این تابع اندازه حافظه را که خوانده است را بر میگرداند.**

[**Writememory**](http://ollydbg.de/Help/Writememory.htm)

**ulong Writememory(const void \*buf,ulong addr,ulong size,int mode);**

**این تابع برای نوشتن بر روی حافظه برنامه ای که در حال دیباگ شدن است مورد استفاده قرار می گیرد. ورودی های این تابع بصورت buf است به این معنی دادههای داخل این بافر در حافظه نوشته می شود و انداره ان نیز باید به اندازه size باشد که ان نیز به عنوان ورودی دیگر به این تابع داده می شود. متغیر addr، آدرس مورد نظر را به جهت نوشتن در آن آدرس مورد استفاده قرار می گیرد. Mode نیز حالت حافظه را مشخص می کند. مقدار خروجی نیز اگر بتواند تمام buf را بنویسد مقدار size را بر می گرداند و در صورت ناتمام بودن و دچار خطا شدن مقدار 0 را در خروجی حاصل می کند.**