قسمت ۹ - 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 نهم: مفاهیم پایه دیباگرها

یک دیباگر قطعه نرم‌افزار یا سخت‌افزاری است که برای آزمایش یا بررسی یک برنامه دیگر استفاده می‌شود. از آن‌جایی که برنامه‌ها در نگارش اول خود دارای خطاهای بسیاری هستند، دیباگرها به فرایند توسعه نرم‌افزار و خطایابی آن‌ها کمک بسیاری می‌کنند. در هنگام توسعه یک نرم‌افزار، همواره به برنامه یک ورودی می‌دهید و خروجی آن را مشاهده می‌کنید، اما نمی‌توانید مشاهده کنید که برنامه چگونه آن خروجی را تولید کرده است. دیباگرها به شما کمک می‌کنند تا فرایند اجرایی درون یک برنامه را مشاهده کرده و حالات داخلی برنامه و اجرای یک برنامه را کنترل و ارزیابی کنید.

دیباگرها اطلاعاتی به شما ارائه می‌دهند که اگر غیرممکن نباشد، بسیار دشوار از دیزاسمبلرها به دست می‌آیند. دیزاسمبلرها قبل از اجرا اولین دستورالعمل، یک شکل کلی از یک برنامه ارائه می‌دهند، درحالی‌که دیباگرها یک شکل پویا از برنامه در زمان اجرای آن به شما ارائه می‌کنند. به عنوان مثال، دیباگرها می‌توانند مقادیر آدرس‌های حافظه را حتی زمانی‌که در حین جریان اجرای برنامه عوض می‌شوند به صورت بلادرنگ نمایش دهند.

توانایی ارزیابی و کنترل اجرای یک برنامه دانش عمیقی در طی تجزیه و تحلیل بدافزار به ما ارائه می‌دهد. دیباگرها اجازه می‌دهند مقادیر هر مکان از حافظه، ثبات‌ها و پارامترهای هر تابع را مشاهده کنید و همچنین هرچیزی که مربوط به اجرای برنامه می‌شود را در زمان اجرا تغییر دهید. به عنوان مثال، شما می‌توانید مقدار یک متغیر را در هر لحظه تغییر بدهید، بدین منظور فقط نیاز به اطلاعات متغیر مانند موقعیت آن در حافظه دارید. در دو فصل بعد، دو دیباگر OllyDBG و WinDBG را بررسی خواهیم کرد. اما این فصل روی مفاهیم و مزئیت‌های رایج همه دیباگرها متمرکز خواهد شد.

دیباگرهای سطح کد در مقابل دیباگرهای سطح اسمبلی[1]

بیشتر برنامه‌نویسان با دیباگرهای سطح کد آشنا هستند، این دیباگرها به یک برنامه‌نویس اجازه می‌دهند در حین کدنویسی، برنامه خود را خطایابی کنند. دیباگرهای سطح کد معمولا در محیط‌های برنامه‌نویسی مجتمع[2]

[1] Source-Level vs. Assembly-Level Debuggers
[2] Integrated Development Environments

یا همان IDE مانند Visual Studio قرار دارند. دیباگرهای سطح کد به شما اجازه می‌دهند در برنامه‌های خود نقطه توقف[1] تعبیه کنید که هر وقت اجرای برنامه به آن خط رسید متوقف شود، همچنین با این کار می‌توانید به صورت گام به گام حالات برنامه، مقادیر متغیرها و ثبات‌ها را بررسی کنید (در ادامه این فصل، در مورد نقاط توقف با جزییات بیشتری بحث خواهیم کرد).

دیباگرهای سطح اسمبلی، گاهی اوقات دیباگرهای سطح پایین[2] هم خوانده می‌شوند، این نوع دیباگرها روی کدهای اسمبلی بجای کدهای منبع عملیات خود را انجام می‌دهند. شما می‌توانید دیباگرهای سطح کد اسمبلی را در کنار دیباگرهای سطح کد منبع مورد استفاده قرار بدهید و همانند دیباگرهای سطح کد منبع با ایجاد نقطه توقف روی خطوط کدهای اسمبلی قدم به قدم دستورات را اجرا کنید و آدرس‌های حافظه را بررسی کنید. قابل ذکر است، از آنجایی‌که تحلیلگران بدافزار به کد منبع بدافزارها دسترسی ندارند، به‌طور گسترده‌ای از دیباگرهای سطح اسمبلی استفاده می‌کنند.

دیباگ در حالت کاربر در برابر حالت کرنل[3]

در قسمت‌های گذشته، ما درباره برخی تفاوت‌های میان حالت کاربر و حالت کرنل موجود در سامانه‌عامل ویندوز بحث کردیم. دیباگ کد در حالت کرنل نسبت به دیباگ در حالت کاربر بسیار دشوارتر است، زیرا معمولا نیاز به دو سامانه مختلف برای دیباگ در حالت کرنل دارید. در حالت کاربر، دیباگر و کد در یک سامانه مشابه قرار دارند. همچنین هنگام دیباگ در حالت کاربر، یک فایل‌اجرایی تنها را دیباگ می‌کنید که از مابقی فایل‌های اجرایی سامانه‌عامل جداست.

اما دیباگ کرنل سامانه‌عامل در دو سامانه انجام می‌گیرد، زیرا فقط یک کرنل برای استفاده وجود دارد؛ اگر یک نقطه توقف در کرنل سامانه‌عامل تنظیم کنید، هیچ برنامه کاربردی در آن سامانه اجرا نخواهد شد، به همین دلیل دیباگ کرنل در یک سامانه دشوار است، اما غیر ممکن نیست. در این روش یک سامانه کدی که باید دیباگ شود را اجرا می‌کند و سامانه دیگر، دیباگر را روی خود اجرا می‌کند. علاوه بر این، سامانه‌عامل باید

[1] Breakpoint
[2] Low-level debuggers
[3] Kernel vs. User-Mode Debugging

طوری پیکربندی شود که به ما اجازه دیباگ کرنل را بدهد و در پایان باید دو سامانه را به همدیگر متصل کنید تا بتوانید عملیات دیباگ کرنل را انجام دهید.

نکته: شما می‌توانید یک دیباگر کرنل را در یک سامانه مشابه که برنامه مد نظر برای دیباگ در آن قرار دارد اجرا کنید، اما این روش خیلی رایج نیست. یک برنامه که SoftICE خوانده می‌شود به متخصصین تحلیلگر و کسانی که مهندسین معکوس انجام می‌دهند، این ویژگی را ارائه می‌دهد، اما متاسفانه این برنامه از سال ۲۰۰۷ دیگر پشتیبانی نمی‌شود. همچنین هیچ محصول جدیدی دیگر ساخته نشده است که همچین ویژگی را ارائه دهد.

بسته‌های نرم‌افزاری متفاوتی برای دیباگ در حالت کاربر و دیباگ در حالت کرنل وجود دارند که از مشهورترین آن‌ها می‌توان به دیباگر WinDBG برای دیباگ کرنل و دیباگر OllyDBG و x64dbg و ImmunityDBG برای دیباگ در حالت کاربر اشاره کرد.

استفاده از یک دیباگر

دو روش برای دیباگ یک برنامه وجود دارد. اولین روش این است که برنامه را مستقیما در دیباگر بارگذاری و اجرا کنید. در این حالت، زمانی‌که برنامه را اجرا می‌کنید و برنامه در حافظه بارگذاری می‌شود، دیباگر پیش از اجرای برنامه در نقطه ورود[1] آن یک نقطه توقف قرار داده و اجرای برنامه را موقتا متوقف می‌کند و کنترل کامل آن را به شما ارائه می‌دهد.

در روش دیگر می‌توانید یک دیباگر را به یک برنامه در حال اجرا پیوست کنید. در این حالت، تمامی تردهای برنامه موقتا متوقف می‌شود و شما می‌توانید آن برنامه را گام به گام دیباگ کنید. شایان ذکر است، روش دوم زمانی‌که می‌خواهید از دیباگر برای دیباگ یک برنامه در حال اجرا یا فرایندی را که توسط یک بدافزار دستکاری شده است، دیباگ کنید یک رویکرد خوب است.

[1] Entrypoint

دیباگ به روش گام به گام[1]

راحت‌ترین چیزی که شما می‌توانید با استفاده از یک دیباگر انجام دهید، دیباگ برنامه به روش گام به گام است، بدین معنی‌که یک دستورالعمل را اجرا می‌کنید و سپس کنترل اجرای برنامه به دیباگر بازگشت داده می‌شود. روش گام به گام در برنامه به شما اجازه می‌دهد تمامی چیزهایی که در برنامه رخ می‌دهد را مشاهده کنید.

شما می‌توانید یک برنامه را به شکل کامل با استفاده از روش گام به گام دیباگ کنید، اما این کار را در برنامه‌های پیچیده و بزرگ نباید انجام دهید زیرا زمان زیادی را از شما خواهد گرفت. دیباگ به روش قدم به قدم یک روش خوب برای فهمیدن جزییات یک بخش از کد برنامه است، اما خود شما باید انتخاب کنید که کدام بخش از کد را بدین شکل تجزیه و تحلیل کنید. با این حال روی مفاهیم کلی تمرکز کنید تا در جزییات برنامه سردرگم نشوید. به عنوان مثال، دیزاسمبلی آورده شده در لیست ۱ نشان می‌دهد که چگونه دیباگر به فهمیدن یک بخش از کد می‌تواند به ما کمک کند.

```
mov     edi, DWORD_00406904
mov     ecx, 0x0d
LOC_040106B2
xor     [edi], 0x9C
inc     edi
loopw   LOC_040106B2
...
DWORD:00406904:    F8FDF3D0❶
```

لیست ۱: گام برداشتن در کد

این لیست آدرس‌های داده[2] که در یک حلقه[3] دسترسی یافته و تغییر داده شده‌اند، نشان می‌دهد. مقدار داده که در پایان (شماره ۱) مشخص شده است، به نظر نمی‌رسد یک متن ASCII یا هر مقدار قابل تشخیص دیگری باشد، با این حال شما می‌توانید از یک دیباگر برای قدم برداشتن در این حلقه استفاده کنید تا متوجه شوید این حلقه چه کاری انجام می‌دهد.

[1] Single-Stepping Debugging
[2] Data Address
[3] Loop

اگر در این حلقه با استفاده از WinDBG یا OllyDBG گام بردارید، مشاهده خواهید کرد که مقدار داده مذکور تغییر پیدا می‌کند. به عنوان مثال، در لیست ۲، مشاهده می‌کنید که در هر بار اجرای حلقه ۱۳ بایت این تابع تغییر پیدا کرده است. (این لیست بایت‌های متعلق به آدرس‌ها را به همراه مقادیر ASCII آن‌ها نمایش می‌دهد.)

```
D0F3FDF8 D0F5FEEE FDEEE5DD 9C (.............)
4CF3FDF8 D0F5FEEE FDEEE5DD 9C (L.............)
4C6FFDF8 D0F5FEEE FDEEE5DD 9C (Lo............)
4C6F61F8 D0F5FEEE FDEEE5DD 9C (Loa...........)
. . . SNIP . . .
4C6F6164 4C696272 61727941 00 (LoadLibraryA.)
```

لیست ۲: قدم به قدم گام برداشتن در یک بخش از کد به منظور مشاهده تغییرات حافظه

با یک دیباگر پیوست شده، به‌طور خیلی واضح می‌توان دریافت که این تابع از یک دستور سلسله مراتبی XOR برای رمزگشایی رشته LoadLibraryA استفاده کرده است. شناسایی این رشته در تجزیه و تحلیل استاتیک بسیار دشوار است.

گام برداشتن از روی[1] در برابر گام برداشتن به درون[2]

هنگام دیباگ کد، دیباگر پس از اجرای هر دستورالعمل توقف می‌کند. اگرچه، تحلیلگران بدافزار به طور کلی علاقمند هستند که متوجه شوند برنامه چه کاری انجام می‌دهد، اما دانستن این‌که هر قسمت از برنامه چه کاری انجام می‌دهد یک کار غیر ممکن است. به عنوان مثال، اگر برنامه تابع LoadLibrary را فراخوانی کند، نیاز نیست هر دستورالعمل تابع LoadLibrary را تک تک مورد بررسی قرار بدهید.

به منظور کنترل دستورالعمل‌ها که در دیباگر خود مشاهده می‌کنید، می‌توانید به دستورالعمل‌ها step-into یا step-over کنید. هنگامی‌که شما یک دستورالعمل فراخوانی را step-over می‌کنید، آن را دور می‌زنید. به عنوان مثال، اگر یک فراخوانی را step-over کنید، دستورالعمل بعدی که در دیباگر مشاهده خواهید کرد، دستورالعملی خواهد بود که پس از خروج تابع اجرا خواهد شد. اما اگر به یک فراخوانی تابع step-into

[1] Stepping-Over
[2] Stepping-Into

کنید، دستورالعمل بعدی که در دیباگر خواهید دید، اولین دستورالعمل اجرایی درون تابع فراخوانی شده خواهد بود.

Step-over به شما اجازه می‌دهد به طور چشمگیری مقدار دستورالعمل‌های اجرایی که نیاز به تحلیل آن‌ها دارید را کاهش دهید، ولی این کار دارای ریسک زیادی است و ممکن است برخی از ویژگی‌های مهم یک تابع را از دست بدهید. علاوه بر این، بعضی فراخوانی‌های توابع هیچ وقت بازگشت داده نمی‌شوند و اگر فراخوانی یک تابع برنامه بازگشت داده نشود و از روی آن پرش کنید، دیباگر نمی‌تواند هرگز کنترل دوباره برنامه را بازیابی کند. هنگامی‌که این اتفاق افتاد (احتمالا این اتفاق رخ خواهد داد)، برنامه را باید راه‌اندازی مجدد یا Restart کنید و به همان محل دوباره قدم بردارید، اما این بار باید به آن تابع step-into کنید.

هنگام step-into کردن به یک تابع، به سرعت قدم برداری میان دستورالعمل‌های تابع آغاز خواهد شد که برخی از آن‌ها هیچ ارتباطی با آنچه که شما در حال تجزیه و تحلیل آن هستید، نخواهند داشت. با این حال برخی از دیباگرها قابلیتی دارند که به شما اجازه می‌دهند به مرحله گذشته برگردید (Undo Step) و دوباره گام برداری در میان کدهای برنامه را از پیش بگیرید یا به پایان اجرای یک تابع پرش کنید (Till return). همچنین باید ذکر کرد در برخی دیباگرها گزینه‌ای وجود دارد با نام step-out که شما می‌توانید با استفاده از آن کاملا از تابع خارج شوید.

توقف موقت اجرای برنامه با نقاط توقف[1]

نقاط توقف برای متوقف ساختن موقت اجرای برنامه و بررسی حالت برنامه استفاده می‌شوند. از آنجایی‌که ما نمی‌توانیم در حین اجرای برنامه به ثبات‌ها یا آدرس‌های حافظه دسترسی پیدا کنیم و مقادیر آن‌ها را به صورت بلادرنگ مشاهده کنیم، لذا به نقاط توقف نیاز خواهیم داشت.

لیست ۳ نشان می‌دهد که کجا یک نقطه توقف مفید خواهد بود. در این مثال، یک فراخوانی با استفاده از ثبات EAX وجود دارد. هنگامی‌که یک دیزاسمبلر نمی‌تواند به شما بگوید چه تابعی فراخوانی شده است، شما می‌توانید به منظور فهمیدن این موضوع یک نقطه توقف روی آن دستورالعمل تنظیم کنید. هنگامی‌که برنامه

[1] Pausing Execution with Breakpoints

با این نقطه توقف برخورد کند، متوقف خواهد شد و دیباگر به شما مقدار EAX را نشان خواهد داد که مقصد تابعی است که باید فراخوانی شود.

```
00401008   mov     ecx, [ebp+arg_0]
0040100B   mov     eax, [edx]
0040100D   call    eax
```

لیست ۳: فراخوانی به EAX

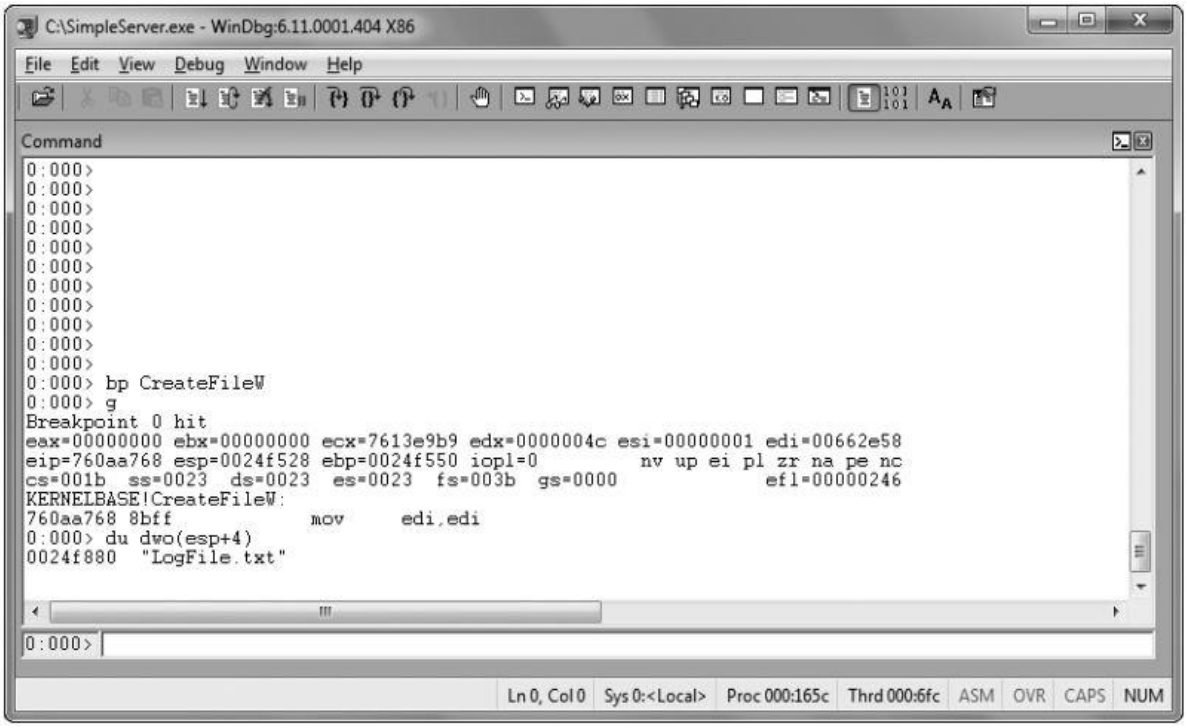
نمونه دیگر از این مثال لیست ۴ است که شروع یک تابع را به همراه فراخوانی تابع CreateFile به منظور ایجاد یک فایل را نشان می‌دهد. اگرچه بخشی از نام فایل در دیزاسمبلی به عنوان یک پارامتر به تابع ارسال شده است، اما مشخص کردن نام کامل فایل در آن بسیار دشوار است. برای شناسایی فایل در دیزاسمبلی، می‌توانید از دیزاسمبلر IDA Pro استفاده کنید. زیرا با استفاده از IDA Pro می‌توانید تمامی دفعاتی که این تابع فراخوانی شده است را جستجو کنید و مقادیر عبور داده شده به آن را مورد بررسی قرار دهید.

```
0040100B  xor     eax, esp
0040100D  mov     [esp+0D0h+var_4], eax
00401014  mov     eax, edx
00401016  mov     [esp+0D0h+NumberOfBytesWritten], 0
0040101D  add     eax, 0FFFFFFFEh
00401020  mov     cx, [eax+2]
00401024  add     eax, 2
00401027  test    cx, cx
0040102A  jnz     short loc_401020
0040102C  mov     ecx, dword ptr ds:a_txt ; ".txt"
00401032  push    0               ; hTemplateFile
00401034  push    0               ; dwFlagsAndAttributes
00401036  push    2               ; dwCreationDisposition
00401038  mov     [eax], ecx
0040103A  mov     ecx, dword ptr ds:a_txt+4
00401040  push    0               ; lpSecurityAttributes
00401042  push    0               ; dwShareMode
00401044  mov     [eax+4], ecx
00401047  mov     cx, word ptr ds:a_txt+8
0040104E  push    0               ; dwDesiredAccess
00401050  push    edx             ; lpFileName
00401051  mov     [eax+8], cx
00401055 ❶call    CreateFileW ; CreateFileW(x,x,x,x,x,x,x)
```

لیست ۴: استفاده از یک دیباگر برای مشخص ساختن نام فایل

ما یک نقطه توقف در زمان فراخوانی تابع CreateFileW (شماره ۱) تنظیم کرده‌ایم. سپس هنگامی‌که اجرای برنامه به نقطه توقف می‌رسد و اجرای برنامه موقتا متوقف می‌شود، مقدار درون پشته را بررسی می‌کنیم.

تصویر ۱ دستورالعمل مشابه یک نقطه توقف در دیباگر WinDBG را نمایش می‌دهد. بعد از نقطه توقف، ما اولین پارامتری که به تابع عبور داده می‌شود را به عنوان یک رشته ASCII در WinDBG مشاهده می‌کنیم.



تصویر ۱: استفاده از یک نقطه توقف برای مشاهده پارامترهای ارسالی به یک تابع. ما یک نقطه توقف در قسمت فراخوانی تابع CreateFileW تنظیم کردیم و سپس اولین پارامتر ارسالی به پشته را مورد بررسی قرار دادیم.

در این نمونه، همانطور که مشاهده می‌کنید، نام فایل ایجاد شده LogFile.txt است. شایان ذکر است، در این مثال ما می‌توانستیم از دیزاسمبلر IDA Pro استفاده کنیم، اما استفاده از یک دیباگر برای دریافت این اطلاعات ساده‌تر و سریع‌تر است.

حالا تصور کنید که ما بخشی از یک بدافزار و همچنین یک بسته کپچر[1] شده را در دسترس داریم. در بسته کپچر شده، ما اطلاعات را به صورت رمز شده مشاهده می‌کنیم. ما می‌توانیم فراخوانی send را پیدا کرده و کد رمزنگاری را کشف کنیم، اما رمزگشایی بسته‌های کپچر شده شبکه توسط خودمان بسیار دشوار است، زیرا کلید رمز و فرایند رمزنگاری را نمی‌دانیم. خوشبختانه، از آنجایی‌که در توابع از روتین‌های رمزنگاری توابع مجزایی هستند که داده‌ها را انتقال می‌دهند ما می‌توانیم از یک دیباگر به منظور ساده کردن این کار استفاده

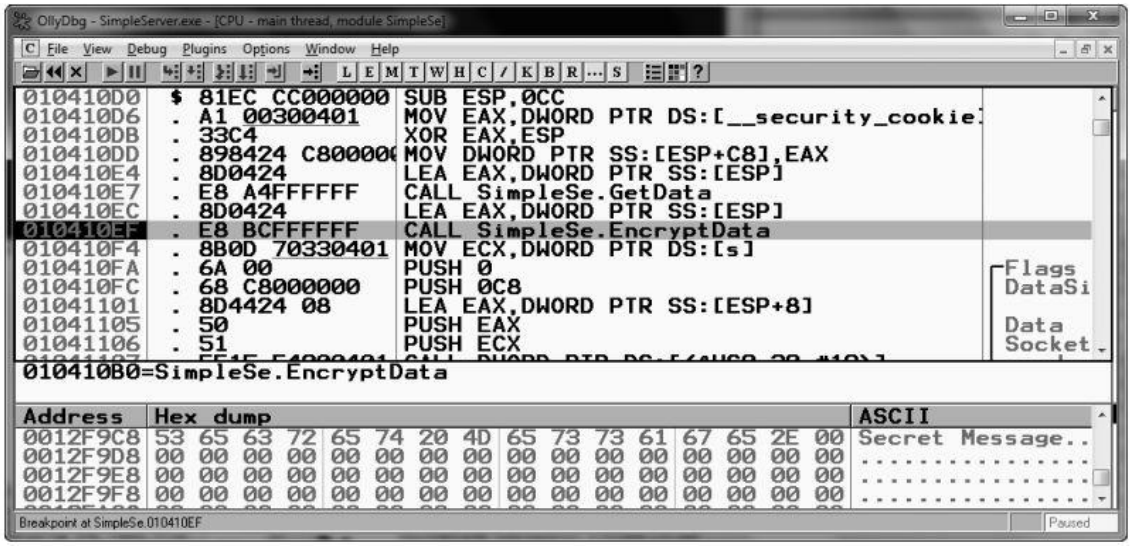
[1] Packet Capture

کنیم. اگر ما بتوانیم محل فراخوانی تابع رمزنگاری را پیدا کنیم، می‌توانیم با تنظیم یک نقطه توقف پیش از رمزنگاری داده‌ها، محتوای داده‌ای که قرار است ارسال شود را مشاهده کنیم. همان‌طور که در دیزاسمبلی لیست ۵ (شماره ۱) مشاهده می‌کنید.

```
004010D0  sub     esp, 0CCh
004010D6  mov     eax, dword_403000
004010DB  xor     eax, esp
004010DD  mov     [esp+0CCh+var_4], eax
004010E4  lea     eax, [esp+0CCh+buf]
004010E7  call    GetData
004010EC  lea     eax, [esp+0CCh+buf]
004010EF ❶call    EncryptData
004010F4  mov     ecx, s
004010FA  push    0               ; flags
004010FC  push    0C8h            ; len
00401101  lea     eax, [esp+0D4h+buf]
00401105  push    eax             ; buf
00401106  push    ecx             ; s
00401107  call    ds:Send
```

لیست ۵: استفاده از یک نقطه توقف برای مشاهده کردن داده قبل ار رمزنگاری

تصویر ۲ نمایانگر محیط دیباگر OllyDBG است که مقادیر حافظه را پیش از ارسال به فرایند رمزنگاری، نمایش می‌دهد. پنجره بالایی دیباگر OllyDBG دستورالعمل‌های اسمبلی برنامه را به همراه نقطه توقف نمایش می‌دهد و پنجره پایین نمایانگر محتوای پیام پیش از رمزنگاری است. در این مورد، داده ارسال شده پیام Secret Message است که در جدول ASCII پنجره پایین نمایش داده شده است.



تصویر ۲: مشاهده داده برنامه پیش از فراخوانی تابع رمزنگاری

شایان ذکر است، شما می‌توانید از نقاط توقف متفاوتی از جمله نقاط توقف نرم‌افزاری، سخت‌افزاری و شرطی استفاده کنید. گرچه تمامی این نقاط توقف اهداف مشابه را دنبال می‌کنند، با این حال، مطابق با موقعیت، بعضی نقاط توقف کار نمی‌کنند در حالیکه برخی دیگر کار می‌کنند.

نقاط توقف نرم‌افزاری

تا این قسمت درباره نقاط توقف نرم‌افزاری صحبت کرده‌ایم که می‌توانند موجب توقف برنامه هنگام اجرای یک دستورالعمل خاص شوند. شایان ذکر است، هنگامی‌که شما بدون در نظر گرفتن هیچ گزینه‌ای یک نقطه توقف در برنامه تنظیم می‌کنید، اکثر دیباگرها نقطه توقف نرم‌افزاری[1] را به عنوان گزینه پیش‌فرض انتخاب می‌کنند.

یک دیباگر می‌تواند با بازنویسی اولین بایت از یک دستورالعمل اسمبلی با مقدار 0xCC که معادل وقفه INT 3 است، یک نقطه توقف نرم‌افزاری را پیاده‌سازی کرده و برنامه مورد دیباگ را موقتا متوقف کند. شایان ذکر است، زمانی‌که این دستورالعمل اجرا می‌شود، سامانه‌عامل یک اکسپشن ایجاد کرده و کنترل برنامه را به دیباگر ارائه می‌دهد. جدول ۱ یک دامپ از حافظه[2] و همچنین دیزاسمبلی یک تابع حاوی نقطه توقف را در کنار هم نشان می‌دهد.

جدول ۱: دیزاسمبلی و دامپ حافظه یک تابع حاوی نقطه توقف

Disassembly view			Memory dump
00401130 55	❶push	ebp	00401130 ❷CC 8B EC 83
00401131 8B EC	mov	ebp, esp	00401134 E4 F8 81 EC
00401133 83 E4 F8	and	esp, 0FFFFFFF8h	00401138 A4 03 00 00
00401136 81 EC A4 03 00 00	sub	esp, 3A4h	0040113C A1 00 30 40
0040113C A1 00 30 40 00	mov	eax, dword_403000	00401140 00

در قسمت دیزاسمبلی جدول ۱ مشاهده می‌کنید که تابع با دستورالعمل push ebp (شماره ۱) معادل کد ماشین 0x55 آغاز شده است، اما در قسمت دامپ از حافظه همانطور که مشاهده می‌کنید کد این تابع با مقدار 0xCC (شماره ۲) شروع می‌شود که نشان‌دهنده یک نقطه توقف نرم‌افزاری است. به این نکته توجه کنید،

[1] Software Breakpoint
[2] Memory Dump

همواره در فرایند دیباگ، پنجره دیزاسمبلی دیباگر دستورالعمل‌های خام برنامه و پنجره دامپ حافظه در دیباگر دستورالعمل‌های واقعی برنامه در حافظه را نشان می‌دهند.

شایان ذکر است، درصورتی‌که این مقادیر در زمان اجرای برنامه تغییر کنند، نقطه توقف اجرا نخواهد شد. به عنوان مثال، اگر یک نقطه توقف را در بخشی از کد قرار دهید و کد خود را تغییر دهد یا کد دیگری آن بخش از کد را تغییر دهد نقطه توقف شما پاک می‌شود. همچنین به این نکته توجه کنید، در حین فرایند دیباگ می‌توانید تعداد نامحدودی نقطه توقف نرم‌افزاری در حالت کاربر قرار دهید، ولی در حالت کرنل محدودیت وجود دارد.

نقاط توقف سخت‌افزاری

معماری x86 با استفاده از ثبات‌های مجزایی از نقاط توقف سخت‌افزاری پشتیبانی می‌کند. هر زمان که پردازنده یک دستورالعمل را اجرا می‌کند، سخت‌افزاری وجود دارد که بررسی می‌کند ثبات اشاره‌گر دستورالعمل (EIP) با آدرس نقطه توقف برابر است یا خیر.

برخلاف نقاط توقف نرم‌افزاری، در نقاط توقف سخت‌افزاری این نکته مهم نیست چه داده‌هایی در یک آدرس ذخیره شده‌اند. به عنوان مثال، اگر در آدرس 0x00401234 یک نقطه توقف قرار بدهید، صرف‌نظر از این‌که در آن آدرس چه چیزی ذخیره شده است، پردازنده در آن آدرس توقف خواهد کرد. این مزیت هنگام تحلیل کدی که خودش را تغییر می‌دهد، بسیار مفید است.

نقاط توقف سخت‌افزاری یک مزیت دیگری نسبت به نقاط توقف نرم‌افزاری دارند و آن قابلیت ایجاد توقف در حین دسترسی[1] بجای زمان اجرا[2] است. به عنوان مثال، می‌توانید یک نقطه توقف در برنامه تنظیم کنید تا هر وقت یک قسمت از حافظه برنامه خوانده یا در آن اطلاعاتی نوشته شد، برنامه متوقف شود. همچنین اگر بخواهید مشخص سازید که چه داده‌ای در یک آدرس حافظه ذخیره می‌شود، می‌توانید یک نقطه توقف سخت‌افزاری در آن مکان از حافظه تنظیم کنید. سپس هنگامی‌که در آن آدرس از حافظه عملیات نوشتن انجام شد، دیباگر با صرف نظر از آدرس دستورالعملی که باید اجرا شود، برنامه را متوقف خواهد ساخت. (همچنین می‌توانید یک نقطه توقف برای خواندن، نوشتن یا برای هر دو تنظیم کنید).

[1] Access Breakpoint
[2] Execution Breakpoint

متاسفانه، نقاط توقف سخت‌افزاری یک ضعف بزرگ دارند، آن هم این است که فقط چهار ثبات سخت‌افزار آدرس‌های نقاط توقف را ذخیره می‌کنند. یک ضعف دیگر نقاط توقف سخت‌افزاری این است که به سادگی توسط برنامه در حال اجرا تغییر می‌کنند. هشت ثبات برای دیباگ در پردازنده وجود دارد، اما فقط از شش تا از آن‌ها استفاده می‌شود. چهار تا اول، DR0 تا DR3 هستند که آدرس‌های نقاط توقف را ذخیره می‌کنند و ثبات بعدی کنترل کننده دیباگ (DR7) است که اطلاعات دیباگ را ذخیره می‌کند.

برنامه‌های مخرب اغلب برای مداخله در فرایند دیباگ این ثبات‌ها را تغییر می‌دهند. خوشبختانه، پردازنده x86 یک ویژگی دارد که در برابر این این موضوع سامانه را محافظت می‌کند. بدین منظور شما باید پرچم General Detect را با ثبات DR7 تنظیم کنید تا قبل از اجرای هر دستورالعمل mov که به ثبات دیباگ دسترسی می‌گیرد، یک نقطه توقف رخ دهد. گرچه این روش خوب نیست (چون فقط دستورالعمل‌های mov که به ثبات‌های دیباگ دسترسی می‌گیرند را شناسایی می‌کند)، با این وجود با ارزش است.

نقاط توقف شرطی

نقاط توقف شرطی، نقاط توقف نرم‌افزاری هستند که فقط زمانی‌که یک شرایط خاصی برقرار باشد رخ می‌دهند و موجب توقف اجرای برنامه می‌شوند. به عنوان مثال، فرض کنید شما یک نقطه توقف در قسمت فراخوانی GetProcAddress دارید. این نقطه باعث متوقف شدن برنامه در هر بار فراخوانی تابع GetProcAddress می‌شود. اما فرض کنید شما می‌خواهید فقط زمانی GetProcAddress را متوقف سازید که به آن پارامتر RegSetValue عبور داده شود. در اینجا شرط ما محتوای اولین پارامتر عبور داده شده به پشته است.

نقاط توقف شرطی به عنوان نقاط توقف نرم‌افزاری پیاده‌سازی می‌شوند که دیباگر همیشه آن‌ها را دریافت می‌کند. دیباگر شرط‌ها را بررسی می‌کند و اگر یک شرط برقرار نباشد، به صورت خودکار اجرای برنامه را ادامه می‌دهد، بدون این‌که به کاربر هشدار دهد. دیباگرهای مختلف از شرط‌های متفاوتی استفاده می‌کنند.

نقاط توقف زمان بیشتری برای اجرا شدن نسبت به دستورالعمل‌های معمولی می‌گیرند و اگر شما یک نقطه توقف شرطی را روی دستورالعملی تنظیم کنید که اغلب مورد فراخوانی قرار می‌گیرد سرعت برنامه شما به شکل قابل توجه‌ای پایین خواهد آمد. در واقع، سرعت برنامه آنقدر پایین خواهد آمد که ممکن است هیچ‌گاه خاتمه نیابد. با این حال ما نگران پایین آمدن سرعت اجرای برنامه برای نقاط توقف غیرشرطی نخواهیم بود،

زیرا میزان کاهش سرعت برنامه در مقایسه با زمان بررسی وضعیت برنامه غیرضروری می‌باشد. به هر حال با وجود این ضعف، نقاط توقف شرطی هنگام بررسی بخشی از کدها می‌توانند مفید واقع شوند.

اکسپشن‌ها[1]

اکسپشن‌ها روش کلیدی هستند که یک دیباگر با استفاده از آن‌ها کنترل یک برنامه را به‌دست می آورد. البته فقط نقاط توقف اکسپشن تولید نمی‌کنند، بلکه خیلی از وضعیت‌های غیر مرتبط با دیباگ مانند دسترسی غیرمجاز به حافظه و خطای تقسیم بر صفر توسط نرم‌افزارها هم اکسپشن تولید می‌کنند.

با استناد به این موضوع، می‌توان دریافت که اکسپشن مخصوص بدافزارها، تحلیل بدافزار یا فرایند دیباگ نیستند. آن‌ها اغلب توسط باگ‌ها در نرم‌افزارها ایجاد می‌شوند، به همین دلیل معمولا دیباگرها آن‌ها را کنترل می‌کنند. همچنین اکسپشن‌ها می‌توانند برای کنترل جریان اجرای یک برنامه عادی مورد استفاده قرار گیرند. در اینجا یک ویژگی وجود دارد که اطمینان حاصل می‌کند که هم دیباگر و هم برنامه بتوانند از اکسپشن‌ها استفاده کنند.

اکسپشن‌های شانس اول و شانس دوم[2]

دیباگرها معمولا به منظور کنترل یک اکسپشن دو موقعیت ارائه می‌دهند که اکسپشن شانس اول و استثنا شانس دوم خوانده می‌شوند.

هنگامی‌که یک دیباگر به برنامه پیوست می‌شود و در این حین یک اکسپشن رخ می‌دهد، برنامه در حال دیباگ متوقف شده و دیباگر فرصت شانس اول را برای کنترل اکسپشن به برنامه ارائه می‌دهد. دیباگر می‌تواند اکسپشن را کنترل کرده یا آن را به خود برنامه ارسال کند. (هنگام دیباگ یک برنامه، باید تصمیم بگیرید که چگونه یک اکسپشن را کنترل کنید، حتی اگر این موضوع به کدی که مورد نظر شماست مرتبط نباشد.)

اگر یک برنامه دارای یک اکسپشن هندلر (Exception Handler) ثبت شده باشد، بعد از اینکه اکسپشن رخ دهد، این فرصت به برنامه داده می‌شود تا اکسپشن رخ داده را کنترل کند. به عنوان مثال، برنامه ماشین حساب سامانه‌عامل ویندوز می‌تواند به منظور کنترل خطای تقسیم بر صفر یک هندلر اکسپشن داشته باشد تا

[1] Exceptions
[2] First- and Second-Chance Exceptions

اگر برنامه همچین عملیاتی را انجام داد، هندلر اکسپشن می‌تواند به کاربر یک پیام خطا نشان داده و سپس اجرای برنامه را ادامه دهد. این وضعیتی بود که دیباگر به برنامه پیوست نشده باشد.

اگر یک برنامه کاربردی یک اکسپشن را کنترل یا هندل نکند، به دیباگر شانس کنترل اکسپشن داده می‌شود که این فرایند اکسپشن شانس دوم خوانده می‌شود. هنگامی‌که دیباگر اکسپشن شانس دوم را دریافت می‌کند به این معنی است که اگر دیباگر به برنامه پیوست نشده باشد، برنامه خراب یا به عبارت دیگر Crash خواهد کرد. لذا دیباگر باید اکسپشن رخ داده را دریافت و سپس رفع کند تا برنامه بتواند به اجرای عادی خود ادامه دهد در غیر اینصورت برنامه هنگام مواجه شدن با اکسپشن Crash خواهد کرد.

در زمان تحلیل بدافزار شما به دنبال باگ نمی‌گردید، بنابراین می‌توانید از کنار اکسپشن شانس اول عبور کرده و به آن توجه نکنید. (بدافزارها ممکن است اکسپشن شانس اول را ایجاد کنند که دیباگ برنامه را سخت کنند، این موضوع را در فصل‌های بعدی خواهید آموخت.) برخلاف اکسپشن شانس اول شما نمی‌توانید اکسپشن شانس دوم را نادیده بگیرید، زیرا برنامه قادر به ادامه اجرا نخواهد بود. در صورتی که شما باعث ایجاد اکسپشن شانس دوم در حین تحلیل بدافزار شوید، ممکن است باگی در بدافزار باشد که منجر به به خراب شدن بدافزار شود. اما این موضوع (اکسپشن شانس دوم) معمولا در یک بدافزار بدین دلیل اتفاق می‌افتد که بدافزار محیطی که در آن اجرا می‌شود را نمی پسندد.

اکسپشن‌های رایج

چندین اکسپشن رایج وجود دارند که هنگام اجرای دستورالعمل INT 3 رخ می‌دهند. گرچه سامانه‌عامل با آن‌ها همانند سایر اکسپشن‌ها برخورد می‌کند ولی دیباگرها دارای کد خاصی هستند تا بتوانند آن‌ها را کنترل کنند. در ادامه برخی از این اکسپشن‌ها را مورد بررسی قرار خواهیم.

به این نکته توجه داشته باشید، برنامه‌ها ممکن است دارای دستورالعمل‌های داخلی به منظور کنترل اکسپشن‌های INT 3 باشند، اما وقتی‌که یک دیباگر به برنامه پیوست شده باشد، این دیباگر است که شانس اول کنترل اکسپشن را دریافت می‌کند مگر اینکه خود دیباگر کنترل اکسپشن رخ داده را به برنامه عبور دهد، تا هندلر اکسپشن برنامه آن را کنترل کند.

مورد بعدی، دیباگ به روش گام به گام است. این روش دیباگ به عنوان یک اکسپشن در سامانه‌عامل پیاده‌سازی شده است. در معماری پردازنده‌ها یک پرچم وجود دارد که پرچم تله یا Trap Flag خوانده می‌شود

که در فصول قبلی درباره آن توضیحاتی ارائه کردیم. این پرچم وضعیت، برای انجام دیباگ گام به گام توسط یک برنامه استفاده می‌شود. هنگامی‌که پرچم تله با مقدار ۱ تنظیم شود، پردازنده یک دستورالعمل را اجرا کرده و سپس یک اکسپشن تولید می‌کند که دیباگر می‌تواند آن را دریافت کرده و سپس آن را کنترل کند.

مورد بعدی هنگامی رخ می‌دهد که یک برنامه تلاش می‌کند به قسمتی از حافظه دسترسی بگیرد که اجازه دسترسی گرفتن به آن را ندارد. در این شرایط یک اکسپشن از نوع تخطی دسترسی حافظه یا Memory-access violation رخ می‌دهد. این اکسپشن معمولا به دلیل نامعتبر (Invalid) بودن آدرس حافظه در سامانه‌عامل تولید می‌شود. همچنین این اکسپشن می‌تواند به دلیل قابل دسترس نبودن حافظه به موجب کنترل‌های امنیتی حافظه نیز صورت پذیرد.

مورد آخر اکسپشن‌های رایج توسط برخی از دستورالعمل‌های ممتاز رخ می‌دهند. این دستورالعمل‌ها از آنجایی که ممتاز یا Privileged هستند، تنها زمانی‌که پردازنده در حالت Privileged باشد، می‌توانند اجرا شوند. از همین روی، هنگامی‌که یک پردازنده تلاش به اجرای این نوع دستورالعمل‌ها در خارج از حالت ممتاز یا Privileged کند، پردازنده یک اکسپشن تولید خواهد کرد.

نکته : حالت Privileged همانند حالت کرنل بوده و حالت Nonprivileged مشابه حالت کاربر می‌باشد. اصطلاح Privileged و Nonprivileged بیشتر زمان‌ها که در مورد پردازنده صحبت می‌کنیم، استفاده می‌شوند. نمونه‌ای از دستورالعمل‌های Privileged می‌تواند نوشتن روی سخت‌افزار یا تغییر جدول‌های صفحه حافظه باشد.

تغییر مسیر اجرا با دیباگر

دیباگرها می‌توانند برای تعویض مسیر اجرای یک برنامه استفاده شوند. شما می‌توانید پرچم‌های کنترلی، اشاره‌گر دستورالعمل یا خود کد را تغییر دهید تا در نحوه اجرای برنامه تغییر ایجاد کنید.

به عنوان مثال، برای ممانعت از فراخوانی یک تابع، می‌توانید یک نقطه توقف جایی‌که تابع فراخوانی می‌شود، تنظیم کنید. سپس هنگامی‌که نقطه توقف اجرا شد، می‌توانید با تغییر اشاره‌گر دستورالعمل و تنظیم آن روی دستورالعمل پس از فراخوانی تابع، جلوی فراخوانی آن را بگیرید. در صورتی‌که این تابع مهم باشد، ممکن است

برنامه به درستی اجرا نشده و یا خراب شود. ولی اگر تابع تاثیری روی قسمت‌های دیگر برنامه نداشته باشد، برنامه می‌تواند بدون هیچ مشکلی به اجرای خود ادامه دهد.

شما همچنین می‌توانید از یک دیباگر برای تغییر اشاره‌گر دستورالعمل (EIP) استفاده کنید. به عنوان مثال، فرض کنید یک تابع با نام encodeString دارید که رشته‌ای را دریافت و دستکاری می‌کند، اما نمی‌توانید تشخیص دهید کجا encodeString فراخوانی می‌شود. در این شرایط می‌توانید از یک دیباگر برای اجرای یک تابع بدون این‌که بدانید کجا فراخوانی شده است، استفاده کنید. همچنین، به منظور دیباگ تابع encodeString برای این‌که بفهمید اگر رشته Hello World را به عنوان ورودی دریافت کند، چه اتفاقی رخ می‌دهد، می‌توانید مقدار esp+4 را با یک اشاره‌گر به رشته Hello World تنظیم کنید. سپس با تنظیم اشاره‌گر دستورالعمل به آدرس اولین دستورالعمل encodeString و مرحله به مرحله گام برداشتن در تابع مشاهده کنید که درون تابع چه اتفاقی رخ می‌دهد. مطمئنا این کار باعث می‌شود پشته برنامه تخریب شود و برنامه به درستی پس از اجرای تابع کار نکند. اما زمانی‌که بخواهید بفهمید بخش مشخصی از یک کد چگونه رفتار می‌کند این روش می‌تواند مفید باشد.

تغییر مسیر اجرا به صورت عملی

آخرین مثال از این فصل شامل ویروسی می‌شود که مطابق با زبان سامانه‌عامل عملیات‌های متفاوتی را انجام می‌دهد. اگر زبان رایانه با Simplified Chinese تنظیم شده باشد، بدافزار خودش را پاک می‌کند و هیچ آسیبی به سامانه نمی‌رساند. اگر زبان ماشین با English تنظیم شده باشد، ویروس یک پنجره باز کرده و پیام "You luck's so good." را در خروجی به نمایش می‌گذارد (البته زبان انگلیسی چینی‌ها خوب نیست و جمله انگلیسی را به غلط نوشتند، اصل جمله You are so Lucky است). اگر زبان ماشین با Japanese یا Indonesian تنظیم شده باشد، ویروس دیسک سخت را با داده‌های آشغال آنقدر بازنویسی می‌کند تا رایانه خراب شود. بگذارید بررسی کنیم چگونه همچین برنامه‌ای را می‌توان روی یک ماشین با زبان Japanese تحلیل کرد.

لیست ۶ کد اسمبلی کنترل تفاوت تنظیمات زبان این بدافزار را نمایش می‌دهد. بدافزار ابتدا تابع GetSystemDefaultLCID را فراخوانی می‌کند، سپس بر مبنای مقدار بازگشتی این تابع، یکی از آن

سه تابع را فرا می‌خواند. شناسه‌های محلی برای English ، Japanese، Indonesian و در نهایت Chinese به ترتیب مقادیر 0x0409، 0x0411 ، 0x0421 و0x0C04 هستند.

```
00411349   call    GetSystemDefaultLCID
0041134F ❶mov     [ebp+var_4], eax
00411352   cmp     [ebp+var_4], 409h
00411359   jnz     short loc_411360
0041135B   call    sub_411037
00411360   cmp     [ebp+var_4], 411h
00411367   jz      short loc_411372
00411369   cmp     [ebp+var_4], 421h
00411370   jnz     short loc_411377
00411372   call    sub_41100F
00411377   cmp     [ebp+var_4], 0C04h
0041137E   jnz     short loc_411385
00411380   call    sub_41100A
```

لیست ۶: اسمبلی تنظیمات مختلف زبان

اگر زبان سامانه English باشد، برنامه کد تابع موجود در آدرس 0x411037 را فراخوانی می‌کند، اگر زبان سامانه Japanese یا Indonesian باشد، برنامه تابع موجود در آدرس 0x41100F را فراخوانی خواهد کرد و در نهایت اگر زبان سامانه با Chinese تنظیم شده باشد، برنامه کد تابع موجود در آدرس 0x411001 را فراخوانی می‌کند. به منظور تحلیل این بدافزار، نیاز داریم این برنامه را زمانی اجرا کنیم که زبان سامانه روی Japanese یا Indonesian تنظیم شده باشد. اما می‌توانیم بدون این‌که تنظیمات سامانه را تغییر بدهیم با استفاده از یک دیباگر و گذاشتن نقطه توقف در قسمت (شماره ۱) مقدار بازگشتی را به آنچه که می‌خواهیم تغییر بدهیم. به طور خاص، اگر سامانه شما با زبان US English اجرا شده باشد، در ثبات EAX مقدار 0x0409 ذخیره خواهد شد که می‌توانید این مقدار ثبات EAX را در دیباگر به مقدار 0x411 تغییر بدهید و سپس برنامه را اجرا کنید تا تابع اصلی یا محموله عملیاتی بدافزار فراخوانی شود.

نتیجه‌گیری

دیباگ یک ابزار حیاطی برای جمع‌آوری اطلاعات از بدافزارهای مخربی است که به سختی می‌توان توسط دیزاسمبلی از آن‌ها اطلاعات به‌دست آورد. شما می‌توانید از یک دیباگر برای قدم برداشتن در کد برنامه‌ها به صورت پویا استفاده کنید و مشاهده کنید چه اتفاقی در هر بار اجرای یک دستورالعمل از برنامه روی سامانه می‌افتد. همچنین می‌توانید با استفاده از یک دیباگر روی بخش‌های مختلف برنامه نقطه توقف قرار بدهید و

آن قسمت از کدها را تحلیل کنید یا می‌توانید از یک دیباگر به منظور جمع‌آوری اطلاعات اضافی، مسیر اجرایی معمول یک برنامه را تغییر بدهید. در هر صورت تجزیه و تحلیل بدافزارها با استفاده از دیباگرها نیاز به تمرین بسیاری دارد. به همین دلیل در دو فصل بعدی ویژگی‌های مخصوص دو دیباگر OllyDBG و WinDBG را مورد بررسی قرار خواهیم داد.