قسمت ۵ – 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

تحلیل دیزاسمبلی در معماری x86

همان‌طور که در فصول گذشته بحث شد، متُدهای تجزیه و تحلیل بدافزار دینامیک و استاتیک ساده برای برداشتن اولین گام مناسب هستند، اما متاسفانه این متُدها اطلاعات کافی به ما برای تحلیل کامل بدافزار ارائه نمی‌دهند.

روش‌های تجزیه و تحلیل استاتیک ساده، شبیه مشاهده جسم خارجی یک بدن در طی عملیات کالبدشکافی است. از تجزیه و تحلیل استاتیک فقط می‌توانید برای به‌دست آوردن برخی از نتایج اولیه استفاده کنید، درحالی‌که برای دریافت اطلاعات کامل از بدافزار، نیاز دارید یک تحلیل کامل روی آن انجام بدهید. به عنوان مثال، ممکن است متوجه شوید یک تابع خاص وارد برنامه شده است، اما تا آن را کالبدشکافی نکنید، متوجه نخواهید شد که چگونه از این تابع در برنامه استفاده می‌شود.

روش‌های تجزیه و تحلیل دینامیک دارای یک سری کاستی‌هایی است. به عنوان مثال، تجزیه و تحلیل دینامیک ساده می‌تواند به شما بگوید بدافزار هنگامی‌که یک پاکت شبکه خاصی را دریافت می‌کند، چگونه به آن واکنش نشان می‌دهد، اما زمانی می‌توانید فرمت پاکت شبکه دریافت شده توسط بدافزار را شناسایی کنید که آن را مورد تجزیه و تحلیل عمیق قرار بدهید.

در این شرایط است که دیزاسمبلی برنامه به کمک شما می‌آید، در این قسمت از سری مقالات تجزیه و تحلیل بدافزار کی‌پاد این مسئله را مورد بررسی قرار خواهیم داد. دیزاسمبلی یک مهارت تخصصی است که می‌تواند برای افرادی که تازه وارد دنیای برنامه‌نویسی شده‌اند، بسیار دلهره‌آور باشد. اما دلسرد نشوید؛ این فصل یک درک ساده از هنر دیزاسمبلی به شما ارائه می‌دهد.

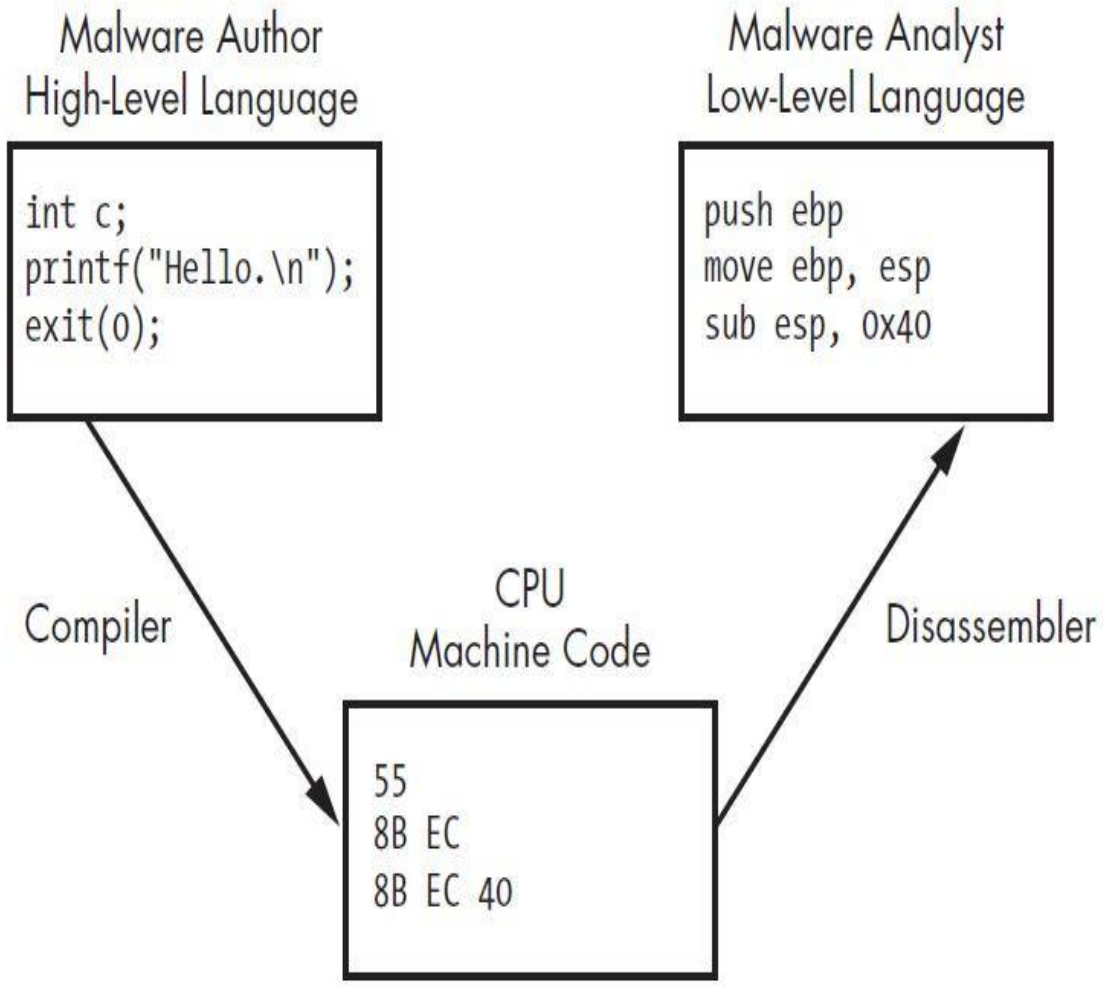
سطوح انتزاعی یا تجریدی[1]

در معماری سنتی کامپیوترها، یک سامانه کامپیوتری را می‌توان با سطوح انتزاعی مختلفی نشان داد که راهی ایجاد می‌کنند تا جزییات پیاده‌سازی سخت‌افزاری مخفی شوند. به عنوان مثال، می‌توانیم سامانه‌عامل ویندوز را روی سخت‌افزارهای گوناگون اجرا کنیم، زیرا اساس سخت‌افزار از سامانه‌عامل انتزاع داده شده است.

[1] Levels of Abstraction

تصویر ۱ سه سطح کدنویسی مختلف در تجزیه و تحلیل بدافزارها را نمایش می‌دهد. نویسندگان بدافزار، برنامه خود را با استفاده از یک زبان سطح بالا می‌نویسند و برای تبدیل کدهای برنامه خود به زبان ماشین به منظور اجرا توسط پردازنده از یک کامپایلر استفاده می‌کنند.

بالعکس توسعه‌دهندگان بدافزار، تحلیلگران بدافزار و کسانی که مهندسی‌معکوس انجام می‌دهند، در سطح زبان ماشین و اسمبلی کار می‌کنند؛ به همین دلیل، مجبور هستند از یک دیزاسمبلر[1] برای به‌دست آوردن کد اسمبلی برنامه کامپایل شده یا باینری استفاده کنند تا بتوانند برای تحلیل آن فایل باینری، کد اسمبلی آن را بخوانند و چگونگی عملکرد آن بدافزار را در سطح پایین مورد تحلیل قرار بدهند.



تصویر ۱: مثالی از سطوح مختلف کدها

[1] Disassembler

نکته: هنگامی‌که یک برنامه را دیزاسمبل می‌کنیم، برنامه اجرایی عموما با زبان اسمبلی در پانل دیزاسمبلر نمایش داده می‌شود. این عملیات را معکوس کردن کد اجرایی برنامه می‌نامند. زیرا در فرایند کامپایل، کد یک برنامه که با زبانی مانند C نوشته شده است، به زبان ماشین تبدیل می‌شود. اما در فرایند معکوس، فایل‌اجرایی یا به عبارتی کدماشین برنامه به یک سطح بالاتر (زبان اسمبلی) منتقل می‌شود تا درک کدهای آن نسبتا برای تحلیلگران قابل فهم‌تر شود. یکی از دلایلی هم که دیزاسمبل کردن کدهای C ممکن است،

تصویر ۱ یک مدل ساده از لایه‌های انتزاعی را نمایش می‌دهد، اما عموما سامانه‌های کامپیوتری امروزی با شش سطح لایه انتزاعی متفاوت توصیف می‌شوند که لیست این سطوح در ادامه آورده شده است. در این قسمت این سطوح را از پایین‌ترین سطح انتزاع فهرست کرده‌ایم. سطوح بالایی به همراه مفاهیم خاص در پایین این لیست قرار دارند، بنابراین هرچه پایین‌تر می‌رویم، قابلیت حمل آن سطوح در میان سیستم‌های کامپیوتری بالاتر می‌رود.

۱. **سخت‌افزار:** سطح سخت‌افزار[1]، تنها سطح فیزیکی، شامل مدارات الکتریکی است که اجرای ترکیبات پیچیده‌ای از اوپراتورهای منطقی مانند OR، XOR، AND و NOT را انجام می‌دهد. این عملیات‌ها با نام عملیات‌های منطقی دیجیتال[2] شناخته می‌شوند که به دلیل ماهیت فیزیکی خود نمی‌توانند به سادگی توسط نرم‌افزارها اعمال شوند.

۲. **میکروکد[3]:** سطح میکروکد همچنین به عنوان فریمور[4] شناخته می‌شود. میکروکد دقیقا فقط روی مداری که برای آن طراحی شده است، کار می‌کند. میکروکد شامل ریزدستورالعمل‌ها[5] می‌شود که به منظور ارائه یک رابط به سخت‌افزار ترجمه می‌شوند. هنگام انجام تجزیه و تحلیل بدافزار، معمولا

[1] Hardware level
[2] Digital Logic
[3] Microcode
[4] Firmware
[5] Microinstruction

درباره میکروکد اضطراب خاصی نداریم، زیرا آن‌ها برای هر سخت‌افزار کامپیوترای به صورت مخصوص نوشته می‌شوند.

۳. **زبان ماشین[1] :** سطح کد ماشین، شامل اُپکدها[2] می‌شود، آن کدها در مبنای هگزادسیمال نمایش داده می‌شوند و به پردازنده می‌گویند چه کاری شما می‌خواهید آن انجام بدهد. کد ماشین معمولا با چندین دستورالعمل میکروکد پیاده‌سازی می‌شود که سخت‌افزار می‌تواند کد را اجرا کند. همچنین قابل ذکر است، هنگامی‌که یک برنامه با زبان سطح بالا نوشته می‌شود، توسط کامپایلر کد ماشین آن تولید می‌شود.

۴. **زبان‌های سطح پایین[3] :** یک زبان سطح پایین نسخه قابل خواندن از مجموعه دستورالعمل‌های معماری (ISA) کامپیوتر است. رایج‌ترین زبان سطح پایین اکنون زبان اسمبلی است. به همین دلیل تحلیلگران بدافزار در سطح زبان اسمبلی عمل می‌کنند، زیرا کد ماشین برای انسان به سختی قابل درک است. در ادامه مشاهده خواهید کرد که به عنوان تحلیلگران بدافزار از یک دیزاسمبلر برای تولید کد اسمبلی برنامه‌های کامپایل شده استفاده می‌کنیم که خروجی آن بجای کدهای باینری شامل دستورالعمل‌های ساده و قابل درک از قبیل mov و jmp می‌شود. قابل ذکر است، زبان اسمبلی دارای الگوهای دستوری مختلفی است که ما در این کتاب فقط قواعد دستوری Intel را مورد بررسی قرار خواهیم داد.

نکته : هنگامی‌که سورس اصلی برنامه در دسترس نیست، در حالت معمول زبان اسمبلی بالاترین سطح زبانی است که می‌توان از کد کامپایل شده یک برنامه توسط ابزارهای دیزاسمبلر به‌دست آورد، البته برخی برنامه‌ها وجود دارند که عملیات دیکامپایل[4] فایل اجرایی یک برنامه را انجام می‌دهند، اما آنها دارای خطای زیادی در تولید کد نهایی هستند زیرا استخراج Context و Semantics از

---

[1] Machine code
[2] Opcodes
[3] Low-level languages
[4] Decompile

کدهای اسمبلی بسیار دشوار و در برخی شرایط غیرممکن است. قابل ذکر است، دیکامپایلرها برنامه‌هایی هستند که می‌توانند برنامه‌های کامپایل شده با یک زبان خاص را به زبان مبدأ آنها تولید کنند. دیکامپایل یک فایل اجرایی معکوس عملیات کامپایل کد منبع یک فایل اجرایی است.

---

۵. **زبان‌های سطح بالا[1] :** بیشتر برنامه‌نویس‌های رایانه در این سطح عمل می‌کنند. زبان‌های سطح بالا یک لایه انتزاعی قدرتمند از سطح ماشین ارائه می‌دهند که می‌توان در آن به سادگی از دستورات منطقی و دستورات کنترل جریان استفاده کرد. زبان‌های سطح بالا شامل C/C++، Delphi و دیگر زبان‌ها می‌شوند. این زبان‌ها معمولا از یک کامپایلر برای تبدیل کد خود به زبان ماشین استفاده می‌کنند که این فرایند با نام Compilation شناخته می‌شود.

۶. **زبان‌های مفسری[2] :** زبان‌های مفسری بالاترین سطح انتزاع را در این گروه دارا هستند. بیشتر برنامه‌نویس‌ها از زبان‌های مفسری مانند C#، Perl، Python و Java استفاده می‌کنند. کدهای موجود در این سطح از زبان مستقیما به کد ماشین کامپایل نمی‌شوند و بجای آن، کد این برنامه‌ها در گام اول به بایت کد[3] یا یک نمایش میانی[4] ترجمه می‌شوند. شایان ذکر است، بایت‌کدها خود همچنین یک نمایش میانی از زبان مفسری هستند که توسط یک مفسر اجرا می‌شوند. مفسرها می‌توانند بایت‌کدهای یک زبان را در هنگام اجرای برنامه به کد اجرایی ماشین تبدیل کنند. همچنین یک مفسر یک سطح انتزاعی خودکار را هنگام مقایسه کدهای کامپایل شده ارائه می‌دهد. زیرا می‌تواند درون خود کنترل خطا[5] و مدیریت حافظه[6] را به صورت مجزا از سامانه‌عامل انجام دهد.

---

[1] High-level languages
[2] Interpreted languages
[3] Byte Code
[4] Intermediate Representation
[5] Handle errors
[6] Memory management

چرا دیکامپایل برنامه‌های نیتیو دشوار است؟

همانطور که در بالا ذکر شد، از آنجایی که استخراج Context و Semantics از کدهای اسمبلی دشوار است، در نتیجه نمی‌توانیم به سادگی کدهای اسمبلی را تبدیل به کدهای سطح بالاتری مشابه C کنیم.

به هر صورت، وقتی با اسمبلی کدنویسی می‌کنید یا از یک فایل باینری نمایش دیزاسمبلی می‌گیرید، با یک سری Mnemonics رو به رو هستید که این Mnemonicsها می‌توانند در زمینه‌های (Context) اجرایی متغییر معنای متفاوتی هم داشته باشند.

مثلا Mov eax, 1 در یک موقعیت می‌تواند فقط معنای انتقال عدد ۱ به ثبات eax را داشته باشد، اما در جای دیگر می‌تواند معنای فراخوانی Syscall داشته باشد. از همین روی، وقتی شما با کدهای اسمبلی خام رو به رو می‌شوید، واقعا استخراج Semantics و Context یک بلاک کد سخت و دشوار است. در نهایت هم بحث بهینه‌سازی[1] بر روی آن بسیار دشوار خواهد بود. فرض کنید یک فایل دارید که ۱ میلیون خط کد اسمبلی است، بهینه‌سازی آن یک کار بسیار سخت و دشوار خواهد بود.

از همین روی بود که طراحان و برنامه‌نویسان پروژه Multics، وقتی پروژه Unix را شروع کردند، یک قسمتی از کار در این پروژه توسعه کامپایلر زبان C شد که در نهایت بتوانند از کدهای C برای استخراج Context و Semantics استفاده کنند و در نهایت عمل بهینه‌سازی را به شکل صحیح و با دشواری حداقلی انجام بدهند. والا انجام عمل بهینه‌سازی بر روی خود کدهای اسمبلی به تنهایی واقعا دشوار است.

به هر صورت، از آنجایی که یک خط C شاید به چند دستور اسمبلی تبدیل شود، گرفتن نمایش معکوس از آن بسیار دشوار است. وقتی از مفهوم استخراج Semantics استفاده می‌کنیم، یعنی اینکه بدون توجه به مفهوم معنی دار آن کارها، نمی‌توانید صرفا با خواندن یه دستور اسمبلی تصمیم بگیرید که آن بهینه‌سازی کنید یا نه.

کارهای دم دستی و کوچک را می‌توانید انجام بدهید (هر چند بعضی مواقع همین مورد هم با دیده تردید باید نگاه شود)، مثل تغییر add به lea، اما این فقط بهینه‌سازی در سطح دستور است، نه بیشتر. بنابراین نمی‌توان آن را بهینه‌سازی دانست، هر چند مواقعی ممکن است وجود داشته باشند که همان بهینه‌سازی‌هایی که مثال

[1] Optimization

زده شد، بر روی دستورات صورت نگیرد. خلاصه علاوه بر سطح دستور، بهینه‌سازی در چندین سطح دیگر هم داریم. مثلا بهینه‌سازی در سطح Basic Blockها که بدون استخراج Semantics چنین چیزی نشدنی است.

مهندسی معکوس[1]

هنگامی‌که بدافزار روی یک دیسک سخت ذخیره می‌شود، معمولا دارای قالب باینری خواهد بود که شامل کدهای ماشین است. به عبارت دیگر، یک فایل باینری چیزی جز یک مجموعه از کدهای باینری (۰ و ۱) نیست که این کدهای باینری برای پردازنده دارای معنا و مفهوم هستند.

همان‌طور که بحث شد، کد ماشین قالبی است که پردازنده می‌تواند آن را با سرعت و بسیار کارآمد اجرا کند. هنگامی‌که ما یک بدافزار را دیزاسمبل می‌کنیم (همان‌طور که در تصویر ۱ نمایش داده شده است) فایل باینری بدافزار را به عنوان ورودی به دیزاسمبلر ارائه می‌دهیم و در خروجی کد اسمبلی آن را دریافت می‌کنیم، معمولا این کار را با یک دیزاسمبلر قدرتمند انجام می‌دهیم. (در قسمت بعدی، در مورد یکی از مشهورترین دیزاسمبلرها یعنی IDA بحث خواهیم کرد.)

زبان اسمبلی در واقع یک کلاس از زبان‌ها است. هر یک از قواعد دستوری اسمبلی معمولا برای برنامه‌نویسی در یک ریزپردازنده خاص مورد استفاده قرار می‌گیرد؛ از قبیل معماری x86، x64، SPARC، PowerPC ، MIPS و ARM که در این بین معماری x86 و x64 مشهورترین معماری‌های استفاده شده در سامانه‌های خانگی هستند.

اکثریت کامپیوترهای شخصی ۳۲ بیتی بر مبنای معماری x86 هستند. این معماری با عنوان Intel IA-32 همچنین شناخته می‌شود و بیشتر نسخه‌های مدرن ۳۲ بیتی ویندوز مایکروسافت برای اجرا روی معماری x86 طراحی می‌شوند. علاوه بر این، اکثریت معماری‌های Amd64 و Intel64 که بر روی آن‌ها سیستم‌عامل ویندوز در حال اجرا است، از باینری‌های ۳۲ بیتی ویندوز همچنین پشتیبانی می‌کنند.

به همین دلیل، چون بیشتر بدافزارها برای معماری x86 کامپایل و ترجمه می‌شوند، لذا تمرکز نگارش این کتاب روی بدافزارهای کامپایل شده برای این معماری خواهد بود (در قسمت‌های بعدی این مقاله البته به

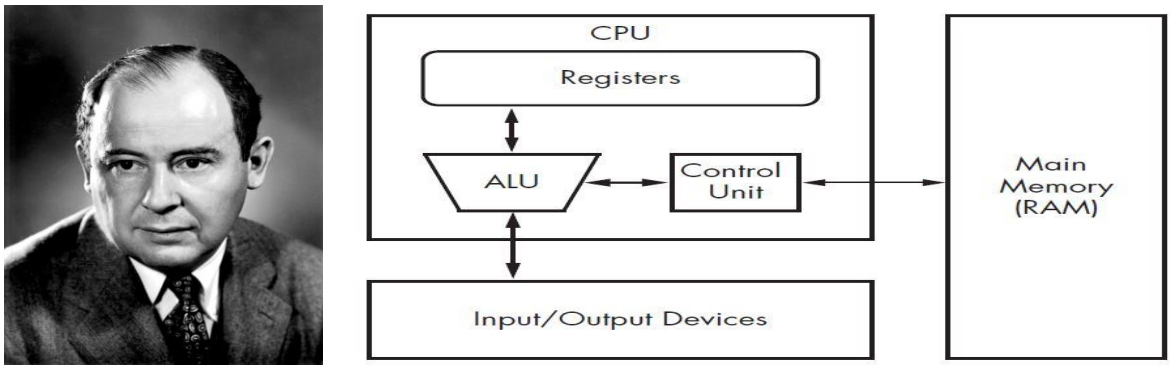
[1] Reverse-Engineering

تحلیل بدافزارهای کامپایل شده برای معماری Intel64 هم خواهیم پرداخت). در این قسمت ما به جنبه‌های معماری x86 خواهیم پرداخت که در طی تجزیه و تحلیل بدافزار با آن‌ها مواجه خواهید شد.

نکته : برای به‌دست آوردن اطلاعات بیشتر در مورد زبان برنامه‌نویسی اسمبلی با قواعد دستوری Intel نسخه هفتم کتاب Assembly Language for x86 Processors نوشته Kip Irvine از انتشارات Pearson را بخوانید و همچنین به منظور آشنایی با برنامه اسمبلی با قواعد دستوری At&t در سامانه‌عامل لینوکس کتاب Professional Assembly Programming نوشته Richard Blum از انتشارات Wiley را مورد مطالعه قرار بدهید. شایان ذکر است، برای درک خود معماری پردازنده‌های اینتل بعد آشنایی کلی با اسمبلی بهتر است راهنمای توسعه‌دهندگان اینتل یا همان Intel Software Developer's Manual یا به اختصار SDM را مطالعه کنید.

معماری x86

ساختار درونی بیشتر معماری‌های کامپیوتر مدرن (از جمله معماری x86) از مدل جان فون نویمان[1] ریاضی‌دان و دانشمند آمریکایی مجاری الاصل پیروی می‌کنند که در تصویر ۲ این مدل به نمایش گذاشته شده است. معماری نویمان دارای سه مولفه سخت‌افزاری است:



تصویر ۲: معماری Von Neumann

[1] John von Neumann

۱. واحد پردازشگر مرکزی[1] که کدها را اجرا می‌کند.
۲. حافظه اصلی[2] سامانه که داده‌های اطلاعاتی و کدها را ذخیره‌سازی می‌کند.
۳. سامانه ورودی و خروجی[3] که رابط دستگاهی از قبیل دیسک سخت، صفحه‌کلید و صفحه نمایش است.

همان‌طور که در تصویر ۲ مشاهده می‌کنید، پردازنده شامل چندین مولفه مختلف می‌شود: واحد کنترل[4] دستورالعمل‌ها را از حافظه با استفاده از یک ثبات (ثبات اشاره‌گر دستورالعمل[5]) می‌گیرد. این ثبات آدرس دستورالعمل‌هایی که باید اجرا شوند، در خود ذخیره می‌کند. ثبات‌ها[6] واحدهای ساده ذخیره‌سازی داده‌های اطلاعاتی برای پردازنده هستند و اغلب به منظور بهینه‌سازی زمان پردازنده برای دسترسی از حافظه اصلی مورد استفاده قرار می‌گیرند. واحد محاسبه و منطق[7] دستورات واکشی شده از حافظه را اجرا کرده و نتایج اجرای آن دستورات را در حافظه یا ثبات‌ها قرار می‌دهد. قابل ذکر است، هنگامیکه یک برنامه اجرا می‌شود، فرآیند واکشی و اجرای دستورالعمل‌ها تکرار می‌شود.

نکته : معماری جان فون نویمان تنها معماری نیست که در صنعت طراحی پردازنده‌ها مورد استفاده قرار می‌گیرد. معماری دیگری هم وجود دارد که با نام Harvard Architecture معروف است. در معماری Harvard، برای داده‌ها ودستورالعمل‌ها، حافظه و گذرگاه داده جداگانه‌ای در نظر گرفته شده است. این نام در پی ساخت رایانه Harvard Mark I که از حافظه‌های جداگانه برای داده‌ها و دستورالعمل‌ها استفاده می‌کرد به این معماری اطلاق شده است. در اغلب کامپیوترهای امروزه از تعدیل شده مدل Harvard یعنی مدل Modified Harvard Architecture استفاده می‌شود. در این مدل حافظه نهان مربوط به

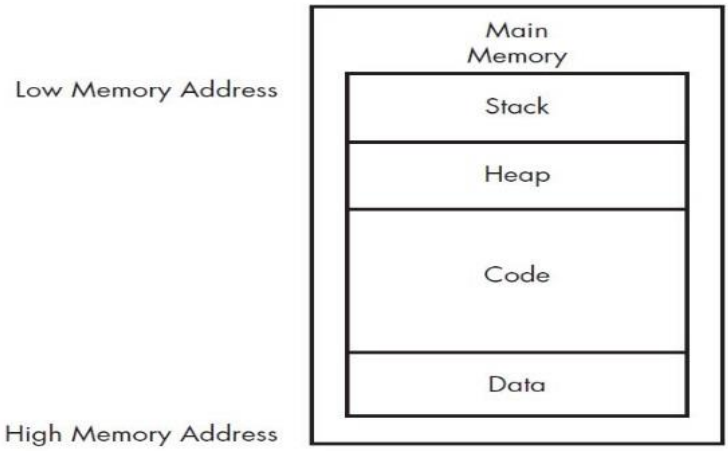
---

[1] Central Processing Unit
[2] Main Memory
[3] input/output system
[4] Control unit
[5] Instruction pointer
[6] Registers
[7] Arithmetic logic unit

دستورالعمل‌ها و داده‌ها مجزا هستند، به نوعی می‌توان این شیوه را ترکیبی از دو نوع معماری نویمان و هاروارد دانست. چنین شیوه‌ای اکنون در پردازنده‌های x86، ARM، PowerPC و MIPS مورد استفاده قرار می‌گیرد.

حافظه اصلی[1]

حافظه اصلی (RAM) برای یک برنامه در حال اجرا بر روی سامانه‌عامل واحد می‌تواند به چهار قسمت آورده شده در تصویر ۳ تقسیم شود.



تصویر ۳: طرح ساده حافظه برای یک برنامه

۱. **داده[2] :** این اصطلاح می‌تواند برای اشاره به یک بخش خاص از حافظه که Data Section نامیده می‌شود، مورد استفاده قرار گیرد که برنامه هنگامیکه بارگزاری می‌شود، مقادیر داده‌ای آن در این قسمت قرار می‌گیرد. به صورت خلاصه، در این قسمت داده‌های اطلاعاتی برنامه ذخیره می‌شوند. این مقادیر گاهی اوقات مقادیر استاتیک[3] خوانده می‌شوند، زیرا در طی اجرای برنامه تغییر نمی‌کنند، یا ممکن است مقادیر عمومی[4] خوانده شوند، زیرا از تمامی قسمت‌های برنامه قابلیت فراخوانی دارند.

[1] Main Memory
[2] Data
[3] Static values
[4] Global values

۲. **کد :** بخش کد شامل دستورالعمل‌های اجرایی برنامه است. قسمت کد آنچه که برنامه می‌خواهد انجام دهد را کنترل می‌کند و وظایف آن را هماهنگ می‌سازد.

۳. **حافظه Heap :** از حافظه Heap به عنوان یک حافظه دینامیک در طی اجرای برنامه استفاده می‌شود. Heap برای اختصاص حافظه به مقادیر جدید و حذف مقادیر (آزادسازی حافظه) که برنامه دیگر به آنها نیاز ندارد، مورد استفاده قرار می‌گیرد. حافظه Heap به یک حافظه دینامیک اشاره دارد، زیرا محتویات آن پیوسته می‌توانند در حین اجرای برنامه عوض شوند.

۴. **پشته[1] :** حافظه پشته برای ذخیره‌سازی متغیرهای محلی و پارامترهای توابع استفاده می‌شود. این حافظه همچنین به جریان اجرای برنامه کمک می‌کند. حافظه پشته را با جزییات دقیق در این قسمت از سلسله مقالات تحلیل بدافزار کی‌پاد تشریح خواهیم کرد.

گرچه دیاگرام نمایش داده شده در تصویر ۳ چهار بخش اصلی حافظه را در عمل نمایش می‌دهد، ولی با این حال این قسمت‌ها ممکن است در سراسر حافظه قرار داده شوند. به عنوان مثال، هیچ تضمینی وجود ندارد که حافظه پشته در زیر قسمت کد یا دیگر قسمت‌ها قرار گیرد.

دستورالعمل‌ها یا اینستراکشن‌ها[2]

دستورالعمل‌ها اسمبلی، بلوک‌های اصلی یک برنامه اسمبلی هستند. در اسمبلی معماری x86، یک دستورالعمل از یک mnemonic و یک یا هیچ اوپرندی[3] ساخته می‌شود. همان‌طور که در جدول ۱ نمایش داده شده است، mnemonic یک واژه است که عملیات دستورالعمل را مشخص می‌کند، از قبیل mov و jmp که داده‌ها و جریان داده را انتقال می‌دهند. اوپرندها معمولا برای مشخص کردن اطلاعات مورد نیاز دستورالعمل‌ها مانند ثبات‌ها و داده‌ها استفاده می‌شوند.

جدول ۱: قالب دستورالعمل‌های اسمبلی

Mnemonic	Destination operand	Source operand
mov	ecx	0x42

[1] Stack
[2] Instructions
[3] Operands

در دستور بالا، mov به عنوان یک Mnemonic و ثبات ecx و مقدار 0x42 دو اوپرند برای این Mnemonic هستند. لذا وقتی درباره Mnemonicها حرف زده می شود، منظور کدهایی مشابه mov و add و lea و ... در اسمبلی است. همچنین وقتی حرف از اوپرندها می‌شود، منظور مقادیری است که به mnemonicها عبور داده می‌شوند. شایان ذکر است، به این ترتیب Mnemonic و Operandها در اسمبلی یک دستورالعمل یا Instruction می‌گویند که این دستورالعمل‌ها برای هر پردازنده دارای معماری و ساختار متفاوتی از یکدیگر هستند.

کدهای عملیاتی و اندیان‌ها [1]

در قسمت قبل بررسی کردیم که Mnemonicها و Operandها با هم چه تفاوتی دارند و چطور یک دستورالعمل یا Instruction را ایجاد می‌کنند. حال نکته دیگر این است که اپکدها یا کدهای عملیاتی ماشین چه چیزی هستند؟ در تصویر ۴ مشاهده می کنید که هر دستورالعمل اسمبلی دارای یک نمایش در قالب کدهای هگزادسیمال است. این کدهای هگزادسیمال هر کدام معرف یک دستورالعمل هستند. مثلا اپکد 33C0 معادل xor eax, eax است.

```
EIP  7748EA  EB 07              jmp ntdll.7748EAAC
     7748EA  33C0               xor eax,eax
     7748EA  40                 inc eax
     7748EA  C3                 ret
     7748EA  8B65 E8            mov esp,dword ptr ss:[ebp-18]
     7748EA  C745 FC FEFFFFFF   mov dword ptr ss:[ebp-4],FFFFFFFE
     7748EA  8B4D F0            mov ecx,dword ptr ss:[ebp-10]
     7748EA  64:890D 00000000   mov dword ptr fs:[0],ecx
     7748EA  59                 pop ecx
     7748EA  5F                 pop edi                          edi:"LdrpInitializeProcess"
     7748EA  5E                 pop esi
     7748EA  5B                 pop ebx
     7748EA  C9                 leave
     7748EA  C3                 ret
     7748EA  64:A1 30000000     mov eax,dword ptr fs:[30]
     7748EA  33C9               xor ecx,ecx
```

تصویر ٤: دستورالعمل‌های اسمبلی به همراه اپکدهای اینتل

به هر صورت، به ترتیب کدهایی که در پانل سمت چپ دستورات اسمبلی در تصویر ۴ مشاهده می‌کنید، اپکدهای پردازنده می‌گویند که فقط توسط خود پردازنده هم قابل فهم هستند. برای اطلاعات بیشتر درباره

[1] Opcodes and Endianness

این اپکدها می‌توانید راهنمای اینتل را بخوانید. به عنوان مثال، در تصویر ۵، اپکدها دستورالعمل mov نمایش داده شده است که دارای قالب‌های گوناگون هستند.

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8*.
REX + 88 /r	MOV *r/m8***,r8****	MR	Valid	N.E.	Move *r8* to *r/m8*.
89 /r	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16*.
89 /r	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32*.
REX.W + 89 /r	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64*.
8A /r	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8*.
REX + 8A /r	MOV *r8***,r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8*.
8B /r	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16*.
8B /r	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32*.
REX.W + 8B /r	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64*.
8C /r	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r/m16*.
8C /r	MOV *r16/r32/m16, Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r16/r32/m16*.
REX.W + 8C /r	MOV *r64/m16, Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r64/m16*.
8E /r	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment register.
REX.W + 8E /r	MOV *Sreg,r/m64***	RM	Valid	Valid	Move *lower 16 bits of r/m64* to segment register.
A0	MOV AL,*moffs8**	FD	Valid	Valid	Move byte at (*seg:offset*) to AL.
REX.W + A0	MOV AL,*moffs8**	FD	Valid	N.E.	Move byte at (*offset*) to AL.

تصویر ۵: راهنمای اپکدهای اینتل **Mov**

این اپکدها به پردازنده می‌گویند، برنامه در عمل چه کاری قرار است، انجام بدهد. در این سلسله مقالات و البته دیگر کتاب‌ها و مقالات از واژه اپکد برای تمامی دستورالعمل‌های ماشین استفاده شده است، اگرچه اینتل آن را از لحاظ فنی با دقت بیشتری تعریف کرده است. شایان ذکر است، برنامه‌های دیزاسمبلر اپکدها را به دستورالعمل‌های قابل درک برای انسان ترجمه می‌کنند.

جدول ۲: کدهای عملیاتی دستورالعمل‌ها

Instruction	mov ecx,	0x42
Opcodes	B9	42 00 00 00

به عنوان مثال، در جدول ۲ مشاهده می‌کنید که دستورالعمل mov ecx, 0x42 برابر با اپکد B9 42 00 00 00 است. دستورالعمل mov ecx برابر با B9 و 0x42 برابر با مقدار 0x42000000 می‌باشد. قابل ذکر است، دیزاسمبلرها برای تبدیل اپکدها به دستورالعمل‌های اسمبلی از همین رویکرد استفاده می‌کنند. به

عبارت دیگر، یک دیزاسمبلر توالی اپکدها را می‌خواند و مبتنی بر اینکه با چه اپکدی رو به رو شده است، آن را به نمایش متانظر اسمبلی خود تبدیل می‌کنند.

همچنین قابل ذکر است، مقدار 0x42000000 مانند مقدار 0x42 رفتار خواهد کرد، زیرا معماری x86 از قالب لیتل اِندیان[1] استفاده می‌کند. روش اندیان‌ها قرارگیری داده‌ها را در بایت‌های حافظه تشریح می‌کنند که با ارزشترین یا بی‌ارزشترین بایت داده ابتدا در حافظه ذخیره شود یا خیر.

تعویض قالب اندیان‌ها چیزی است که بدافزارها در حین ارتباط با شبکه انجام می‌دهند. زیرا داده‌های شبکه از بیگ اِندیان[2] استفاده می‌کنند، در حالی‌که یک برنامه بر پایه معماری x86 از لیتل اِندیان استفاده می‌کند. به عنوان مثال، IP آدرس 127.0.0.1 در قالب بیگ اِندیان (بر روی شبکه) به شکل 0x7F000001 نمایش داده می‌شود و در حافظه با قالب لیتل اِندیان به شکل 0x0100007F نمایش داده خواهد شد. به عنوان یک تحلیلگر بدافزار، باید با اندیان‌ها آشنا باشید تا تصادفاً نشانه‌های مهم را اشتباها مانند یک آدرس IP معکوس نکنید.

اوپرندها[3]

همانطور که پیش از این ذکر شد، اوپرندها در دستورالعمل‌ها برای مشخص کردن داده‌های مورد نیاز دستورالعمل‌های اسمبلی استفاده می‌شوند. شایان ذکر است، در برنامه‌هایی که با زبان اسمبلی نوشته می‌شوند، سه نوع اوپرند فقط می‌توانند مورد استفاده قرار گیرند که در لیست زیر توضیح داده شده‌اند:

۱. اوپرندهای فوری[4] که مقادیر ثابت دارا هستند، از قبیل مقدار 0x42 که در جدول ۲ نمایش داده شده است.
۲. اوپرندهای ثباتی[5] که به ثبات‌های پردازنده از قبیل ecx که در جدول ۲ نمایش داده شده اشاره دارند.

[1] Little-Endian
[2] Big-Endian
[3] Operands
[4] Immediate operands
[5] Register operands

۳. اوپرندهای آدرس حافظه[1] به یک آدرس حافظه اشاره دارند که شامل یک مقدار خاص می‌شود. معمولا این مقدار توسط یک مقدار مشخص، ثبات یا یک معادله بین براکت، مانند [EAX] نمایش داده می‌شود.

ثبات‌ها یا رجیستری[2]

ثبات‌ها فضاهای کوچکی برای ذخیره‌سازی داده‌ها هستند که در پردازنده تعبیه شده‌اند. محتویات ثبات‌ها با سرعت بسیار بالایی نسبت به حافظه اصلی می‌توانند در دسترسی قرار گیرند. پردازنده‌های x86 یک مجموعه از ثبات‌ها برای ذخیره‌سازی موقت داده‌ها یا فضای‌کاری دارند. جدول ۳ رایج‌ترین ثبات‌های معماری x86 را نشان می‌دهند که در چهار گروه زیر قرار می‌گیرند.

۱. ثبات‌های عمومی[3] که توسط پردازنده در طی اجرای برنامه مورد استفاده قرار می‌گیرند.
۲. ثبات‌های سگمنت[4] که برای دنبال کردن قسمت‌های مختلف حافظه مورد استفاده قرار می‌گیرند.
۳. ثبات‌های وضعیت[5] که برای تصمیم‌گیری مورد استفاده قرار می‌گیرند.
۴. ثبات اشاره‌گر دستورالعمل[6] که آدرس دستورالعملی که باید توسط پردازنده اجرا شود، نگه می‌دارد.

می‌توانید از جدول ۳ به عنوان یک مرجع در طول این سلسله مقالات تحلیل بدافزار استفاده کنید و همواره به آن برای مشاهده دسته‌بندی ثبات‌ها رجوع کنید. در ادامه همین قسمت هر یک از این ثبات‌ها را با جزییات دقیق مورد بررسی قرار خواهیم داد.

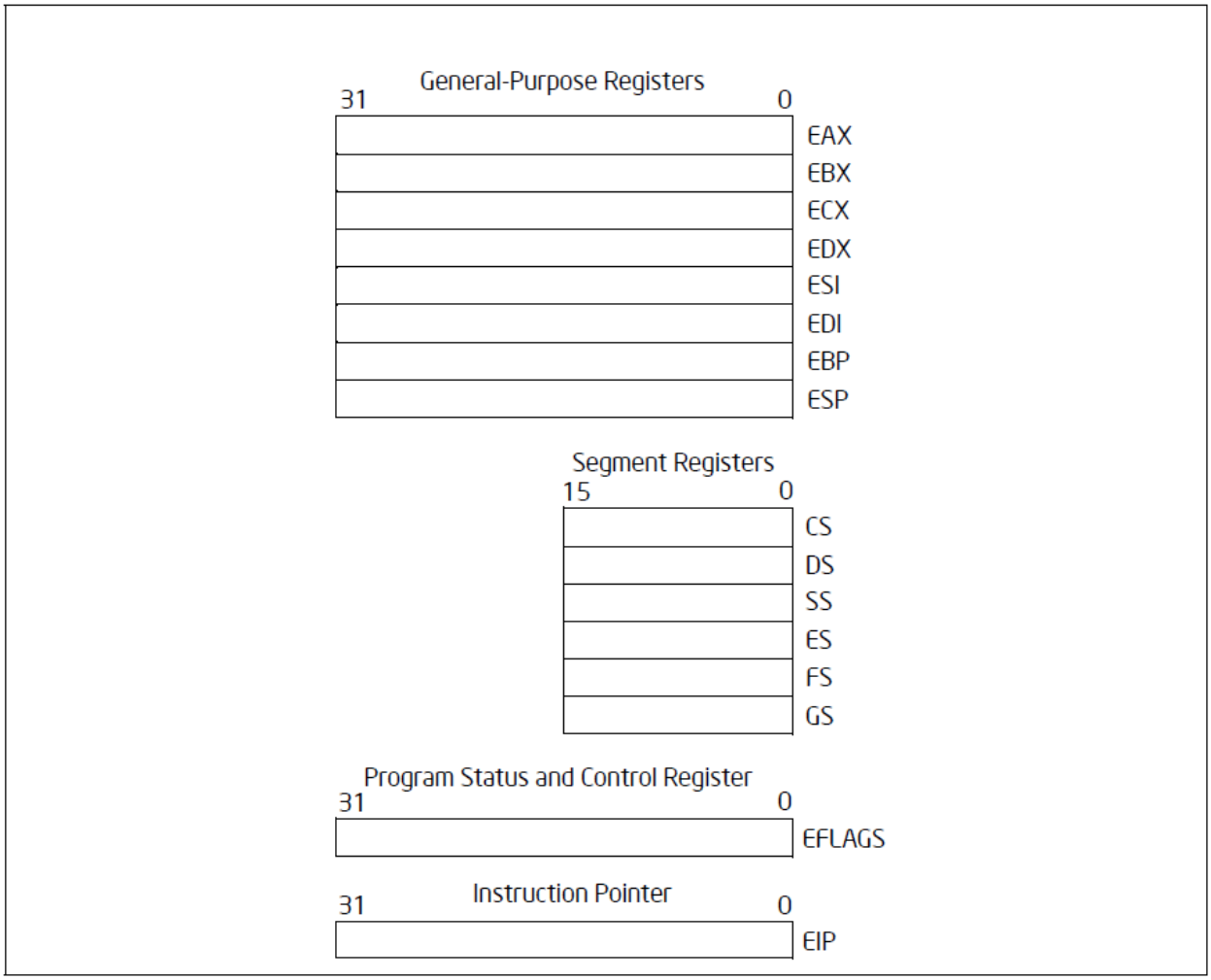
جدول ۳: ثبات‌ها در معماری x86 اینتل

General registers	Segment registers	Status register	Instruction pointer
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		

[1] Memory address operands
[2] Registers
[3] General registers
[4] Segment registers
[5] Status flags
[6] Instruction pointer

ESP (SP) ESI (SI)	GS		

اگر با زبان اسمبلی برنامه‌نویسی کرده باشید، به احتمال زیاد می‌دانید که تمامی ثبات‌های عمومی معماری x86 دارای ۳۲ بیت اندازه هستند. با این حال برنامه‌نویس می‌تواند در کدهای اسمبلی از ۳۲ بیت یا ۱۶ بیت و یا ۸ بیت فضای حافظه این ثبات‌ها استفاده کند. به عنوان مثال، ثبات EDX یک ثبات ۳۲ بیتی است (تمامی ثبات‌هایی که با حرف E شروع می‌شوند ۳۲ بیتی هستند) و ثبات DX، ۱۶ بیت کم ارزش ثبات توسعه داده شده EDX است.



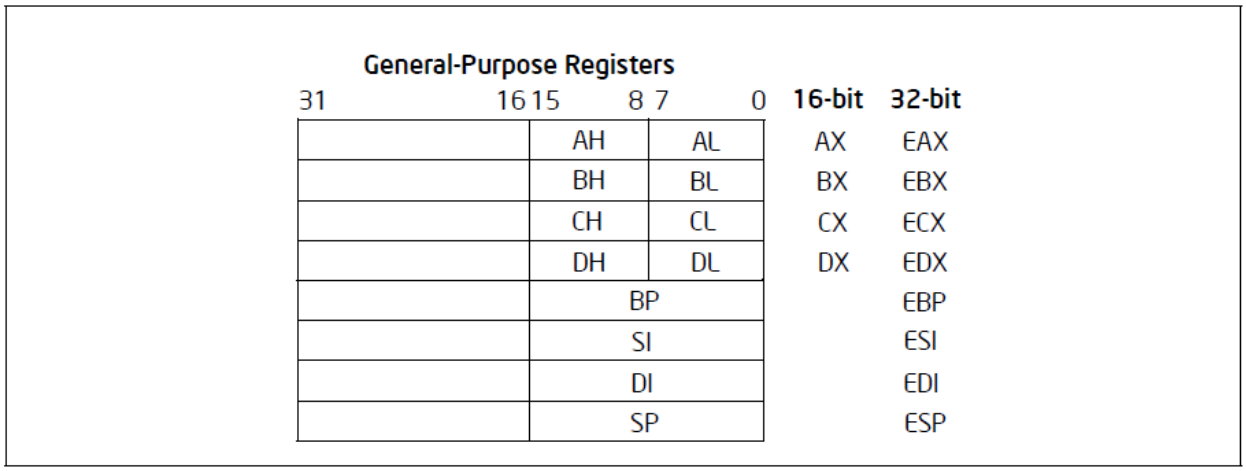
تصویر ٦: تصویری از اندازه ثبات‌های معماری **x86** اینتل

همچنین می‌توان به ۳ ثبات (EAX، EBX، ECX) به عنوان ثبات‌های ۸ بیتی یا ۱۶ بیتی همانند EDX رجوع کرد. به عنوان مثال همان‌طور که در تصویر ۷ مشاهده می‌کنید، ثبات AL برای ارجاع به ۸ بیت کم ارزش و از AH برای ۸ بیت قسمت دوم و از ثبات AX برای ارجاع به ۱۶ بیت ثبات EAX استفاده می‌شود.

جدول ۳ لیست تمامی ارجاعات ممکن به ثبات‌های عمومی را نمایش داده است. همان‌طور که مشاهده می‌کنید، در این مثال ثبات EAX به قسمت‌های کوچکتری شکسته شده است و اجزای شکسته شده آن به شکل (AX, AH, AL) به نمایش گذاشته شده است.

ثبات‌های عمومی

ثبات‌ها عمومی معمولا داده‌ها و آدرس‌های حافظه را در خود ذخیره می‌کنند و اغلب اوقات به جای یکدیگر برای دریافت عملیات‌های انجام گرفته درون برنامه مورد استفاده قرار می‌گیرند. به هر حال، علیرغم این‌که آن‌ها را ثبات‌های عمومی می‌خوانند، با این حال، آن‌ها همیشه در این راه مورد استفاده قرار نمی‌گیرند.



تصویر ۷: اندازه گوناگون ثبات‌های x86

برخی از دستورالعمل‌های معماری x86 از ثبات‌های خاص در تعریف خود استفاده می‌کنند. به عنوان مثال، دستورالعمل‌های ضرب و تقسیم همیشه از ثبات‌های EAX و EDX استفاده می‌کنند. علاوه بر تعریف دستورالعمل‌ها، ثبات‌های عمومی می‌توانند در یک مدل سازگار[1] در طی کامپایل کد استفاده شوند.

[1] Consistent fashion

استفاده از ثبات‌های عمومی در یک مدل سازگار طی کامپایل کد یک قرارداد[1] شناخته می‌شود. داشتن یک درک عمیق درباره قراردادهای استفاده شده توسط کامپایلرها به تحلیلگرهای بدافزار اجازه می‌دهند به سرعت یک کد را تحلیل کنند. زیرا زمان خود را در بررسی و چگونگی استفاده محتوای یک ثبات هدر نمی‌دهند.

به عنوان مثال، ثبات EAX معمولا شامل مقادیر بازگشتی از توابع فراخوانی شده می‌شود. از همین روی اگر بلافاصله بعد از فراخوانی یک تابع از ثبات EAX استفاده شد، می‌توانید مقدار بازگشتی تابع را در ثبات EAX مشاهده کنید.

پرچم‌ها[2]

ثبات EFLAGS، ثبات وضعیت اجرای دستورالعمل‌ها است. در معماری x86، اندازه این ثبات ۳۲ بیت است و هر بیت یک پرچم می‌باشد. در حین اجرا، هر پرچم دارای مقدار ۱ یا مقدار ۰ می‌شود و برای کنترل عملیات پردازنده یا نمایش نتیجه اجرای یک دستورالعمل و عملیات پردازنده مورد استفاده قرار می‌گیرند. پرچم‌های جدول ۴ مهم‌ترین پرچم‌های وضعیت برای تحلیل بدافزارها هستند.

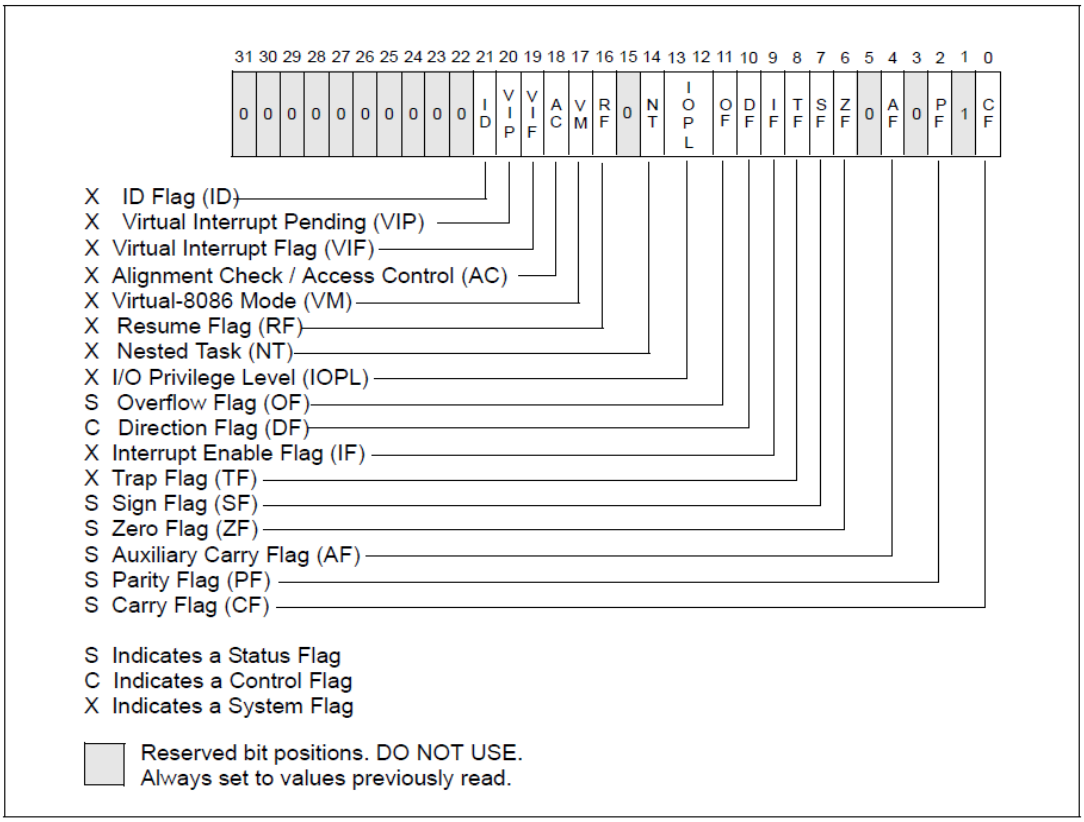
جدول ۴: تشریح ثبات‌های پرچم وضعیت

نام پرچم	توضیحات
پرچم صفر[3]	این پرچم وضیعت، نتیجه یک عمل محاسباتی یا مقایسه‌ای را نشان می‌دهد. (مقدار ۰ این پرچم نشان‌دهنده نتیجه غیرمساوی عملیات محاسباتی است و مقدار ۱ این پرچم نشان‌دهنده نتیجه مساوی عملیات محاسباتی است)
پرچم نقلی[4]	پرچم نقلی زمانی تنظیم می‌شود که نتیجه یک عملیات برای اوپرند مقصد خیلی بزرگ یا خیلی کوچک باشد؛ در غیر اینصورت این پرچم تنظیم نخواهد شد.
پرچم علامت[5]	پرچم علامت زمانیکه حاصل یک عملیات محاسباتی منفی شود، تنظیم می‌گردد. این پرچم همچنین زمانیکه پس از یک عملیات محاسباتی با ارزش‌ترین بیت تنظیم شود، تنظیم می‌گردد.

[1] Convention
[2] Flags
[3] Zero Flag
[4] Carry Flag
[5] Sign Flag

پرچم تله[1]	در معماری x86 اگر این پرچم با مقدار ۱ تنظیم شود، اجازه می‌دهد که عمل پردازشگر در یک حالت تک مرحله‌ای انجام گیرد (اجرا یک دستورالعمل در هر ثانیه). دیباگرهایی از قبیل DEBUG ویندوز، پرچم تله را با ۱ مقدار می‌دهند، به طوری که با اجرای هر دستور از برنامه، می‌توان اثر آن را روی حافظه و ثبات‌ها مشاهده کرد. در حالت کلی این پرچم در دیباگ کردن برنامه‌ها مورد استفاده قرار می‌گیرد.

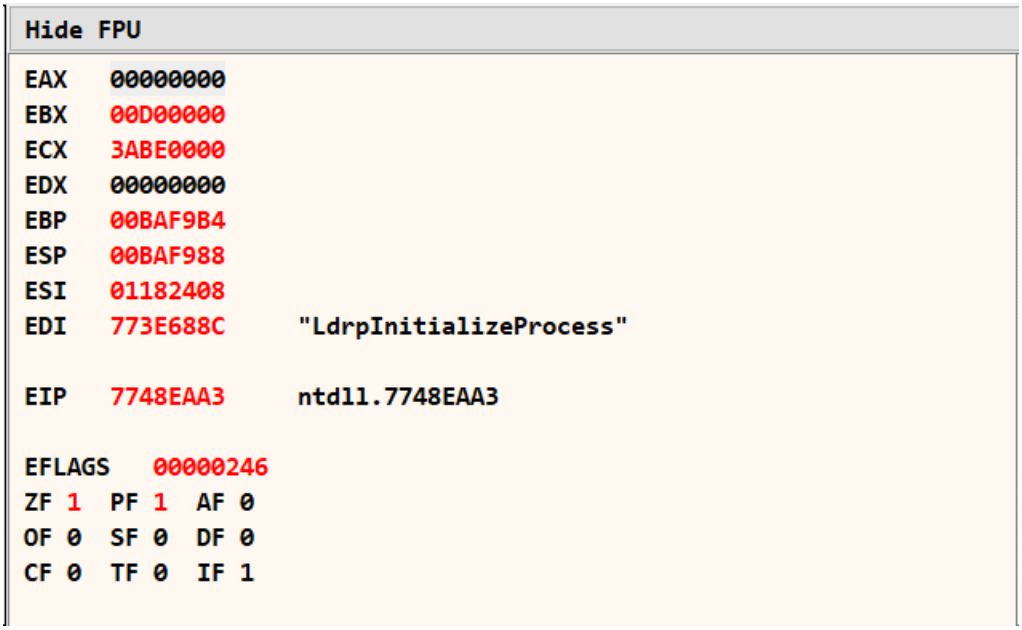
در تصویر ۸، تصویری از راهنمای توسعه‌دهندگان اینتل را مشاهده می‌کنید که ساختار این ثبات را نمایش داده است. به هر صورت، تمامی بیت‌های درون این ثبات برای پردازنده اهمیت دارند و مبتنی بر اعلام وضعیت در این ثبات، پردازنده تصمیم‌گیری می‌کند که چطور رفتار کند یا چه دستورالعملی را اجرا کند. این ثبات مخصوصا در هنگام رویارویی با دستورات پرش شرطی (Conditional Jump) اسمبلی اهمیت بسیاری زیادی دارند.



تصویر ۸: ساختار ثبات **EFLAGS**

[1] Trap Flag

در دیباگرها و حتی دیزاسمبلرهایی که دارای دیباگر هستند، همچنین یک قسمت برای نمایش وضعیت این ثبات همواره در نظر گرفته می‌شود. با مشاهده مقادیر این ثبات در هنگام تحلیل ساختارهای دیزاسمبلی، شخص تحلیلگر می‌تواند روند اجرای برنامه را در سطح کدهای اسمبلی پیش‌بینی کند. در تصویر ۹ مشاهده می‌کنید که در دیباگر x32dbg وضعیت این ثبات نمایش داده شده است.



تصویر ۹: وضعیت ثبات **EFLAGS** در معماری **x86** و برنامه **x32dbg**

نکته: همانطور که پیش از این ذکر شد، برای دریافت اطلاعات بیشتر در مورد پرچم‌های موجود در پردازنده، راهنمای Intel 64 and IA-32 Architectures Software Developer's Manuals را بخوانید.

گرچه این کتاب بسیار عظیم است (در حال حاضر دارای ۵ هزار و ۵۰ صفحه است) و تمامی مباحث برنامه‌نویسی اسمبلی برای معماری Intel را شامل می‌شود، اما همواره مطالعه آن برای افرادی که قصد دارند، مهندسی‌معکوس را به صورت حرفه‌ای فرا بگیرند، توصیه می‌شود.

EIP، ثبات اشاره‌گر دستورالعمل

در معماری x86، ثبات EIP به عنوان اشاره‌گر دستورالعمل[1] یا شمارنده برنامه[2] شناخته می‌شود. این ثبات آدرس دستورالعمل بعدی در حافظه را در خود نگه می‌دارد. تنها وظیفه ثبات EIP این است که به پردازنده بگوید در گام بعدی چه دستورالعملی را در چه آدرسی باید اجرا کند.

نکته : هنگامی‌که ثبات EIP تخریب می‌شود (به یک آدرس از حافظه اشاره می‌کند که شامل آدرس برنامه قانونی نیست) پردازنده قادر نخواهد بود کدهای برنامه اصلی که در حال اجرای آن بوده است را واکشی و اجرا کند، بنابراین برنامه‌ای که در حال اجرا است احتمالا خراب می‌شود یا در اصطلاح Crash خواهد کرد.
هنگامی‌که کنترل ثبات EIP را به‌دست می‌آورید، هر آنچه که توسط پردازنده اجرا خواهد شد را می‌توانید کنترل کنید. به همین دلیل مهاجمین همواره تلاش می‌کنند در طی فرایند Exploiting یا بهره‌برداری از آسیب‌پذیری‌ها، کنترل این ثبات را به‌دست گیرند تا بتوانند شلکدی را روی سامانه قربانی اجرا کنند.
شایان ذکر است، مهاجمین باید کد محموله حمله یا شلکد را در حافظه داشته باشند و سپس ثبات EIP را با آدرس آن تعویض کنند تا بتوانند سامانه قربانی را مورد بهره‌برداری قرار دهند.
شلکد، تکه کدی است که به آن محموله مخرب[3] یا محموله بهره‌برداری اکسپلویت گویند، این کد پس از اجرای خود، کنترل پوسته سامانه عامل قربانی را به مهاجم ارائه می‌دهد. در فصل بیستم شلکد را کامل مورد بررسی قرار خواهیم داد.

---

[1] Instruction pointer
[2] Program counter
[3] Malicious Payload

دستورالعمل‌های ساده[1]

ساده‌ترین و رایج‌ترین دستورالعمل تشریح شده در این کتاب و کلاً زبان اسمبلی، فرمان MOV است که برای انتقال داده از یک محل به محل دیگر مورد استفاده قرار می‌گیرد. به عبارت دیگر، دستورالعمل MOV یک فرمان برای خواندن و نوشتن در حافظه می‌باشد.

دستورالعمل MOV می‌تواند داده را به ثبات‌ها یا حافظه RAM انتقال دهد. قالب استفاده از آن به شکل mov destination, source است که عبارت destination به معنی اوپرند مقصد و عبارت source به معنی اوپرند منبع است.

قابل ذکر است، ما در سراسر این سلسله مقالات تحلیل بدافزار از قواعد دستوری شرکت Intel استفاده خواهیم کرد که در آن ابتدا اوپرند مقصد و سپس اوپرند منبع قرار می‌گیرد. قواعد دستوری شرکت اینتل درست برعکس قواعد دستوری شرکت AT&T می‌باشد، زیرا در قواعد دستوری اسمبلی AT&T ابتدا اوپرند منبع و سپس اوپرند مقصد در دستورالعمل قرار می‌گیرد.

جدول ۵ شامل مثال‌هایی از دستورالعمل MOV می‌باشد. اوپرندهایی که دور آن‌ها براکت قرار داده شده است به عنوان اشاره‌گر به حافظه مورد استفاده قرار می‌گیرند. به عنوان مثال، [ebx] به آدرس ذخیره شده در ثبات EBX اشاره دارد.

آخرین مثال، در جدول ۵ از یک معادله برای محاسبه آدرس حافظه استفاده می‌کند. این موضوع باعث صرف‌جویی در حافظه می‌شود زیرا نیاز به دستورالعمل‌های جدا برای انجام محاسبه مقدار میان براکت‌ها ندارد. انجام محاسبه از قبیل این نوع موارد غیر ممکن است مگر این‌که آدرس حافظه را محاسبه کنید. به عنوان مثال، دستور mov eax, ebx+esi*4 (بدون براکت) یک دستورالعمل غیر متعبر است.

جدول ۵: مثال‌هایی از دستورالعمل MOV

دستورالعمل	توضیحات
mov eax, ebx	این دستور محتوای EBX را درون EAX منتقل می‌کند.
mov eax, 0x42	این دستور مقدار 0x42 را درون EAX منتقل می‌کند.

[1] Simple Instructions

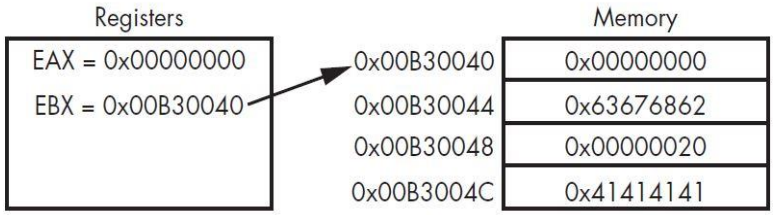
mov eax, [0x4037C4]	این دستور ۴ بایت مشخص شده در محل حافظه 0x4037C4 را به ثبات EAX منتقل می‌کند.
mov eax, [ebx]	این دستور ۴ بایت مشخص شده حافظه در EBX را به درون ثبات EAX منتقل می‌کند.
mov eax, [ebx+esi*4]	این دستور ۴ بایت در آدرس حافظه که با نتیجه محاسبه ebx+esi*4 مشخص شده را به درون ثبات EAX منتقل می‌کند.

یک دستورالعمل دیگر شبیه به فرمان MOV دستورالعمل LEA می‌باشد؛ این دستورالعمل به معنی بارگذاری آدرس موثر[1] است. قالب این دستور همانند قالب دستورالعمل MOV به شکل lea destination, source است.

دستورالعمل lea برای بارگذاری آدرس حافظه در اوپرند مقصد مورد استفاده قرار می‌گیرد. به عنوان مثال، دستورالعمل lea eax, [ebx+8] آدرس حافظه [ebx+8] را در ثبات EAX قرار می‌دهد. برخلاف دستور mov eax, [ebx+8] که داده موجود در آدرس [ebx+8] را به ثبات EAX انتقال می‌داد.

تصویر ۱۰ مقادیر ثبات‌های EAX و EBX را در سمت چپ و اطلاعات آن‌ها در حافظه را در سمت راست نشان می‌دهد. همان‌طور که مشاهده می‌کنید، ثبات EBX با آدرس 0x00B30040 تنظیم شده است و همین‌طور که در شکل نمایش داده شده است در آدرس 0x00B30048 مقدار 0x20 قرار دارد.

با استناد به این موضوع و مباحثی که در مورد دستورالعمل‌های mov و lea ارائه شد، شما می‌توانید با استفاده از دستورالعمل mov eax, [ebx+8] مقدار 0x20 را در ثبات EAX و با استفاده از دستورالعمل lea eax, [ebx+8] آدرس حافظه 0x00B30048 را در ثبات EAX قرار بدهید.



تصویر ۱۰: ثبات EBX برای دسترسی به حافظه استفاده شده است.

[1] Load Effective Address

دستورالعمل LEA منحصرا برای اشاره به آدرس حافظه مورد استفاده قرار نمی‌گیرد. این دستور در هنگام محاسبه مقادیر عددی هم مفید است، زیرا به دستورالعمل‌های کمتری برای انجام محاسبه نیاز دارد. به عنوان مثال، دیدن دستورالعمل‌هایی از قبیل lea ebx, [eax*5+5] بسیار رایج است که در آن eax بجای آدرس حافظه یک عدد است.

```
inc eax
mov ecx, 5
mul ecx
mov ebx, eax
```

این دستورالعمل معادل دستور ebx = (eax+1)*5 کارکرد دارد. اما دستورالعمل اولی برای کامپایلر دارای یک قالب کوتاه و موثرتری است. زیرا پردازنده به منظور انجام همان محاسبه دیگر نیاز نیست چهار دستورالعمل بالا را اجرا کند.

دستورالعمل‌های محاسباتی

زبان اسمبلی معماری x86 دستورالعمل‌های بسیاری را برای عملیات‌های محاسباتی دارد که محدوده آن‌ها شامل دستورالعمل‌های جمع، تفریق و عملیات‌های منطقی می‌شود. در این فصل بیشتر دستورالعمل‌هایی را که نسبت به بقیه رایج هستند، مورد بررسی قرار خواهیم داد.

دستورالعمل‌های جمع (Add) و تفریق (Sub) یک مقدار را از اوپرند مقصد کم یا جمع می‌کنند. قالب دستورالعمل جمع به شکل add destination, value است و قالب دستورالعمل تفریق به شکل sub destination, value می‌باشد.

همچنین شایان ذکر است؛ دستورالعمل تفریق دو پرچم وضعیت "***پرچم صفر*** و ***پرچم نقلی***" را تغییر می‌دهد. با این حال، پرچم صفر هنگامی تنظیم می‌شود که نتیجه محاسبه دو اوپرند برابر با صفر شود و پرچم نقلی زمانی تنظیم می‌شود (با ۱ مقداردهی می‌شود) که اوپرند مقصد کمتر از مقدار تفریق شده باشد.

علاوه بر این، دو دستورالعمل inc و dec مقدار ثبات را یک واحد زیاد و یک واحد کم می‌کنند. جدول ۶ مثال‌هایی از دستورالعمل‌های جمع و تفریق را نشان می‌دهد.

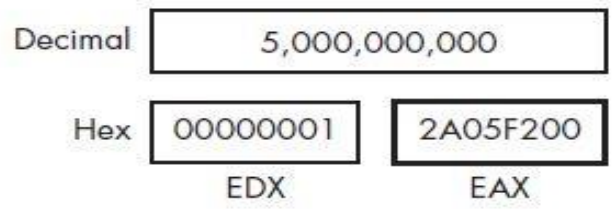
جدول ٦: مثال هایی از دستورالعمل جمع و تفریق

دستورالعمل	توضیحات
sub eax, 0x10	این دستورالعمل مقدار 0x10 را از ثبات EAX کم می‌کند.
add eax, ebx	این دستورالعمل مقدار درون ثبات ebx را با مقدار درون ثبات eax جمع می‌کند و نتیجه را در ثبات مقصد یعنی EAX ذخیره می‌کند.
inc edx	این دستورالعمل مقدار ثبات EDX را یک واحد افزایش می‌دهد.
dec ecx	این دستورالعمل مقدار ثبات ECX را یک واحد کاهش می‌دهد.

علاوه بر دو دستورالعمل جمع و تفریق، دو دستورالعمل برای ضرب و تقسیم هم وجود دارد. اما دو دستورالعمل ضرب و تقسیم یک سری تفاوت‌ها با دستورالعمل‌های جمع و تفریق دارند که آن‌ها را در این قسمت بررسی خواهیم کرد. دستورالعمل ضرب[1] و دستورالعمل تقسیم[2] هر دو با استناد به مقدار درون ثبات EAX مقدار اوپرند خود را ضرب یا تقسیم می‌کنند.

در هر حال، دستورالعمل mul مقدار اوپرند خود را همیشه با مقدار موجود در EAX ضرب می‌کند، بنابراین، ثبات EAX باید قبل از انجام عملیات ضرب مقداردهی شود. سپس نتیجه در قالب یک مقدار ۶۴ بیتی در طول دو ثبات EAX و EDX ذخیره می‌شود.

ثبات EDX با ارزشترین مقادیر ۳۲ بیتی و ثبات EAX کم ارزشترین مقدار ۳۲ بیتی را ذخیره می‌کنند. تصویر ۱۱ مقادیر در ثبات‌های EDX و EAX را نشان می‌دهد که نتیجه اعشاری عملیات ضرب برابر ۵,۰۰۰,۰۰۰,۰۰۰ شده است و چون این مقدار برای قرارگیری در یک ثبات بیش از حد بزرگ می‌باشد، لذا در دو ثبات قرار گرفته است.



تصویر ۱۱: ذخیره‌سازی نتیجه ضرب در دو ثبات **EDX** و **EAX**

[1] Multiplication
[2] Division

شایان ذکر است، دستورالعمل div value عملیات مشابهی مانند دستور mul انجام می‌دهد، با این تفاوت‌که دستورالعمل Div عملیاتی مخالف دستورالعمل mul انجام می‌دهد و مقدار اوپرند خود را با مقدار ثبات EAX تقسیم می‌کند و مقادیر را در EDX و EAX ذخیره می‌کند. در حالی‌که، مقادیر ثبات‌های EDX و EAX پیش از آن باید با مقادیر مناسب مقداردهی شده باشند.

نتیجه محاسبه دستورالعمل div در ثبات EAX و باقی‌مانده نتیجه محاسبه در EDX ذخیره‌سازی می‌شود. یک برنامه نویس باقیمانده عملیات تقسیم را با استفاده از عملیاتی به نام modulo یا "به پیمانه (N)" به دست می‌آورد که با استفاده از ثبات EDX پس از دستورالعمل div به جمع تبدیل می‌شود (از آنجایی‌که باقیمانده را در بر می‌گیرد).

جدول ۷: مثال‌هایی از دستورالعمل‌های ضرب و تقسیم

دستورالعمل	توضیحات
mul 0x50	این دستورالعمل مقدار 0x50 را با مقدار موجود در ثبات EAX ضرب می‌کند و نتیجه را در ثبات‌های EDX:EAX ذخیره می‌کند.
div 0x75	این دستورالعمل مقدار 0x75 را با مقدار درون ثبات‌های EDX:EAX تقسیم می‌کند و نتیجه را در EAX و باقی مانده را در EDX ذخیره می‌کند.

همچنین عملیات‌های منطقی از قبیل or، xor و and هم در معماری x86 استفاده می‌شوند. از این دستورالعمل‌ها می‌توانید برای عملیات‌های جمع و تفریق هم استفاده کنید. این دستورالعمل‌ها، عملیات مد نظر خود را بین اوپرند مقصد و منبع انجام داده و نتیجه را در اوپرند مقصد ذخیره می‌کنند.

دستورالعمل xor فرمانی است که در حین دیزاسمبلی برنامه‌ها بسیار با آن مواجه خواهید شد. به عنوان مثال، دستورالعمل xor eax, eax را در نظر بگیرید، این دستورالعمل سریع‌ترین راه ممکن برای تنظیم ثبات eax با صفر است و از آن برای بهینه‌سازی اجرای برنامه‌ها استفاده می‌شود، زیرا که تنها برای انجام عملیات خود به دو بایت نیاز دارد، در حالی‌که دستورالعمل mov eax, 0 به پنج بایت نیاز دارد.

دستورالعمل‌های shr و shl فرامینی هستند که برای حرکت[1] بیت‌های یک ثبات مورد استفاده قرار می‌گیرند. قالب دستورالعمل shr به شکل shr destination, count است. همچنین دستورالعمل shl دارای قالب مشابه با دستورالعمل shr است. دستورالعمل‌های shr و shl بیت‌های اوپرند مقصد را با عددی که در قسمت اوپرند count مشخص می‌کنید به ترتیب به راست و چپ حرکت می‌دهند.

قابل ذکر است، بیت‌هایی که فراتر از محدوده اوپرند مقصد حرکت داده می‌شوند، درون پرچم نقلی قرار داده می‌شوند. به عنوان مثال، اگر مقدار باینری ۱۰۰۰ را داشته باشید و آن را به راست ۱ بار حرکت دهید نتیجه آن برابر ۰۱۰۰ می‌شود. در پایان دستورالعمل انتقال، پرچم نقلی شامل آخرین بیت خارج شده از اوپرند مقصد می‌شود.

دستورات چرخشی (rol و ror) رشته‌های بیتی را به صورت دایره‌ای حرکت می‌دهند، این دستورات مشابه انتقال بیت‌ها عمل می‌کنند، با این تفاوت، بیتی که از یک طرف از داده خارج می‌شود به طور دوار از جهت دیگر وارد آن می‌شود. پردازنده x86 چهار دستورالعمل چرخشی (rol، ror، rcl و rcr) دارد که ما دو تا از آن‌ها را مورد بررسی قرار خواهیم داد. جدول ۸-۴ مثال‌هایی از این دستورالعمل‌ها را نمایش می‌دهد.

جدول ۸: مثال‌هایی از رایج‌ترین دستورالعمل‌های منطقی و محاسباتی انتقال بیت‌ها

دستورالعمل	توضیحات
xor eax, eax	این دستورالعمل ثبات EAX را پاک می‌کند.
or eax, 0x7575	این دستور با مقدار 0x7575 روی EAX عملیات منطقی OR را انجام می‌دهد.
mov eax, 0xA **shl eax, 2**	این دستور مقدار 0xA را به ثبات EAX انتقال داده و سپس مقدار بیت‌های درون ثبات EAX را دو بار به سمت چپ انتقال می‌دهد و مقدار 0xA در باینری ۱۰۱۰ به مقدار 0x28 در باینری ۱۰۱۰۰۰ تبدیل می‌شود.
mov bl, 0Xa **ror bl, 2**	این دستور مقدار 0xA را به ثبات BL انتقال داده و سپس مقدار بیت‌های درون ثبات EAX را دو بار به سمت راست چرخش می‌دهد و مقدار 0xA در باینری ۱۰۱۰ به مقدار ۱۰۰۰۰۰۱۰ در باینری تبدیل می‌شود.

[1] Shift

اغلب اوقات از دستورالعمل‌های حرکت‌دهنده بیت‌ها برای انجام بهینه عملیات‌های ضرب و تقسیم استفاده می‌شود. با این حال می‌توان گفت، انتقال بیت‌ها یک راه سریع و ساده برای انجام عملیات‌های ضرب و تقسیم هستند، زیرا شخص برنامه‌نویس نیاز ندارد مقدار درون ثبات‌ها را تنظیم کند و به منظور تقسیم یا ضرب محتوای درون اوپرند مقصد انتقال داده انجام بدهد. به عنوان مثال، نتیجه دستورالعمل shl eax, 2 با ضرب مقدار ثبات eax با ۲ برابر است و همچنین مقدار shr eax , 2 با تقسیم مقدار اوپرند eax با ۲ نتیجه برابر تولید می‌کند.

شایان ذکر است، هر بار که یک بیت اطلاعات را به سمت چپ انتقال می‌دهیم، اوپرند مقصد ضرب در ۲ می‌شود و اگر ۲ بار به سمت چپ انتقال داده شود در ۴ ضرب می‌شود و اگر سه بار به سمت چپ انتقال داده شود در ۸ ضرب می‌شود و اگر N بار به سمت چپ انتقال داده شود محتوای اوپرند مقصد به ۲^n می‌رسد.

در حین تجزیه و تحلیل بدافزار، اگر با تابعی مواجه شدید که بطور مکرر شامل دستورالعمل‌های xor, or, and, shl, ror, shr یا rol شده است، می‌توانید احتمال دهید که با یک تابع رمزنگاری یا فشرده شده مواجه شده‌اید.

با این حال خود را درگیر تحلیل تک تک دستورالعمل‌های آن تابع نکنید، مگر این‌که واقعا به تحلیل آن نیاز داشته باشید. همچنین می‌توانید، آن تابع را به عنوان یک تابع رمزنگاری شده مشخص سازید و از آن بگذرید.

دستورالعمل NOP

آخرین فرمان، دستورالعمل ساده NOP است که هیچ عملی انجام نمی‌دهد. هنگامی‌که این دستورالعمل به خدمت گرفته می‌شود بدون این‌که عملیاتی رخ دهد، پردازنده به منظور اجرای فرامین به دستورالعمل بعدی منتقل می‌شود.

دستورالعمل NOP نام مستعاری برای فرمان xchg eax, eax است، بدین دلیل که تعویض ثبات EAX با خودش هیچ عمل خاصی انجام نمی‌دهد، مشهور به دستورلعمل No Operation به معنای هیچ عملی است. با این حال استفاده از دستورالعمل NOP برای انجام هیچ عملیاتی، ساده‌تر و بهینه‌تر است.

کد ماشین برای این دستورالعمل برابر 0x90 می‌باشد. این دستورالعمل به طور رایج در حملات سرریز بافر، زمانی‌که مهاجمان کنترل کامل از بهره‌برداری خود ندارند، به عنوان NOPهای حامل[1] استفاده می‌شوند. NOPهای حامل یک لایه اجرایی ارائه می‌دهند که خطر شروع اجرای شلکد در وسط و خرابی اجرای آن را کاهش می‌دهند. در فصل نوزدهم در مورد NOPهای حامل و شلکدها بحث کاملتری خواهیم کرد.

پشته

حافظه توابع، متغیرهای محلی و کنترل جریان برنامه در پشته ذخیره می‌شود که یک ساختمان داده با خصوصیت PUSH و POP است. شما می‌توانید با استفاده از دستور PUSH در این حافظه داده قرار دهید و با استفاده از دستور POP آن داده‌ها را از روی پشته بردارید.

شایان ذکر است، پشته ناحیه‌ای از حافظه اصلی است که دارای ساختار LIFO (last-in-first-out) است. یعنی آخرین داده‌ای که به آن وارد می‌شود، اولین داده‌ای است که از آن خارج می‌شود. به عنوان مثال، اگر در پشته اعداد ۱، ۲ و سپس ۳ را قرار بدهید، می‌توانید در ابتدا ۳ را از حافظه پشته بخوانید و سپس ۲ و ۱ را، زیرا آخرین ورودی به حافظه ۳ بوده است.

معماری x86 به منظور پشتیبانی از مکانیزم پشته ساخته شده است و از ثبات‌های ESP و EBP پشتیبانی می‌کند. ثبات ESP اشاره‌گر پشته است و معمولا شامل آدرس حافظه‌ای می‌شود که به بالای پشته اشاره می‌کند. مقدار این ثبات هنگامی‌که داده‌ای درون پشته قرار می‌گیرد یا از آن برداشته می‌شود، تغییر می‌کند.

یک برنامه خارجی که یک پارامتر را از طریق پشته ارسال می‌کند را در نظربگیرید. وقتی ساب‌روتینی درخواست می‌شود، پارامتر می‌تواند با آدرس‌دهی غیرمستقیم [ESP+4] در دسترس قرار گیرد. اگر از حافظه پشته در ساب‌روتین برای ذخیره داده استفاده شود، عدد بیشتری باید به ESP اضافه شود.

ثبات EBP را برای ارجاع به داده‌های درون پشته می‌توان به کار برد. ثبات ESP با هر push و pop تغییر می‌کند اما EBP ابتدا برابر با ESP می‌شود و سپس ثابت می‌ماند و در انتهای اجرای ساب‌روتین، مقدار اولیه EBP باید برگردانده شود. بعد از این‌که ساب‌روتینی تمام شد پارامترهایی که در پشته اضافه شده‌اند باید حذف گردند.

---

[1] NOP sled

دستورالعمل‌های کار روی پشته شامل push, pop, call, leave, enter و ret می‌شوند. پشته در یک قالب از بالا به پایین به حافظه اختصاص داده شده است و بالاترین آدرس اختصاص داده شده برای اولین بار استفاده می‌شود. مقادیری که در پشته قرار می‌گیرند، آدرس‌های کوتاه‌تری استفاده می‌کنند (این موضوع در تصویر ۱۲ نمایش داده شده است).

پشته تنها برای ذخیره‌سازی کوتاه مدت داده‌ها مورد استفاده قرار می‌گیرد و بطور پیوسته متغیرهای محلی، پارامترها و آدرس‌های بازگشتی را شامل می‌شود. از این حافظه اصالتاً برای مدیریت داده‌های تبادل شده میان توابع استفاده می‌شود. پیاده‌سازی پشته در کامپایلرهای مختلف متفاوت است، اما قرار داد استفاده از پشته برای متغیرهای محلی و ارجاع به پارامترها نسبت به EBP رایج‌تر است.

فراخوانی توابع

توابع بخشی از کد یک برنامه هستند که یک عملیات خاص را انجام می‌دهند و نسبتا مستقل از کد اصلی برنامه هستند. فراخوانی یک تابع موقتاً اجرای کد اصلی برنامه را به تابع فراخوانی شده منتقل می‌کند. چگونگی استفاده پشته توسط یک برنامه در سراسر یک فایل باینری ثابت است. در حال حاظر، ما روی رایج‌ترین قرارداد تمرکز خواهیم کرد که به عنوان قرارداد cdecl شناخته می‌شود. در قسمت‌های بعدی این سلسله مقالات جایگزین‌های این قرارداد از قبیل stdcall و fastcall و ... را مرور خواهیم کرد.

بیشتر توابع شامل prologue (چند خط کد در آغاز تابع) می‌شوند. prologue پشته و ثبات‌ها را برای استفاده در تابع آماده می‌کند. به همین ترتیب، یک epilogue در پایان توابع قرار دارد که پشته و ثبات‌ها را به حالت ابتدایی قبل از فراخوانی تابع بازیابی می‌کند. لیست زیر جریان رایج‌ترین پیاده‌سازی فراخوانی‌های توابع را خلاصه می‌کند. کمی بعد، تصویر ۱۳ نمودار طرح یک پشته را نشان می‌دهد و ساختار پشته را شفاف‌سازی می‌کند.

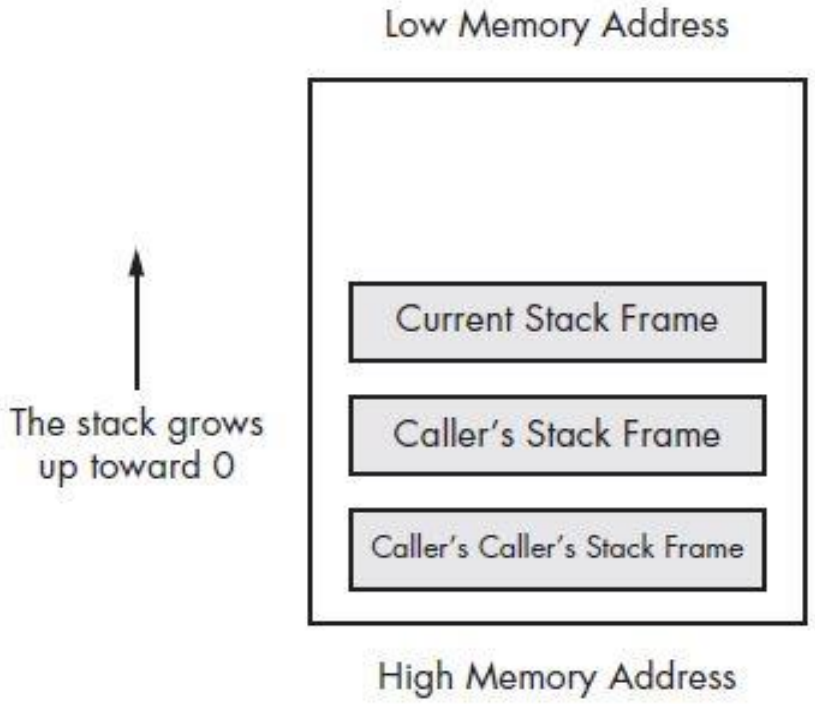
۱. پارامترها با استفاده از دستورالعمل Push در پشته قرار داده می‌شوند.
۲. یک تابع با استفاده از call memory_location فراخوانی می‌شود. این دستور باعث می‌شود، آدرس دستورالعمل جاری (محتوای جاری ثبات EIP) در پشته قرار گیرد. این آدرس هنگامی‌که کار تابع به پایان می‌رسد، سپس برای بازگشت به کد اصلی برنامه استفاده خواهد شد. هنگامی‌که تابع اجرا می‌شود، ثبات EIP با memory_location (آدرس شروع تابع) تنظیم می‌شود.

۳. با استفاده از prologue یک تابع، فضای تخصیص داده شده به پشته برای متغیرهای محلی و اشاره‌گر پایه (EBP) در پشته قرار می‌گیرد. این کار برای ذخیره EBP به منظور فراخوانی توابع است.
۴. تابع کار خودش را انجام می‌دهد.
۵. با استفاده از epilogue تابع، پشته به حالت اول خود بازیابی می‌شود. ثبات ESP برای آزادی متغیرهای داخلی تنظیم می‌شود و سپس ثبات EBP بازیابی می‌شود تا تابع فراخوانی شده بتواند متغیرهای خودش را درست آدرس‌دهی کند. دستورالعمل leave می‌تواند به عنوان یک epilogue استفاده شود، زیرا ثبات ESP را با EBP تنظیم می‌کند و آدرس ثبات EBP را از روی پشته خارج می‌کند.
۶. تابع با فراخوانی دستورالعمل ret بازگشت داده می‌شود. این دستور آدرس بازگشتی را از حافظه پشته به درون EIP قرار می‌دهد، بنابراین برنامه اجرای جریان عادی برنامه را ادامه می‌دهد.
۷. پشته برای حذف پارامترهای ارسال شده تنظیم می‌شود، مگر این‌که آن‌ها دوباره مورد استفاده قرار گیرند.

طرح پشته[1]

همان‌طور که بحث شد، پشته در یک مدل پایین به بالا اختصاص داده می‌شود. از همین روی بالاترین آدرس‌های حافظه پشته ابتدا مورد استفاده قرار می‌گیرند. تصویر ۱۲ نحوه طرح ریزی پشته در حافظه را نمایش می‌دهد. هر بار که یک فراخوانی صورت می‌گیرد، یک قاب پشته جدید تولید می‌شود و تا زمانی‌که فراخوانی قاب پشته به حالت اول بازگردد و اجرای تابع به برنامه فراخوانی کننده برگردد تابع قالب پشته خود را نگه می‌دارد.

[1] Stack Layout



تصویر ۱۲: نقشه حافظه پشته در معماری x86

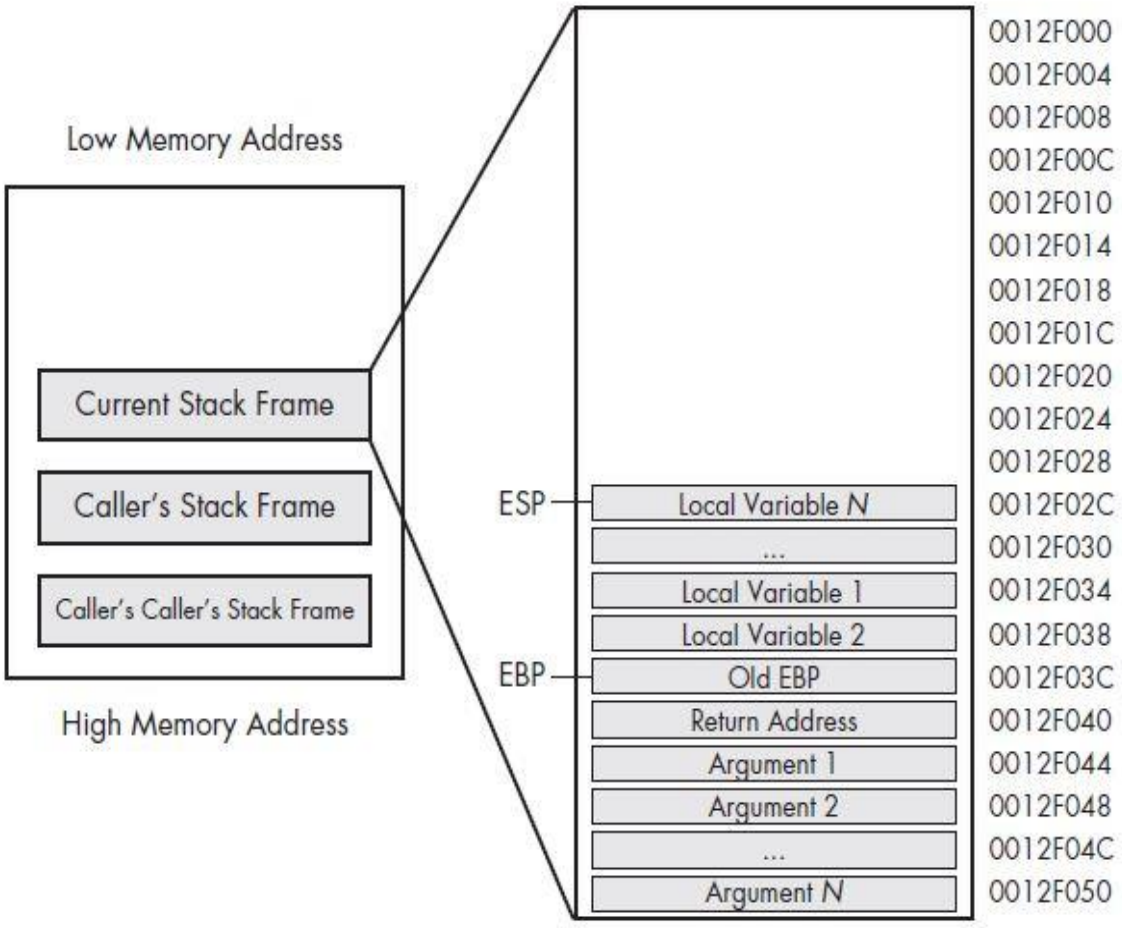
تصویر ۱۳ کالبد یک قاب پشته منحصربفرد از تصویر ۱۲ را نشان می‌دهد. همچنین مکان‌های حافظه ایتم‌های فردی هم در تصویر ۱۳ نشان داده شده است. در این دیاگرام، ثبات ESP به نقطه بالای پشته اشاره می‌کند که آدرس آن برابر با مقدار 0x12F02C است. در طول اجرای تابع ثبات EBP ممکن است با مقدار آدرس 0x12F03C تنظیم شود، زیرا متغیرهای محلی و پارامترها با استفاده از ثبات EBP ارجاع داده می‌شوند.

پارامترهایی که قبل از فراخوانی روی پشته قرار گرفته‌اند در پایین قاب پشته نمایش داده شده‌اند. بعدی، شامل آدرس بازگشتی می‌شود که بطور خودکار توسط دستورالعمل فراخوانی تابع در پشته قرار داده می‌شود. ثبات EBP قدیمی گزینه بعدی در پشته است، این ثبات EBP از قاب پشته فراخوانی‌کننده[1] است.

هنگامی‌که اطلاعات در پشته قرار می‌گیرد، ESP کاهش پیدا می‌کند. در مثال تصویر ۱۳ اگر دستورالعمل push eax اجرا شده بود، ثبات ESP به مقدار ۴ بایت کاهش پیدا می‌کرد و شامل آدرس 0x12F028 می‌شد و داده ثبات eax در آن آدرس کپی قرار می‌گرفت. همچنین اگر دستورالعمل pop ebx اجرا شده

[1] Caller's stack frame

بود، داده درون آدرس 0x12F028 به درون ثبات EBX منتقل می‌شد و سپس ثبات ESP به مقدار چهار بایت افزایش پیدا می‌کرد.



تصویر ۱۳: قالب منفرد پشته

علاوه بر همه این نکات، خواندن اطلاعات از حافظه پشته بدون دستورات push و pop ممکن است، به عنوان مثال، دستورالعمل mov eax ss:[esp] مستقیما دسترسی به بالای پشته را ارائه می‌دهد. این دستورالعمل مشابه دستورالعمل pop eax است، جزء این‌که در استفاده از آن ثبات esp درگیر نیست. قرارداد استفاده شده در این دستورالعمل بستگی به کامپایلر و چگونگی پیکربندی کامپایلر دارد (در فصل ششم با جزییات دقیق‌تر این موضوع را مورد بررسی قرار خواهیم داد.).

معماری x86 دستورالعمل‌های اضافه دیگری برای pop و push ارائه می‌دهد. معروفترین آنها، دستورالعمل‌های pusha و pushad هستند که تمامی ثبات‌ها را درون پشته قرار می‌دهند و سپس از

popa و popad برای بازیابی تمامی ثبات‌ها قرار داده شده در پشته استفاده می‌شود. توابع pushad و pusha به شرح زیر عمل می‌کنند:

- دستورالعمل pusha ثبات‌های ۱۶ بیتی CX, DX, BX, SP, BP, SI, DI,AX را به ترتیب در پشته قرار می‌دهد.
- دستورالعمل pushad ثبات‌های ۳۲ بیتی EAX, ECX, EDX, EBX, ESP, EBP, ESI, AX EDI را به ترتیب در پشته قرار می‌دهد.

این دستورالعمل‌ها هنگامی‌که یک شخص می‌خواهد وضعیت جاری ثبات‌ها را در پشته ذخیره کند، به طور معمول در شلکدها به کار گرفته می‌شوند، زیرا آن‌ها را در هر زمان می‌توانند حالت تمامی ثبات‌ها را بازیابی کنند. کامپایلرها به ندرت از این دستورالعمل‌ها استفاده می‌کنند، بنابراین دیدن این دستورالعمل‌ها اغلب نشان می‌دهد شخصی آن برنامه را مونتاژ دستی کرده یا آن یک شلکد است.

دستورالعمل‌های شرطی[1]

همه زبان‌های برنامه‌نویسی توانایی مقایسه و تصمیم‌گیری بر مبنای دستورالعمل‌های مقایسه‌ای خود را دارند. فرامین شرطی دستورالعمل‌هایی هستند که عمل مقایسه را انجام می‌دهند و مشهورترین دستورالعمل‌های شرطی test و cmp هستند. دستورالعمل test مشابه دستورالعمل and است؛ این دستورالعمل هیچ تغییری روی اوپرندهای خود ایجاد نمی‌کند. بلکه دستورالعمل test فقط پرچم‌ها را تنظیم می‌کند.

پرچم صفر (Zero Flag) معمولا پرچمی است که بعد از اجرای دستورالعمل test تحت تاثیر قرار می‌گیرد. انجام test محتوای یک مقدار با خودش اغلب برای بررسی مقادیر NULL استفاده می‌شود. مثلا، دستورالعمل test eax, eax یک مثال برای این مضمون است. همچنین می‌توانید EAX را با صفر مقایسه کنید، اما دستورالعمل test eax, eax از بایت و چرخه‌های پردازنده کمتری استفاده می‌کند و در نتیجه بهینه‌تر عمل می‌کند.

[1] Conditionals

دستورالعمل cmp مشابه دستورالعمل sub است؛ به هر حال، در این دستورالعمل اوپرندها دوباره تحت تاثیر قرار نمی‌گیرند. دستورالعمل cmp همانند test برای تنظیم پرچم‌ها استفاده می‌شود. پرچم صفر (Zero Flag) و پرچم نقلی (Carry Flag) ممکن است بر مبنای نتیجه دستورالعمل cmp تغییر یابند. جدول شماره ۹-۴ چگونگی تاثیر دستورالعمل cmp روی این پرچم‌ها را نمایش می‌دهد.

جدول ۹: پرچم ها و دستورالعمل **CMP**

cmp dst, src	ZF	CF
اوپرند منبع اگر با مقصد برابر باشد	۱	۰
اوپرند منبع اگر بزرگتر از مقصد باشد	۰	۱
اوپرند مقصد اگر از منبع بزرگتر باشد	۰	۰

دستورات انشعابی[1]

ساختارهای کنترلی نظیر عبارات شرطی و حلقه‌های تکرار توسط دستورات پرش ساخته می‌شود. پردازنده ۸۰۸۶ چند نوع دستورالعمل پرش را در اختیار می‌گذارد.

۱. دستورات پرشی بدون شرط
۲. دستورات پرشی با شرط
۳. ساختار شرطی

دستورات برنامه پشت سر هم اجرا می‌شوند یعنی پردازنده دستورات را به ترتیبی که در برنامه ظاهر شده‌اند اجرا می‌کند. ساختارهای کنترلی نظیر عبارات شرطی، حلقه‌ها و فراخوانی زیربرنامه، روتین اجرای برنامه را تغییر می‌دهد. زبان‌های سطح بالا ساختارهای کنترلی مانند دستورات if و while را در اختیار می‌گذارند که اجرای برنامه را کنترل می‌کنند.

زبان اسمبلی چنین ساختارهای پیچیده‌ای را ندارد در عوض از دستورات پرش برای پیاده‌سازی این ساختارهای کنترلی استفاده می‌کند (که البته استفاده نامناسب آن باعث کد اسپاگتی می‌شود). دستورات پرش اجرای برنامه را به نقطه دلخواهی منتقل می‌کنند. دو نوع دستورالعمل پرش وجود دارد:

[1] Branching

۱. دستورات پرشی بدون شرط
۲. دستورات پرشی با شرط

گونه های مختلفی از دستورات پرشی وجود دارند:

۱. **کوتاه (short) :** این نوع پرش بسیار محدود است و تنها می‌تواند ۱۲۸ بایت بالا یا پایین بپرد. مزیت آن در مصرف کمتر حافظه است. میزان جابجائی تنها توسط یک بایت مشخص می‌شود که تعیین می‌کند چند بایت جلوتر یا عقب تر برود. این فاصله به ثبات IP اضافه می‌شود.
۲. **نزدیک (near) :** این نوع پرش می‌تواند به هر موقعیت درون یک سگمنت پرش کند.
۳. **دور (far) :** این نوع پرش اجازه حرکت به سگمنت های دیگر را می‌دهد.

دستورات پرش شرطی

دستورالعمل jmp بدون هیچ شرطی کنترل را به نقطه دیگری در برنامه منتقل می‌کند و مشابه دستور goto در زبان‌های سطح بالا عمل می‌کند. فرم کلی آن به صورت jmp target است. target می‌تواند آدرس درون همین سگمنت یا سگمنت کد دیگری باشد. معمولا آدرس مقصد توسط یک برچسب معین می‌شود.

برچسب شناسه‌ای است که به دنبال آن علامت کلون (:) می‌آید. اسمبلر با توجه به آفست دستور بعد از برچسب، فاصله پرش را به طور اتوماتیک محاسبه می‌کند. دستورالعمل بعد از jmp هیچوقت اجرا نمی‌شود مگر این که از دستور دیگری به آن پرش شده باشد. دستور jmp به تنهایی در برنامه موثر نیست و برای ساختن ساختارهای کنترلی همراه با دستورات پرش شرطی استفاده می‌شود.

دستورات پرش شرطی برای ساختن حلقه‌ها و عبارات شرطی مانند if نیز بکار می‌روند. پرش‌های شرطی یک یا چند پرچم را بررسی می‌کنند و با توجه به وضعیت آن‌ها کنترل را به آدرس معینی منتقل می‌کنند. اگر پرش انجام نشود اجرا از دستورالعمل بعد از ادامه پیدا می‌کند. با توجه به اینکه دستورات پرش شرطی پرچم‌ها را بررسی می‌کنند قبل از دستور پرش باید دستوری وجود داشته باشد که روی پرچم ها تاثیر بگذارد.

برای مثال بعد از اجرای دستور shl می‌توانید پرچم نقلی را بررسی کنید تا ببینید بیت ۱ از سمت چپ عدد خارج شده است یا خیر. یا بعد از دستورالعمل test می‌توانید پرچم صفر را بررسی کنید تا ببینید بیت‌های

مشخصی در یک عدد ۱ بوده‌اند یا خیر. البته در اکثر موارد ساختارهای کنترلی بر اساس مقایسه مقادیر و توسط cmp پیاده‌سازی می‌شوند.

دستورالعمل cmp با توجه به حاصل تفریق دو اوپرند خود پرچم‌های وضعیت را تنظیم می‌کند بنابراین می‌تواند برای بررسی بزرگتر، کوچکتر یا مساوی بودن مقادیر استفاده شود. برای مقدارهای بدون علامت دو پرچم نقلی و پرچم صفر از ثبات پرچم وضعیت بسیار مهم هستند و برای اعداد علامتدار پرچم های علامت و پرچم صفر اهمیت دارند.

پرچم صفر مساوی بودن اوپرند را نشان می‌دهد. دستورات پرش تنها پرچم‌ها را بررسی می‌کنند و روی آن‌ها تاثیری ندارند. دستورالعمل‌های پرش شرطی انواع مختلفی دارند که همگی با حرف J شروع می‌شوند و بعد از آن حروف دیگر قرار می گیرند. بدنبال دستور یک آدرس یا برچسب ذکر می‌شود. این دستورات را به سه دسته کلی می‌توان تقسیم کرد:

۱. دستورات پرش بر اساس پرچم‌ها
۲. دستورات پرش بعد از مقایسه اوپرندهای بدون علامت
۳. دستورات پرش بعد از مقایسه اوپرندهای علامتدار

به هر حال، بیش از ۳۰ نوع مختلف از پرش‌های شرطی می‌تواند مورد استفاده قرار گیرد که فقط ما با یک مجوعه کوچک از آن‌ها بسیار مواجه خواهیم شد. جدول شماره ۱۰-۴ رایج‌ترین دستورالعمل‌های پرش شرطی و جزییات چگونگی عملکرد آن‌ها را نمایش می‌دهد.

جدول ۱۰: پرش های شرط

دستورالعمل	توضیحات
jz loc	پرش کن به محل مشخص شده در صورتی که پرچم صفر با ۱ تنظیم شده باشد.
jnz loc	پرش کن به محل مشخص شده در صورتی که پرچم صفر با ۰ تنظیم شده باشد.
je loc	مشابه JZ است، با این تفاوت‌که به صورت رایج‌تری بعد از دستورالعمل cmp مورد استفاده قرار می‌گیرد. هنگام استفاده از این دستورالعمل؛ پرش زمانی رخ خواهد داد که اوپرند مقصد با اوپرند منبع برابر باشد.

jne loc	مشابه JNZ است، با این تفاوت‌که به صورت رایج‌تری بعد از دستورالعمل cmp مورد استفاده قرار می‌گیرد. هنگام استفاده از این دستورالعمل؛ پرش زمانی رخ خواهد داد که اوپرند مقصد با اوپرند منبع برابر نباشد.
jg loc	این دستورالعمل بعد از دستورالعمل cmp یک مقایسه علامت‌دار انجام می‌دهد. هنگام استفاده از این دستورالعمل؛ پرش زمانی رخ خواهد داد که اوپرند مقصد بزرگتر از اوپرند منبع باشد.
jge loc	این دستورالعمل بعد از دستورالعمل cmp یک مقایسه علامت‌دار انجام می‌دهد. هنگام استفاده از این دستورالعمل؛ پرش زمانی رخ خواهد داد که اوپرند مقصد بزرگتر از یا مساوی اوپرند منبع باشد.
ja loc	این دستورالعمل مشابه دستورالعمل jg است با این تفاوت‌که یک مقایسه بدون علامت انجام می‌دهد.
jae loc	این دستورالعمل مشابه دستورالعمل jae است با این تفاوت‌که یک مقایسه بدون علامت انجام می‌دهد.
jl loc	این دستورالعمل بعد از دستورالعمل cmp یک مقایسه علامت‌دار انجام می‌دهد. هنگام استفاده از این دستورالعمل؛ پرش زمانی رخ خواهد داد که اوپرند مقصد کوچکتر از اوپرند منبع باشد.
jle loc	این دستورالعمل بعد از دستورالعمل cmp یک مقایسه علامت‌دار انجام می‌دهد. هنگام استفاده از این دستورالعمل؛ پرش زمانی رخ خواهد داد که اوپرند مقصد کوچکتر از یا برابر اوپرند منبع باشد.
jb loc	این دستورالعمل مشابه دستورالعمل jl است با این تفاوت‌که یک مقایسه بدون علامت انجام می‌دهد.
jbe loc	این دستورالعمل مشابه دستورالعمل jle است با این تفاوت‌که یک مقایسه بدون علامت انجام می‌دهد.
jo loc	پس از استفاده از دستورالعمل jo پرش زمانی رخ خواهد داد که دستورالعمل قبلی پرچم سرریز بافر OF را با ۱ تنظیم کند.
js loc	پس از استفاده از دستورالعمل js پرش زمانی رخ خواهد داد که دستورالعمل قبلی پرچم علامت را با ۱ تنظیم کند.
jecxz loc	پس از استفاده از این دستورالعمل پرش زمانی رخ خواهد داد که ثبات ECX برابر با صفر باشد.

دستورالعمل‌های تکرار (REP)

دستورالعمل‌های تکرار[1] مجموعه‌ای از فرامین برای دستکاری داده‌های بافر هستند. آن‌ها معمولا آرایه‌ای از بایت‌ها هستند، همچنین می‌توانند یک کلمه[2] یا دو کلمه باشند. در این قسمت ما روی آرایه‌ای از بایت‌ها تمرکز خواهیم کرد. (اینتل به این دستورالعمل‌ها با واژه دستورالعمل‌های رشته‌ای[3] اشاره می‌کند، اما ما به منظور جلوگیری از سردرگمی با رشته‌هایی که در فصل ۱ مورد بحث قرار گرفت از این واژه استفاده نخواهیم کرد.).

رایج‌ترین فرمان‌ها برای تغییر داده‌های درون بافر، دستورالعمل‌های movsx، cmpsx، stosx و scasx هستند که بجای X می‌توان از حروف w یا b برای مشخص‌سازی اندازه بایت‌ها به ترتیب مورد استفاده قرار گیرد. این دستورالعمل‌ها با هر نوع داده‌ای کار می‌کنند، اما در این قسمت ما روی بایت‌ها تمرکز خواهیم کرد. بنابراین از دستورالعمل‌های movsb، cmpsb و غیره استفاده خواهیم کرد. ثبات‌های ESI و EDI در این عملیات‌ها استفاده می‌شوند. ESI ثبات ایندکس منبع و EDI ثبات ایندکس مقصد است و همچنین از ثبات ECX به عنوان یک ثبات شمارشی استفاده می‌شود.

این دستورالعمل‌ها برای عملیات روی داده‌ها نیاز به یک پیشوند دارند که اندازه آن بزرگتر از یک باشد. دستورالعمل movsb تنها یک بایت انتقال می‌دهد و از ثبات ECX استفاده نمی‌کند. در معماری x86، پیشوندهای تکرار برای عملیات‌های چند بایتی مورد استفاده قرار می‌گیرند.

دستورالعمل rep آفست ثبات‌های ESI و EDI را افزایش و مقدار ثبات ECX را کاهش می‌دهد. دستورالعمل REP تا زمانی‌که ECX برابر صفر نباشد ادامه پیدا خواهد کرد. در جدول ۱۱-۴ مثال‌هایی از این دستورالعمل‌ها نمایش داده شده است. بنابراین در بیشتر دستورالعمل‌های دستکاری کننده داده‌های بافر، ثبات‌های ESI، EDI و ECX باید مقداردهی اولیه برای دستورالعمل REP شده باشند.

جدول ۱۱: دستورالعمل‌های REP

دستورالعمل	توضیحات
rep	عمل تکرار تا زمانی‌که ثبات ECX برابر صفر شود ادامه پیدا خواهد کرد.

[1] Rep instructions
[2] Word
[3] String Instructions

repe, repz	عمل تکرار تا زمانی‌که ثبات ECX برابر صفر شود یا پرچم صفر برابر ۰ شود، ادامه پیدا خواهد کرد.
repne, repnz	عمل تکرار تا زمانی‌که ثبات ECX برابر صفر شود یا پرچم صفر برابر ۱ شود، ادامه پیدا خواهد کرد.

دستور movsb یک رشته یک بایتی را از یک محل به محل دیگر انتقال می‌دهد. پیشوند rep معمولا با movsbبرای کپی دنباله‌ای از بایت‌ها با اندازه تعریف شده در ECX مورد استفاده قرار می‌گیرد. دستورالعمل rep movsb معادل منطقی تابع memcpy زبان برنامه‌نویسی C است.

دستورالعمل movsb بایت‌ها را از آدرس ESI برداشته و آن‌ها را در آدرس EDI ذخیره می‌کند و سپس ثبات‌های ESI و EDI را با توجه به تنظیم جهت پرچم DF یکی کاهش یا افزایش می‌دهد. اگر پرچم جهت یا DF برابر با ۰ باشد، آن‌ها افزایش پیدا می‌کنند و اگر اینطور نباشد آن‌ها کاهش پیدا می‌کنند.

شما به ندرت این را در کدهای C کامپایل شده خواهید دید، اما در شلکد، افراد گاهی اوقات پرچم جهت را وارونه می‌کنند تا این‌که بتوانند داده‌ها را در جهت معکوس ذخیره سازند. اگر پشوند REP وجود داشته باشد، ثبات ECX بررسی می‌شود تا شامل صفر نباشد. اگر نبود، دستورالعمل movsb بایت‌ها را از ESI به EDI منتقل می‌کند و مقدار ثبات ECX را کاهش می‌دهد. این فرایند تا زمان برابری ECX با صفر ادامه پیدا خواهد کرد. خانواده دستورالعمل‌های CMPS برای مقایسه مقادیر رشته‌ای استفاده می‌شوند.

دستورالعمل cmpsb برای مقایسه دو دنباله از بایت‌ها که شامل داده مشابه هستند مورد استفاده قرار می‌گیرد. دستورالعمل cmpsb مقادیر موجود در EDI را از مقادیر موجود در ESI کم کرده و پرچم های وضعیت را به روز رسانی می‌کند. این دستورالعمل معمولا با استفاده از پیشوند repe استفاده می‌شود. وقتی این دستورالعمل با پیشوند repe همراه شود، دستورالعمل cmpsb هر بایت از دو رشته را با هم مقایسه کرده تا زمانی‌که یک اختلاف پیدا کند، سپس عمل مقایسه خود را به اتمام می رساند.

دستورالعمل cmpsb بایت‌های درون ESI را به‌دست آورده و با مقادیر موجود در EDI مقایسه می‌کند و سپس پرچم های وضعیت را تنظیم می‌کند و در پایان ثبات‌های EDI و ESI را یکی کاهش یا افزایش می‌دهد. اگر پیشوند repe وجود داشته باشد، ثبات ECX و پرچم ها بررسی می‌شوند، اگر ECX برابر با صفر و پرچم صفر برابر با مقدار ۰ باشد عملیات تکرار متوقف خواهد شد. این فرمان معادل تابع memcmp در زبان c است.

دستورالعمل scasb برای جستجوی یک مقدار درون دنباله‌ای از بایت‌ها مورد استفاده قرار می‌گیرد. مقدار مورد جستجو در ثبات AL تعریف می‌شود. این دستورالعمل مشابه دستورالعمل cmpsb کار می‌کند با این تفاوت‌که بایت‌های درون ESI را با ثبات AL بجای EDI مقایسه می‌کند. عملیات repe تا زمانی‌که مقدار ثبات ECX برابر صفر شود، ادامه پیدا خواهد کرد. اگر مقدار مورد نظر در دنباله بایت‌های مد نظر ما پیدا شد، ESI محل آن مقدار را ذخیره می‌کند.

از دستورالعمل stosb برای ذخیره مقداری در یک محل که توسط EDI مشخص شده است مورد استفاده قرار می‌گیرد. این دستورالعمل مشابه scasb است، اما بجای جستجو یک بایت، بایت مشخص شده را در محلی که در EDI مشخص شده است ذخیره‌سازی می‌کند. پیشوند REP با scasb برای مقداردهی اولیه بافر حافظه استفاده می‌شود جایی که هر بایت شامل یک مقدار مشابه است. این دستورالعمل معادل تابع memset در زبان برنامه‌نویسی C است. جدول ۱۲-۴ برخی از دستورالعمل‌های rep رایج و توضیح عملیاتشان را نمایش می‌دهد.

جدول ۱۲: مثال هایی از دستورالعمل‌های REP

دستورالعمل	توضیحات
repe cmpsb	از این دستورالعمل برای مقایسه دو بافر داده استفاده می‌شود. ثبات‌های EDI و ESI باید با محل دو بافر و ثبات ECX باید با اندازه بافر تنظیم شود. عمل مقایسه تا زمانی‌که ثبات ECX برابر صفر شود ادامه پیدا خواهد کرد.
rep stosb	برای مقداردهی اولیه تمامی بایت‌های یک بایت با یک مقدار خاص استفاده می‌شود. ثبات EDI شامل محل بافر می‌شود و ثبات AL باید شامل مقدار اولیه شود. این دستورالعمل را اغلب اوقات با xor eax, eax خواهید دید.
rep movsb	معمولا از این دستورالعمل برای کپی کردن بایت‌های یک بافر استفاده می‌شود. ثبات ESI باید با آدرس بافر منبع و ثبات EDI باید با آدرس بافر مقصد تنظیم شود و ثبات ECX هم باید با طول کپی مقداردهی شود. عمل کپی کردن، بایت به بایت تا زمانی‌که ECX برابر صفر شود، ادامه پیدا خواهد کرد.
repne scasb	از این دستورالعمل برای جستجوی یک بافر داده برای یک بایت استفاده می‌شود. ثبات EDI شامل آدرس بافر می‌شود و ثبات AL باید شامل بایتی که به دنبال آن هستید مقداردهی شود و ثبات ECX هم باید با طول بافر

	تنظیم گردد. عمل مقایسه تا زمانی‌که ECX برابر صفر شود یا بایت مد نظر کشف گردد ادامه پیدا خواهد کرد.

زیرا بیشتر بدافزارها اغلب با زبان C نوشته می‌شوند، دانستن نحوه ترجمه تابع اصلی یک برنامه با زبان C به زبان اسمبلی برای ما بسیار مهم و حیاتی است. این دانش نیز به شما کمک می‌کند، هنگامی‌که برنامه C به اسمبلی ترجمه می‌شود، تفاوت آفست‌های آن را درک کنید. یک برنامه استاندارد با زبان C دارای دو پارامتر برای تابع اصل خود است، معمولا قالب آن به شکل زیر می‌باشد.

```
int main(int argc, char ** argv)
```

پارامترهای argc و argv در زمان اجرا تعیین می‌شوند. پارامتر argc یک عدد صحیح است که شامل تعداد پارامترها در خط فرمان، از جمله نام برنامه می‌شود. پارامتر argv یک اشاره‌گر به یک آرایه از رشته‌ها است که شامل پارامترهای تحت خط فرمان می‌شود. خروجی زیر مثالی از یک برنامه تحت خط فرمان است و نتیجه Argc و argv در زمان اجرا برنامه را نمایش می‌دهد.

```
filetestprogram.exe -r filename.txt
argc = 3
argv[0] = filetestprogram.exe
argv[1] = -r
argv[2] = filename.txt
```

لیست ۱: کد یک برنامه ساده با زبان C را نمایش می‌دهد.

```
int main(int argc, char* argv[])
{
        if (argc != 3) {return 0;}
        if (strncmp(argv[1], "-r", 2) == 0){
                DeleteFileA(argv[2]);
        }
```

```
        return 0;
}
```

لیست ۲: یک برنامه با زبان C

لیست ۲-۴ کد زبان C را در قالب کامپایل شده نمایش می‌دهد. این مثال به شما کمک می‌کند، بفهمید چگونه پارامترهای لیست شده در جدول ۱-۴ در زبان اسمبلی قابل دسترس هستند. همان‌طور که در خروجی مشاهده می‌کنید، argc با عدد ۳ (شماره ۱) و argv[1] با پارامتر r- (شماره ۲) در طی استفاده از تابع strncmp مقایسه شده‌اند. به نحوه دسترسی argv[1] توجه کنید : ابتدا محل شروع آرایه در ثبات eax بارگذاری شده و سپس ۴ (آفست) به eax افزوده شده است تا به argv[1] دسترسی پیدا گردد. چون هر یک از ورودی‌ها به آرایه argv یک آدرس به یک رشته می‌باشد و هر آدرس در سامانه‌های ۳۲ بیتی ۴ بایت است، عدد ۴ مورد استفاده قرار گرفته است. با این حال، اگر پارامتر r- در خط فرمان ارائه شود، کد شروع شده (شماره ۳) اجرا خواهد شد و این زمانی است که می بینیم به argv[2] در آفست ۸ مربوط به argv دسترسی ایجاد می‌شود و به عنوان پارامتر در تابع DeleteFileA ارائه شده است.

```
004113CE    cmp     [ebp+argc], 3 ❶
004113D2    jz      short loc_4113D8
004113D4    xor     eax, eax
004113D6    jmp     short loc_411414
004113D8    mov     esi, esp
004113DA    push    2               ; MaxCount
004113DC    push    offset Str2     ; "-r"
004113E1    mov     eax, [ebp+argv]
004113E4    mov     ecx, [eax+4]
004113E7    push    ecx             ; Str1
004113E8    call    strncmp ❷
004113F8    test    eax, eax
004113FA    jnz     short loc_411412
004113FC    mov     esi, esp ❸
004113FE    mov     eax, [ebp+argv]
00411401    mov     ecx, [eax+8]
00411404    push    ecx             ; lpFileName
00411405    call    DeleteFileA
```

لیست ۳: کد اسمبلی تابع اصلی برنامه C

نتیجه‌گیری

داشتن دانش در مورد زبان برنامه‌نویسی اسمبلی و فرایند دیزاسمبلی برنامه‌های کامپایل شده کلیدی برای تبدیل شدن به یک متخصص موفق در زمینه تجزیه و تحلیل بدافزار است. در این قسمت از سلسله مقالات تحلیل بدافزار کی‌پاد، ما مفاهیم مهم معماری x86 که در طی تحلیل بدافزار با آن‌ها مواجه خواهید شد را ارائه کردیم که اگر در حین تحلیل بدافزار با دستورالعمل یا ثبات ناآشنایی رو به رو شدید، بتوانید از آن به عنوان یک راهنما استفاده کنید.

در قسمت‌های بعدی، ما نگاهی به برنامه IDA Pro و البته دیگر دیزاسمبلرهای معروف خواهیم انداخت. IDA یک برنامه فوق‌العاده قدرتمند برای دیزاسمبل کردن برنامه‌های کامپایل شده است و به شما در تحلیل بدافزارها و درک عملیات دیزاسمبلی برنامه‌ها کمک شایانی می‌کند.