جلوگیری از تحلیل پویای نرم افزار با مکانیزم ضددیباگر فارغ از شناسایی ابزار یا محیط تحلیل

*عباسعلی رضایی[1]، نیما نیک جوی تبریزی[2]

[1*] استادیار، گروه مهندسی کامپیوتر و فناوری اطلاعات، دانشگاه پیام نور، **صندوق پستی۳۶۹۷-۱۹۳۹۵ تهران، ایران**

[2] کارشناس ارشد امنیت اطلاعات و تحلیلگر بدافزار، مدرس دوره‌های امنیت اطلاعات دانشگاه صنعتی شریف

چکیده

در سالهای اخیر تقابل بین مهندسی معکوس و روش‌های مقابله با آن، به علت پدید آمدن و تاثیر موضوعات جدید از جمله نفوذ هک، مفهوم تازه‌ای یافته و هر روز شاهد پیدایش روش‌های جدید و متنوعی در حوزه مهندسی معکوس و بالطبع آن روش‌های مقابله با آن هستیم. مقاله پیش رو به معرفی روشی می پردازد که برنامه نویسان را قادر می‌سازد بدون نیاز به تشخیص حضور محیط دیباگرها در حافظه یا استفاده از آسیب پذیری‌های شناخته شده یا شناخته نشده، فقط با طی یک روند منطقی که در حال حاضر فقط در خانواده سیستم عامل ویندوز قابل اجرا می‌باشد از اجرای برنامه مورد نظر در محیط‌های تحلیل اعم از دیباگرها، سندباکس و موارد دیگر ممانعت به عمل آورد.

کلمات کلیدی

ضدتحلیل، ضددیباگ، مکانیزم‌های منطقی، قرص قرمز، آسیب پذیری‌های دیباگرها، ضد مهندسی‌معکوس، سامانه‌های محافظتی

۱- مقدمه

اولین گام برای بررسی دقیق یا به عبارت بهتر تحلیل یک نرم افزار، تحت کنترل داشتن روند اجرای آن می باشد. از این رو استفاده از دیباگر ها برای نیل به این هدف امری کاملا ضروری و بدیهی است. متخصصین امنیت اطلاعات و تحلیلگران بدافزار عمدتا با استفاده از روشهای جدید تحلیل سعی در کاهش زمان ارائه راهکار برای مقابله با بدافزارها دارند و از طرف دیگر بدفزارنویسان با ارائه متدهای پیچیده، روند تحلیل را به هر شکل ممکن روز به روز سخت تر می کنند.

استفاده از روش های ضد دیباگ در این حوزه دلایل بسیار زیادی دارد. اما یکی از دلایل عمده که برنامه نویس های حرفه ای بدافزار و هکرها از این روش ها بهره می برند، افزایش طول عمر یک برنامه به منظور انجام فعالیت خود به صورت مخفیانه است. به عبارت دیگر، آنها ازاین روش ها به منظور اینکه متخصصین امنیت و اطلاعات نتوانند به سادگی از مقاصد عملیاتی بدافزارهای تولیدی آنها اطلاعات به دست آورند از این روش ها استفاده می کنند. لذا این روش ها، فرآیند تجزیه و تحلیل بدافزار را بسیار دشوار می کنند و به سختی می توان متوجه شد که بدافزار چه اهدافی داشته است و چه کسانی آن ها را طراحی کرده اند.

همانطور که ابتدا بیان شد، هدف استفاده از روش های ضد دیباگ کردن، دشوار ساختن فرآیند مهندسی معکوس است اما چگونه یک بدافزار وجود یک دیباگر را تشخیص می دهد؟ بدین منظور روش های بسیار زیادی وجود دارد که در این مقاله به رایجترین آنها پرداخته خواهد شد. با فرض اینکه بدافزار با استفاده از یک روش توانست با موفقیت دیباگر را تشخیص دهد، در ادامه چه کاری انجام خواهد داد؟ اصولا بدافزارها هنگامی که یک دیباگر را تشخیص می دهند، به شکل متفاوتی عمل می کنند تا بتوانند هویت مخرب خودشان را از چشم تحلیلگر مخفی نگه دارند یا در برخی از شرایط موجب می شوند که دیباگر متوقف شود. در بدافزارهای عملیاتی شده در دهه های اخیر، بیشتر از مکانیزم گمراه کننده استفاده می‌شود. یعنی هنگامی که بدافزار وجود یک دیباگر را تشخیص می دهد، به شکل متفاوتی رفتار می کند و تحلیلگر را در شرایطی قرار می دهد که نمی تواند هویت آن را به درستی تشخیص دهد. بعلاوه، دیگر مکانیزم های ضد تحلیل با توجه به وضعیت نابسامان صنعت نرم‌افزار و استفاده از نسخه های

غیرقـانونی و قفـل شکسـته در سـطح جهـانی، بسـیار مـورد توجـه تولیدکننده گان محصولات امنیت نرم افزاری قرار گرفتـه اسـت، تـا بـا استفاده از از این روش ها لایه های امنیتی نرم افزارها را افزایش داده و مدت زمان و به دنبال آن هزینه شکستن قفل نـرم افـزاری محصـولات محافظت شده توسط مجرمین سایبری را افـزایش دهنـد. لـذا در ایـن مقاله رویکرد اصلی بررسی روشهای پیشین که بعضا توسط متخصصین امنیت اطلاعات طراحی یا کشف شـده و برخـی از آنهـا توسـط تولیـد کننده گان بدافزار ابداع شده است، می باشد. بـا اتمـام بررسـی، نقـاط ضعف و قوت هر کدام و در نهایت کارایی هر روش بـه چـالش کشـیده خواه شد. البته این روش نیز کاستی‌های خـود را خواهـد داشـت و در بدترین حالت ممکن، می‌تواند به عنوان یک ایـده جدیـد الهـام بخـش محققین فعال در حوزه امنیت اطلاعات جهت ارائه تکنیـک‌هـای بهتـر باشد؛ در ادامه مقاله در بخـش اول بـه تاریخچـه موضـوع مـورد بحـث نگاهی گذرا خواهیم داشت و با برخی مفاهیم آشنا خواهیم شد. سپس در بخش دوم فعالیت‌های صورت پذیرفتـه در ایـن حـوزه معرفـی مـی شوند و تا جای ممکن سعی می شد نقاط قوت و ضـعف هرکـدام بیـان شود. در بخش سوم روش پیشنهادی ارائه خواهـد شـد و در نهایـت در بخش چهارم به ارزیابی و مقایسه اجمالی روش پیشنهاد شده با روش-های موجود بیان می شود.

۲- کارهای مرتبط

همانطور که پیشتر عنوان شد روش ضـددیباگ یکـی از معمـول‌تـرین روشهای جلوگیری از تحلیل می باشد که امروزه توسط تولید کنندگان سیستم های امنیتی و برنامه نویسان بدافزار به شکل گسترده ای مورد استفاده قرار می گیرد. استفاده از تکنیک های شناسایی محیط دیباگ یکی از ساده ترین و ابتدایی ترین روشهای ضدتحلیل می باشد کـه بـه طور عمده به ۷ بخش عمده تقسیم می شوند که در ادامه معرفی مـی-شوند[۳] [۱] [٤] [۱۰]:

۱. تکنیک های ضددیباگ مبتنی بر استفاده از توابع سیستمی (API-Based).
۲. استفاده از وقفه ها و خطاهای مدیریت نشـده کـه سیسـتم عامل توانایی مدیریت آنها را دارد؛ ولی نرم افزار دیبـاگر بـه خاطر ضعف در طراحی یا تولیـد، توانـایی مـدیریت آنهـا را ندارد (Exception-Based) .
۳. استفاده از ساختار فرآیند ها و نخ؛ طی این روند بخشهایی از ساختار فرآیندها که توسط سیستم عامـل مـدیریت شـده و شامل اطلاعات مفیـدی از تحـت دیبـاگ بـودن یـا نبـودن فرآیند است، اسـتفاده مـی شـود (Process and Thread Block Checking).
٤. بررسـی بَـرِخط کـدها و حافظـه فرآینـد جـاری در مقابـل تغییـرات ایجـاد شـده توسـط محـیط‌هـای دیبـاگر (RuntimeCode Modification Check).
٥. بررسی سخت افـزار سیسـتم میزبـان بـه جهـت شناسـایی محیط‌های شـبیه سـازی و مقایسـه پـرچم‌هـای پردازشـگر (Register and Hardware based).
٦. بررسی زمان اجرای کدها، به طور منطقی اجرای یک قطعـه کد تحت دیباگر کُندتر از اجرا بدون واسطه و توسط سیستم عامل می‌باشد (Timing Detection).
۷. استفاده از مکانیزم Self-Debugging که طـی آن فرآینـد مورد نظر توسط یک دیباگرِ از پـیش طراحـی شـده، تحـت دیباگ قرار می‌گیرد. از آنجـایی کـه طبـق قـوانین سیسـتم عامل ویندوز یک فرآیند فقط توسط یک دیباگر قابل دیباگ می‌باشد امکان اتصال دیباگر ثانویه ممکن نمی‌شود.

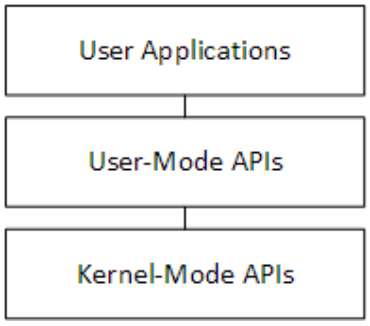
با معرفی اجمالی این تکنیک های عنوان شده در ادامه نقاط ضعف هـر کدام بیان خواهد شد تا در نهایت امکـان مقایسـه بهتـر تکنیـک هـای مرسوم با روش پیشنهادی فراهم شود.

از مزایای روش مبتنی بر استفاده از توابع سیستمی راحتی کاربری آن می‌باشد، که موجب شـده اسـت برنامـه نـویس بـه راحتـی بتوانـد مکانیزم ابتدایی ضدتحلیل خود را پیاده سازی نماید. لیکن با توجه بـه تغییرات ساختار توابع API طی انتشار نسخه هـای متعـدد از سیسـتم عامل ویندوز، این روش با مشکلاتی مواجه است از جمله نیاز بـه بـروز رسانی مکرر دارد و با توجه به در دسترس عموم بودن مسـتندات ایـن توابع، امکان شناسایی، ردگیری و حذف فراخوانی های فـوق از برنامـه توسط نفوذگر یا تحلیلگر به سادگی مقدور می‌باشد.

استفاده از مکانیزم مبتنی بر خطاهای مـدیریت نشـده در بسـیاری موارد می تواند مفید واقع شود ولی بـا ایجـاد دیباگرهـای سفارشـی و مقاوم سازی آنها در قبال وقفه ها میتوان بدون ایجاد تغییـر در پرونـده اجرایی تمامی روتین‌های ضددیباگ آن را خنثی نمود.

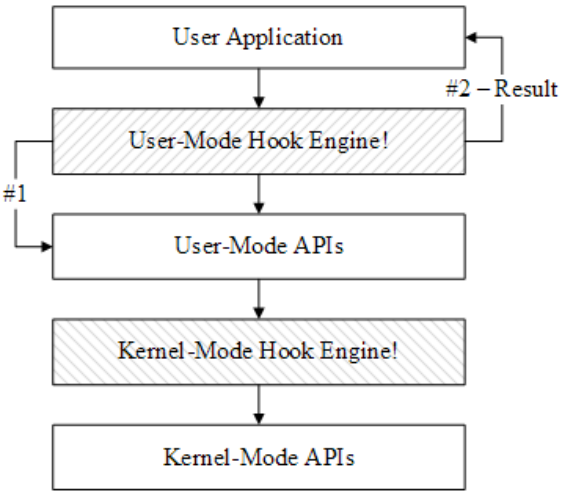
هر پروسه یا فرآیند در سیستم عامل ویندوز شامل ساختمان داده-ای می‌باشد (Process Environment Block). هـر بلـوک محتـوای اطلاعاتی به جهت استفاده سیستم عامل و در صورت نیاز کـاربر مـی-باشد. از این اطلاعات می توان برای دسترسی به مواردی همچـون نـام

فرآیند در حافظه، نام فرآیند والد، مسیر اجرایی فرآیند و مواردی از این دست اشاره داشت؛ به عنوان نمونه در سیستم عامل ویندوز در صورتی که یک فرآیند تحت دیباگ قرار بگیرد فرآیند والد آن، نام فرآیند دیباگر خواهد بود در غیر این صورت فرآیند explorer.exe به عنوان فرآیند والد تشخیص داده می‌شود از اینرو می توان با دسترسی به ساختار داده ای فوق و استخراج اطلاعات آن دریافت که فرآیند والد یک دیباگر است یا خیر. ولی با توجه به معماری مربوط به مدیریت فرآیندهای سیستم عامل ویندوز که در شکل ۱ نمایش داده شده است با قطع دسترسی برنامه کاربردی به توابع API اصلی و جایگزینی آنها با توابع جعلی و یا جعل مقادیر بازگشتی که به مکانیزم Hook شهرت دارد می‌توان در تشخیص صحیح دیباگر یا به عبارت بهتر در دریافت اطلاعات صحیح، اخلال ایجاد کرد. البته لازم به ذکر است که به علت تعدد روشهای عبور از تکنیک شناسایی فوق، تنها به بررسی نمونه عنوان شده به عنوان یک مثال نقض سیستم امنیتی پرداخته و از باقی موارد صرفنظر می شود[۱۲].



شکل ۱ فرایند معمول فراخوانی توابع سیستمی توسط برنامه و مدیریت آن توسط سیستم عامل

همانطور که در شکل (۲) مشخص است، برنامه کاربردی با فراخوانی توابع سیستمی به جای ارجاع به لایه های اصلی سیستم عامل به برنامه موسوم به User-Mode Hook Engine منتقل می‌شوند. در این حالت تمامی خروجی های حاصل از توابع سیستمی می تواند در بخش مذکور دستخوش تغییر شده و به برنامه اصلی برگشت داده شود. بدین ترتیب باعث جلوگیری از فعالیت صحیح برنامه در شناسایی محیط دیباگر می گردد[۷].[۱۱] .



شکل ۲ نمای کلی بعد از قرار گرفتن یک برنامه Hook کننده توابع سیستمی مابین برنامه کاربردی و سیستم عامل

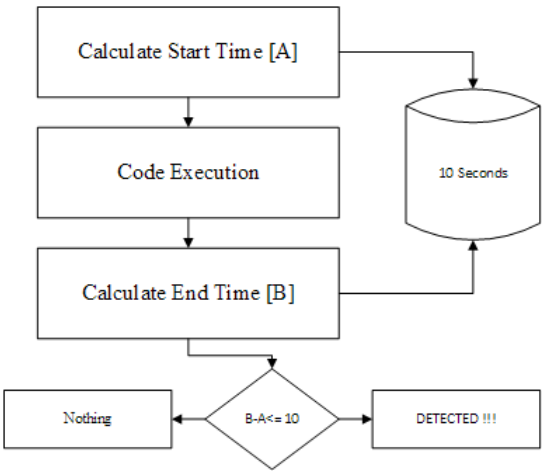
از سایر مکانیزم‌های ضددیباگ که امروزه به ندرت استفاده می‌شود بررسی مرتب مقدار Integrity کُدها در حافظه و پس از اجرا می‌باشد؛ تا در صورت ایجاد نقاط توقف توسط دیباگر یا هر برنامه ناظر امکان شناسایی فراهم گردد. این مکانیزم به علت ایجاد سربار بسیار بالا در سیستم میزبان، کاربرد بسیار کمی دارد و صرفا در موارد خیلی محدود و ابتدایی از بدافزارها به چشم می‌خورد.

روش بعدی مورد بحث، مکانیزم موسوم به استفاده از قرص‌های قرمز می‌باشد که طی آن بدافزار یا نرم‌افزاری که تمایل به حضور در محیط-های شبیه ساز را نداشته باشد با استفاده از آسیب پذیری‌های این محیط‌ها که گاه به علت نوع معماری شبیه ساز و گاه به خاطر اشتباهات برنامه نویسی ایجاد شده اند سوء استفاده کرده و نهایتا حضور خود در محیط واقعی یا شبیه سازی شده را تشخیص می‌دهد. برای کسب درک بهتری از روش مذکور،مثالی ارائه می‌شود. اکثر تحلیلگران بدافزار معمولا از سیستم های مجازی همچون VMWare و Virtual Box و ابزار مشابه استفاده می کنند تا در صورت اجرای سهوی کدهای بدافزار به زیر ساخت اصلی مجموعه آسیب وارد نگردد. لذا بدافزارنویسان نیز برای گریز از تحلیل در محیط‌های ایزوله مجازی، سعی در شناسایی آنها و در نهایت تغییر رفتار خود دارند. این روند، طی ۲ مرحله مجزا صورت می‌پذیرد: مرحله اول استفاده از مکانیزم های ساده شناسایی محیط مجازی با استفاده از حضور درایورها و نام سخت افزارهای شبیه سازی شده در سیستم عامل مهمان می‌باشد و مرحله دوم استفاده از نقاط آسیب پذیری که پیشتر به آنها پرداخته شده است. در ادامه اقدام به شناسایی محیط شبیه سازی شده می کند

که تشریح فرآیند آن از حیطه موضوع بحث خارج می باشد و فقط به معرفی و ارجاع به چند مرجع معتبر بسنده می‌شود.

روش‌های ذکر شده در نوع خود برای سردرگم کردن تحلیلگر، روند زیرکانه‌ای را دنبال می کنند ولی در واقع هیچ کدام روش ضددیباگ به حساب نمی‌آیند. چراکه حتی در صورتی که یک بدافزار یا نرم افزار محافظت شده به تمامی تکنیک های شناسایی محیط شبیه سازی شده مجهز شده باشد باز می توان با اجرای خط به خط و تحلیل عکس العمل برنامه، نقاط شناسایی کننده را کشف و آنها را خنثی نمود و این مساله نقطه ضعف مشترک تمامی تکنیک‌های موجود می‌باشد.

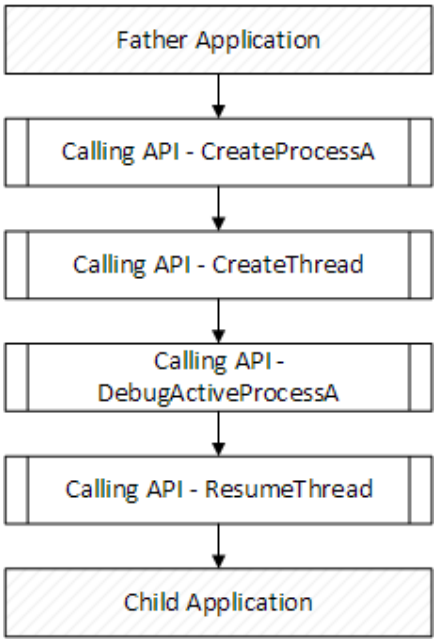
روش Timing Detection روشی تلفیقی است که در شکل (۳) روند کلی آن مشاهه می شود. این روش در صورتی که از توابع سیستمی برای محاسبه زمان استفاده نکند می‌تواند کارآمد باشد ولی با توجه به پیچیدگی بسیار بالای محاسبه زمان بدون استفاده از توابع سیستمی، مشکل موجود در روش شناسایی با استفاده از توابع سیستمی برای این مدل نیز به وجود خواهد آمد و امکان حضور قطعه کد ثالث برای دستکاری مقادیر ارسالی یا بازگشتی وجود خواهد داشت[۱۲] [۱۴].



شکل ۳ فرآیند شناسایی برنامه تحت دیباگ توسط مکانیزم Timing

همانگونه که در شکل ۳ مشخص شده است برنامه نویس ابتدا با اجرای چندین باره کدهای خود، زمان تخمینی از شروع و پایان آنها را نسبت به سخت افزار و محیط اجرا بدست می‌آورد. زمانی این امر قطعیت پیدا می‌کند که زمان اجرای قطعه کد برنامه در صورت عادی بین ۱۰ تا نهایت ۱۵ ثانیه باشد و این مقدار در برنامه ذخیره می‌شود. قبل از اجرای قطعه، کد زمان فعلی سیستم میزبان دریافت می‌شود و سپس قطعه کد برنامه اجرا می‌شود. از آنجاییکه اجرا در محیط یک دیباگر کُندتر از اجرا در حالت بدون واسطه می‌باشد لذا اگر برنامه داخل محیط دیباگر اجرا شود زمان صرف شده بیشتر از مقدار تخمین زده شده خواهد بود. در مرحله آخر بعد از اجرای کد، مجدد زمان سیستم میزبان دریافت می‌شود و با تفریق این دو مقدار، زمان سپری شده برای اجرای قطعه کُد بدست می‌آید. در پایان اگر عدد بدست آمده با مقدار تخمین زده شده مغایرت داشته باشد فرض بر این خواهد بود که برنامه تحت یک دیباگر در حال اجرا می‌باشد[۱۴].

در ادامه مکانیزم Self-Debugging که شباهت زیادی نسبت به روش پیشنهادی دارد معرفی می شود. روش پیشنهادی به نوعی نسخه ارتقا یافته به همراه مدلی نوین از روش پیشین می‌باشد.



شکل ۴ روند فرآیند اجرای مکانیزم Self-Debugging

نخست نقاط قوت مدل ارائه شده در شکل (۴) ارائه می شود:

۱. طبق قوانین سیستم عامل ویندوز، یک فرآیند تنها در یک زمان می‌تواند تحت دیباگ و کنترل یک دیباگر قرار گیرد. لذا با اتصال یک دیباگر جعلی به فرآیند مدنظر، از اتصال دیباگرهای ثانویه ممانعت به عمل می‌آید.
۲. با توجه به مدل پدر- فرزندی در فرآیندها، امکان تبادل اطلاعات مابین آنها به سادگی فراهم می‌شود و از اینرو می‌توان بررسی‌های امنیتی را در قالب دو فرآیند مجزا انجام داد.

۳. این مدل نسبت به مدل‌های پیشین درصد خطای کمتری دارد و با توجه به آزمایشات تجربی، احتمال ناسازگاری کمتری در طول ارتقاء نسل سیستم عامل ویندوز وجود دارد.
٤. مکانیزم «نانومیتس» با الهام از همین شیوه طراحی گردید که در زمان خود یکی از بهترین مکانیزم‌های Anti-Dump به شمار می‌رفت.

در ادامه نقاط ضعف مدل ارائه شده در شکل (۴) بیان می شود:

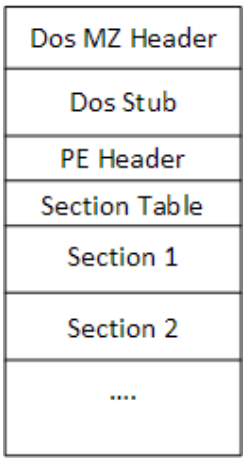
۱. استفاده از توابع سیستمی، خطر سرقت اطلاعات توسط مکانیزم Hook را افزایش می‌دهد.
۲. فرآیند فرزند قبل از اجرا به صورت کامل از حافظه قابل استخراج می‌باشد.
۳. در صورت استفاده از فناوری نانومیتس، با استفاده از ابزار Hook کننده توابع سیستمی، می‌توان تمامی اطلاعات حذف شده را بازیابی کرد.
٤. در صورت وجود پردازش سنگین در فرآیند فرزند، بار پردازشی به فرآیند پدر نیز منتقل خواهد شد که این امر باعث می‌شود میزان سربار پردازشی دو برابر شود.
٥. روش فوق توسط تمامی محصولات ضد بدافزار به عنوان یک رفتار پرخطر شناسایی می‌شود.
٦. با حذف فراخوانی تابع DebugActiveProcessA از فرآیند والد، فرآیند فرزند بدون حضور دیباگر و بدون هیچ گونه مشکلی اجرا شده و در برابر تحلیل آسیب پذیر خواهد بود.

۳- روش پیشنهادی

روش پیشنهادی طی ۳ گام ارائه خواهد شد. در گام نخست به معرفی اجمالی از ساختار فایل اجرایی استاندارد در سیستم عامل ویندوز پرداخته می شود. در گام دوم مکانیزم تزریق کُد به صورت ایستا به فایل باینری استاندارد سیستم عامل ویندوز معرفی می شود و در مرحله آخر روش پیشنهادی به همراه ارائه شبه‌کد جهت سهولت بررسی توسط خواننده تشریح می شود.

۳-۱- ساختار فایل اجرایی

فایل قابل حمل یا همان (Portable Executable) قالب اصلی فایل Win۳۲ است. مشخصات آن تاحدی از قالب عمومی شیء فایل یونیکس(Unix COFF) مشتق شده است.
فایل قابل حمل یعنی اینکه آن فایل بین همه سکوهای Win۳۲ قابل اجراست. بارکننده PE در هر سکوی Win۳۲ ، این قالب را تشخیص داده و اطلاعات فایل اجرایی را در حافظه، حتی زمانیکه ویندوز در سکوهای CPU ای غیر از Intel اجرا شود بارگذاری می کند. اما این بدین معنی نیست که فایل قابل حمل می تواند بدون تغییر در همه CPU ها اجرا شود. هر فایل قابل اجرای Win۳۲ از قالب فایل PE استفاده می کند. حتی راه اندازهای هسته ویندوز NT از این قالب فایل استفاده می کند. لذا قالب فایل PE اطلاعات مفیدی در مورد ساختار ویندوز ارائه می دهد.



شکل۵ طرح کلی قالب فایل PE

همه فایلهای PE حتی کتابخانه های ۳۲ بیتی باید با یک عنوان DOS MZ شروع شوند. این قسمت برای زمانی پیش بینی شده است که برنامه در سیستم عامل DOS اجرا شود. بنابراین DOS این فایل را یک فایل قابل اجرا تشخیص داده و می تواند تابع برنامه کوتاه DOS را که بعد از عنوان MZ ذخیره شده است، اجرا کند. بخش یاد شده به عنوان نقطه آغازین یک فایل PE استاندارد شناخته می شود که از آن به عنوان Entry Point یاد می شود. آدرس فیزیکی EP اولین قطعه ای است که پس از بارگذاری فایل در حافظه اجرا می‌گردد. تابع برنامه کوتاه (DOS Stub) در حقیقت یک EXE ی معتبر هست و در مواردی اجرا می شود که سیستم عامل هیچ چیزی در مورد قالب فایل PE نداند. این تابع فقط پیغام " This program requires Windows" را نمایش می دهد یا اینکه می تواند برنامه کامل DOS باشد، که نوع آن بستگی به نویسنده برنامه دارد. قسمت

DOS Stub به علت منسوخ شدن آن مورد بحث این مقاله نمی باشد. این تابع معمولا بوسیله اسمبلر یا کامپایلر ایجاد می شود. در بسیاری موارد این تابع از سرویس شماره ۹ وقفه H۲۱ استفاده میکند تا پیغام "This program cannot run in DOS mode" را اعلان کند.

بعد از DOS Stub (تابع برنامه کوتاه) قسمت عنوان PE یا همان PE header قرار می گیرد. ساختار PE به نام IMAGE_NT_HEADERS در اصطلاح PE header نامیده می شود. این قسمت محتوی فیلدهایی ضروری می باشد که بارکننده PE از آنها استفاده می کند. اگر سیستم عاملی این قسمت را بشناسد برنامه اجرا خواهد شد. بار کننده PE می تواند نقطه شروع PE header را از قسمت DOS MZ header پیدا نماید و قسمت DOS Stub را نادیده گرفته و مستقیما به قسمت PE Header که عنوان حقیقی فایل (Real File Header) هست، هدایت شود.

محتوای حقیقی فایل PE به قسمتهایی با نام بخش یا Section تقسیم شده است. یک Section چیزی نیست جز یک بلوک داده با خواص معمولی مانند Code/Data و دسترسی‌های Read/Write و موارد دیگر (می توان فایل PE را مانند یک دیسک منطقی فرض کرد). عنوان فایل PE بجای سکتور بوت و Section ها فایلهای روی دیسک هستند. فایلها خواص مختلفی دارند مانند خواص فقط خواندنی، سیستمی، مخفی، بایگانی شده و موارد دیگر. قابل توجه است که گروه بندی داده درون یک Section بر اساس خواص عمومی انجام شده است نه برمبنای منطق. اهمیت ندارد که DATA یا CODE چگونه استفاده می شوند، اگر DATA یا CODE در فایل PE خواص یکسانی داشته باشند، می توانند درون یک Section با هم جمع شده باشند. Section می تواند محتوی CODE و DATA که خواص یکسانی دارند، باشد. اگر بخواهیم یک بلوک از داده خاصیت فقط خواندنی داشته باشد می توان درون یک Section که مشخصه فقط خواندنی دارد، قرار داد. در مواقعی که بارکننده، Section ها را درون حافظه ترسیم می کند، مشخصه های Section ها را می خواند و آن مشخصات را به بلوکهای حافظه اِشغال شده بوسیله Section ها، نسبت می دهد. اگر فایل PE را بصورت یک دیسک منطقی فرض کنیم بطوریکه که عنوان PE، سکتور بوت و Section ها فایلهای روی دیسک باشند، هنوز اطلاعات کافی در مورد محل فایلهای روی دیسک وجود ندارد. یعنی در مورد مسیر و فهرست (Directory) در فایل PE بحث نشده است. بلافاصله بعد از عنوان PE جدول Section قرار گرفته است که آرایه ای از ساختارهاست. هر ساختار محتوی اطلاعاتی درباره هر Section در فایل PE همچون مشخصه آن، برچسب فایل و برچسب مجازی است. اگر پنج Section در یک فایل PE وجود داشته باشد، پنج عنصر در آرایه ساختار وجود دارد. می توان جدول Section را به صورت فهرست ریشه ی (root directory) یک دیسک منطقی فرض نمود که هر عنصر آرایه ی ساختار، معادل با هر فهرست ثبت شده در فهرست ریشه است.

در میان تمام این Section ها، بخش idata حاوی جدول آدرس های وارد شده (IAT) یا بعبارتی جدول تمامی ارجاعات برنامه در فراخوانی توابع و روتین های کتابخانه‌ای پویا می‌باشد. این جدول حاوی آدرس نسبی تمام توابع Import شده به فایل اجرایی است. پس از اجرای یک فایل PE، سیستم عامل بخش مذکور را درون حافظه بارگذاری کرده و این آدرسها را به آدرس صحیح توابع مذکور در حافظه تغییر می‌دهد. دلیل وجود این جدول این است که فایل های اجرایی همیشه در مکان ثابتی از حافظه بارگذاری نمی‌شوند[۲] [۲۰] .

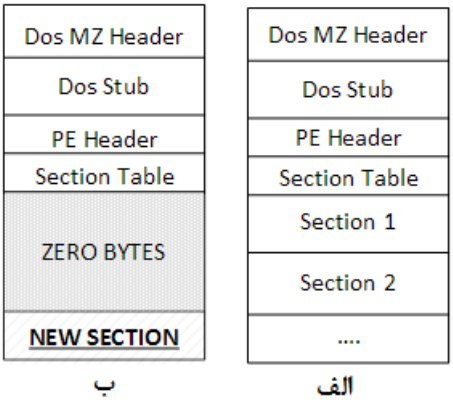
در ادامه مراحل اصلی بارگذاری یک فایل PE در حافظه بیان شده است:

۱. زمانی که فایل PE اجرا شد، بارکننده فایل PE عنوان DOS MZ را برای پیدا کردن «برچسب عنوان» فایل PE بررسی می کند و اگر موفق بود به آن برچسب می رود.
۲. بارکننده، عنوان فایل PE را بررسی می کند اگر معتبر بود، به آخر عنوان فایل PE Header می رود.
۳. بلافاصله بعد از عنوان PE، جدول Section وجود دارد. بارکننده، اطلاعات Section ها را می خواند و درون حافظه ترسیم می کند. همچنین خواص هر Section را مطابق آنچه در جدول Section آمده است، به آنها می دهد.
٤. بعد از ترسیم فایل PE درون حافظه، بارکننده PE به قسمت های منطقی فایل PE مانند Import Table و سایر موارد دیگر می‌پردازد.
٥. در مرحله بعد مقدار آدرس فیزیکی – منطقی مربوطه به Entry Point که قبلا به آن اشاره شد درون مقدار ثبات EIP قرار گرفته و اجرای فایل بارگذاری شده در حافظه شروع می‌شود.

در نهایت کلیه اطلاعات لازم جهت اجرای فایل اجرایی توسط PE loader که جزئی از سیستم عامل می باشد، در حافظه قرار می‌گیرد و فایل PE آماده اجرا می‌گردد.

۲-۳- روش تزریق کد به فایل باینری (ایستا)

تزریق کد به فایل اجرایی در حالت ایستا به چندین روش مختلف صورت می‌پذیرد که ساده ترین آن در این مقاله مورد استفاده قرار گرفته است. لازم به ذکر است که "کد منبع" روش فوق رایگان است و در اختیار همگان قرار دارد. طی این روش کدهای لازم به صورت اسمبلی و در قالب Shell Code به یک Section جدید اضافه شده و مقدار آدرس Entry Point در سرآیند فایل (PE Header) به اولین دستورالعمل جدید که در Section جدید اضافه شده است تغییر پیدا می‌کند. در آخر کدهای اسمبلی نیز با قرار دادن یک پرش کوتاه به آدرس اصلی Entry Point، باعث می‌گردد تا روند اجرای برنامه بعد از اجرای کدهای تزریق شده به دست کدهای اصلی برنامه سپرده شود. در شکل (۶) نمایی از تزریق کد به فایل اجرایی در حالت ایستا نشان داده شده است[۱۰] [۱] [۲] .



شکل ۶ نمایی از ساختار فایل اجرایی استاندارد قبل (الف) و بعد(ب) از تزریق کد به صورت ایستا

همانطور که در شکل (۶) مشخص است می توان از این تکنیک به منظور فشرده سازی فایل اجرایی استفاده نمود؛ بدین صورت که ابتدا اطلاعات تمامی Section ها را خوانده، سپس آنها را رمز یا فشرده کرده و در Section جدید ذخیره می نماید. سپس اطلاعات تمامی Section ها را با مقدار صفر جایگزین شده و کُدهای رمزگشایی اطلاعات فشرده شده را در آدرس آغازین Section جدید قرار می گیرد. با اجرای برنامه، اطلاعات فشرده یا رمز شده در حافظه بارگذاری شده و همانند روش توضیح داده شده پس از اجرای تمامی کدهای تزریق شده روند اجرای برنامه به کدهای اصلی آن در حافظه سپرده می‌شود. مثال عنوان شده یکی از پرکاربردترین، استفاده های تزریق کد به فایل اجرایی می‌باشد.

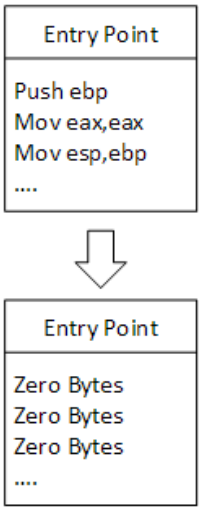
۳-۳- روش پیشنهادی

در ادامه روش پیشنهادی، به صورت مرحله به مرحله همراه با نمایش اشکال بمنظور درک بهتر فرآیند تشریح می شود:

۱. ابتدا مقدار ۱۰۰ بایت از شروع آدرس Entry Point حذف شده و به صورت رمز یا فشرده ذخیره می‌گردد.

۲. سپس هسته دیباگر داخلی که جزئی از لایه محافظتی می-باشد به جریان اطلاعات ایجاد شده در مرحله اول اضافه می‌شود. لازم به ذکر است که تمامی اطلاعات به جهت امنیت بیشتر و ایجاد مانع به جهت مهندسی معکوس ایستا توسط متدهای رمزنگاری پیشرفته از جمله AES ۲۵۶ bit و کلیدهای طولانی و منحصر به فرد هر فایل اجرایی، رمزنگاری می‌شوند.

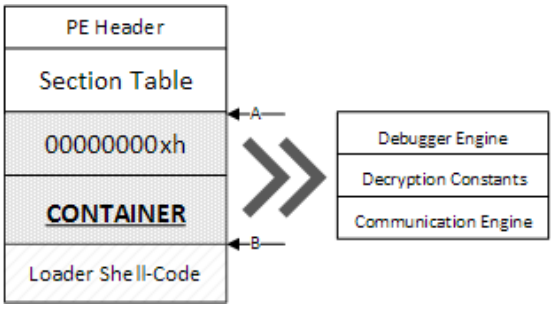
۳. تمامی جریان داده تولید شده در مرحله دوم به عنوان یک Section داده به فایل اجرایی تزریق می‌شود. بدیهی است که اطلاعات مذکور قابلیت خواندنی خواهند داشت و قابل اجرا نخواهند بود.



شکل ۷ حذف مقدار مشخصی از اطلاعات (به عنوان مثال ۳۰ بایت) از آدرس Entry Point فایل اجرایی

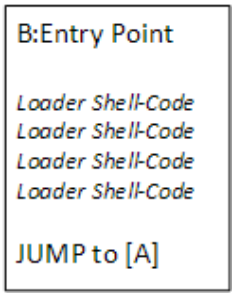
٤. کدهای اسمبلی راه انداز، به عنوان یک Section جدید به فایل اجرایی تزریق شده و آدرس Entry Point به آدرس آغازین کدهای راه انداز تغییر پیدا می‌کند.

٥. با اجرای برنامه، "قطعه کد راه انداز" اطلاعات موجود در Container را در حافظه رمزگشایی کرده و هسته دیباگر داخلی را در حافظه اجرا می‌کند. در این مرحله هسته دیباگر در حالت انتظار در حافظه باقی می‌ماند.



شکل ۸ نمایی از فایل اجرایی پس از تزریق کدهای راه انداز و هسته دیباگر داخلی

همانطور که در شکل (۸) مشاهده می‌شود آدرس مشخص شده توسط پیکان A به آدرس Entry Point اولیه اشاره دارد که ۱۰۰ بایت از اطلاعات آن در مرحله (۱) حذف شده است. مرحله بعدی اضافه کردن هسته دیباگر داخلی، مقادیر رشته‌ای و ماژول ارتباط بین پروسه‌ها تحت عنوان بسته Container می‌باشد که در نهایت با تغییر آدرس Entry Point به محل جدید که با پیکان B اشاره می‌کند، این آدرس در واقع به ابتدای کدهای راه انداز اشاره می‌کند که تحت عنوان یک Section جدید به فایل اجرایی تزریق شده اند و قبل از اجرای برنامه اصلی باعث اجرای محتویات بسته Container می‌شود تا هسته اصلی سامانه در حافظه بارگذاری و باقی مراحل که در ادامه به شرح آنها می‌پردازیم اجرا شوند.

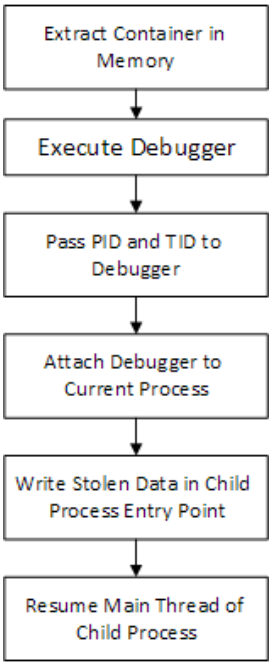


شکل ۹ نمایی از اجرای کدهای مربوطه به راه انداز

در شکل (۹) نیز روند اجرای کدهای راه انداز نمایش داده شده است. در واقع با دوبار کلیک بر روی فایل اجرایی ثانویه (فایلی که اطلاعات به آن با موفقیت تزریق شده است و آماده تست می‌باشد) اجرای کدها از Entry Point ثانویه آغاز می‌گردد. با اجرای کدهای لازم که در بخش های بعدی به آن اشاره می‌شود و بازسازی ۱۰۰ بایت اطلاعات حذف شده، به محل Entry Point اولیه بازمی گردد و روند اصلی برنامه طی می‌شود.

٦. در حال حاضر قبل از اتمام کدهای راه انداز و ارجاع به آدرس Entry Point اصلی فایل اجرایی، وضعیت پروسه آن در حافظه به حالت Suspend تغییر پیدا می‌کند (این وضعیت برای پروسه فرزند توسط کد راه انداز اعمال می گردد). با نزدیک شدن به انتهای کدهای راه انداز، اطلاعاتی اعم از Process ID و Thread ID از پروسه فرزند، توسط فراخوانی هسته ارتباطی بین پروسه، در اختیار پروسه دیباگر داخلی قرار می‌گیرد.

٧. با دریافت اطلاعات پروسه فرزند توسط دیباگر داخلی، پروسه دیباگر با فراخوانی تابع سیستمی DebugActiveProcessA پروسه فرزند را تحت دیباگ خود قرار می‌دهد و تمامی Exception های آن را مدیریت می‌کند. در صورتیکه عملیات اتصال به پروسه فرزند موفقیت آمیز باشد اطلاعات حذف شده در مرحله اول، در حافظه رمزگشایی شده و در آدرس اصلی پروسه فرزند توسط تابع سیستمی WriteProcessMemory بازنویسی می‌شوند. سپس با استفاده از ساختار Context Control، دیباگر مقدار ثبات EIP پروسه فرزند را به آدرس Entry Point اصلی تغییر داده و وضعیت پروسه را از حالت Suspend به Running تغییر می‌دهد. در این حالت برنامه فرزند فعالیت خود را در حافظه آغاز می‌کند.



شکل ۱۰ مراحل اجرای برنامه طی مراحل ۶ و ۷

٤- ارزیابی روش پیشنهادی

برای ارزیابی روش پیشنهادی با سایر روش‌های ضددیباگ و مهندسی معکوس، لازم است ابتدا نگاهی مجدد به ساختار سیستم عامل ویندوز داشته باشیم. در سیستم عامل ویندوز کاربر یک پروسه فعال را فقط توسط یک دیباگر می‌تواند تحت کنترل خود درآورده و فعالیت های آن را مورد بررسی قرار دهد و امکان اتصال دیباگر ثانویه ممکن نیست. از

این رو در روش پیشنهادی با اتصال دیباگر داخلی به برنامه عملا امکان اتصال دیباگر خارجی به پروسه فرزند که همان پروسه تحت حفاظت خواهد بود ممکن نیست. از طرفی شاید به ذهن خطور کند که با قطع دسترسی به تابع سیستمی DebugActiveProcessA می‌توان مانع اجرای فرمان دیباگ کردن پروسه فرزند گردید؛ ولی با توجه به اینکه پروسه فرزند فاقد ۱۰۰ بایت از اطلاعات آدرس است و این اطلاعات توسط پروسه دیباگر پس از اتصال موفق به پروسه فرزند (بعد از بررسی صلاحیت پروسه فرزند توسط مقایسه میزان حافظه اشغال شده یا مقادیر ۱۶ بیتی از آدرس هایی تصادفی از حافظه) رمزگشایی و بازنویسی می‌شود امکان اجرای پروسه فرزند بدون پروسه دیباگر ممکن نیست؛ چرا که این دو پروسه برای اجرا به یکدیگر نیاز دارند و در صورت حذف هر کدام اجرای دیگری ممکن نخواهد بود.

در تصویر (۱۱) شبه کد مربوط به محافظت از فایل اجرایی و در تصویر (۱۲) شبه کد مربوط به مرحله اجرای فایل محافظت شده ذکر شده است.

```
PE = ReadPEFile("Address of Sample PE File");
EP = GetEntryPoint(PE);
EP_Array[100] = ReadBytesfromEP(100);
Rnd_Num = Random(1,GetFileSize);
Encrypted_EP = AES_256(EP_Array,Rnd_Num);

define ShellCode = "Assembly shell-code of loader section" + "E8" + VA
VA = EP;

Define ENC_Container as Data_Container
    ENC_Container.Add(EP_Array);
    ENC_Container.Add(Rnd_Num);

for i=(EP) to i+100
  Write(PE,EP,00000000xh);

Add_Section(PE,".NEW_Section1",@ENC_Container);
Add_Section(PE,".NEW_Section2",@ShellCode);

NEW_Entry_Point = CalculateNewEP();
Change_EntryPoint(PE,NEW_Entry_Point);

RebuiltIAT(PE);
SaveChanges(PE);
```

شکل ۱۱ شبه کد مربوط به مرحله محافظت از فایل اجرایی

```
Define M as MemoryStream;
Define P as String;

M = Read_PE_Section(".NEW_Section1");
M.Extract_In_Memory(Data_Container);
P = M.Extract_In_Memory(P);
M.Decrypt("AES",Data_Container,P);
M.MemoryExecute("Debugger.exe",S_Wait);
Send_IPC_Message("Debugger.exe",CurrentProcessID);
Send_IPC_Message("Debugger.exe",CurrentThreadID);

Suspend(CurrentProcess);
```

شکل ۱۲ شبه کد مربوط به مرحله اجرای دیباگر در حافظه

```
if ( IPD and ITD <> 0 )
{
    DebugActiveProcess(IPD);
    Access_Data_Container(Data_Container);
    Read_Data_Container(IPD,Buffer,Data_Container);
    ReadProcessMemory(IPD,Buffer2,AddressOf(P))
    // P is key
    if (P)
    {
        def Buf = Decrypt_Buffer(Data_Container.ENC_Container,P);
        WriteProcessMemory(IPD,Buf,@Data_Container.ShellCode[20]);
    }
    Context = Full_Control
    GetThreadContext(ITD);
    Context.EIP = Data_Container.ShellCode[20];
    SetThreadContext(ITD);
    ResumeThread(ITD);
}
```

شکل ۱۳ شبه کد مربوط به دیباگر و کنترل پروسه فرزند

مراجع

[۱] Themida Software Protection & Anti-Reverse Engineering Tool, http://www.oreans.com

[۲] Specialized forum about reverse engineering and software security including exploit development and malware analysis, http://www.tuts٤you.com/forum

[۳] Ultimate software protection and licensing tool, http://www.enigmaprotector.com

[٤] Anti-Anti-DebuggerPlugins, https://code.google.com/p/aadp/.

[٥] Adam J. Smith, T. S., USAF. (٢٠١٤). AUTOMATED STATIC DETECTION OF OBFUSCATED ANTI-DEBUGGING TECHNIQUES. *DEPARTMENT OF THE AIR FORCE AIR UNIVERSITY*.

[٦] Craing S, A. A. (٢٠١٠). Packer Analysis Report – Debugging and unpacking the NsPack ٣,٤ and ٣,٧ packer. (SANS).

[٧] Ferrie, P. (٢٠١١). The Ultimate Anti-Debugging Reference.

[٨] Francisco Falcón, N. R. (June ٢٠١٢). Dynamic Binary Instrumentation Frameworks. (Core Security).

[٩] Hao Shi, A. A., Jelena Mirkovic. (٢٠١٥). Cardinal Pill Testing of System Virtual Machines. *USC/Information Sciences Institute*.

[١٠] JaeKeun Lee (٢٠١٤). Evading Anti-debugging Techniques with Binary Substitution. *International Journal of Security and Its Applications, Vol.8, No.1 (2014)*.

[١١] MILLER (٢٠١٢). Binary-Code Obfuscations in Prevalent Packer Tools. (University of Wisconsin).

[١٢] Peter Ferrie, S. A.-V. R., Microsoft Corporation. (٢٠١٠). ANTI-UNPACKER TRICKS. *MSDN*(Microsoft).

[١٣] Ping Chen, C. H., Lieven Desmet, and Wouter Joosen. (٢٠١٦). A comparative study of theuse of anti-debugging and anti-VM techniques in generic and targeted malware. (iMinds-DistriNet).

[١٤] Shields, T. (٢٠١٥). Anti-Debugging – A Developers View. (Veracode Inc., USA).

[١٥] Tengfei Yi, A. Z., Miao Yu, Zhong Ren, Qian Lin, Zhengwei Qi. (٢٠١٢). Anti Debugging Framework Based on Hardware Virtualization. *Shanghai University*(School of Software, Shanghai Jiao Tong University).

[١٦] Vincent, M. (٢٠١٢). The Art of Unpacking. (IBM Internet Security Systems).

[١٧] Xu Chen, J. A., Z. Morley Mao, Michael Bailey, Jose Nazario. (٢٠١٥). Towards an Understanding of Anti-

virtualization and Anti-debugging Behavior in Modern Malware. (University of Michigan).
[١٨] Intel® ٦٤ and IA-٣٢ Architectures Software Developer Manuals. www.intel.com/products/processor/manuals/.
[١٩] O. Yuschuk, Ollydbg. http://www.ollydbg.de/.
[٢٠] A. Honig, Practical Malware Analysis. No Starch Press, ٢٠١٢.