

دانشگاه صنعتی خواجه نصیرالدین طوسی

دانشکده مهندسی کامپیوتر

گزارش گروهی دوم درس تکامل نرم افزار

نویسندگان:

محمد شمس الدینی	محمد اکبرپورجنت	محمدسینا الهکرم
علیرضا پرستار	ارسلان واثق	رضا لشنی زند
معراج سوسن	رضا قنبرزاده	محمد سجاد مهدوی اصل
محمد شیبانی	فاطمه عاصی آتشکاهی	بهنام کاظمی
		نیما گمرکیان

استاد راهنما: دکتر محمدهادی علائیان

چکیده

گزارش حاضر با هدف تحلیل معماری دفاعی نرم افزار، به بررسی روش های متقابل در برابر تحلیل کدهای اجرایی می پردازد. با توجه به اهمیت حیاتی جلوگیری از مهندسی معکوس و سوءاستفاده از مالکیت فکری و آسیب پذیری های امنیتی، توسعه دهندگان به سمت استفاده از تکنیک های ضد دیس اسمبلی (AD) سوق داده شده اند. در این راستا، تکنیک های اصلی AD شامل ابهام سازی کد در سطوح واژگانی، داده ای و جریان کنترل، بهره گیری از دستورات و ساختارهای غیرمعمول و وابسته به حالت پردازنده برای شکستن تحلیل استاتیک، روش های پنهان سازی مبتنی بر تجزیه و تحلیل پویا نظیر رمزگشایی زمان اجرا و تکنیک های ضد دیباگ، و همچنین تکنیک های رمزگذاری مانند کدهای پلی مورفیک و متامورفیک مورد بررسی قرار می گیرند.

این گزارش به فلسفه و تکنیک های ضد دیس اسمبلی (AADA) می پردازد که در پاسخ به پیچیدگی های AD توسط تحلیل گران امنیتی ایجاد شده اند. این تکنیک ها شامل تحلیل دینامیک و رصد کد برای کشف مسیر اجرای واقعی، استفاده از دیس اسمبلرهای پیشرفته با قابلیت های تحلیل هوشمند و اسکریپت نویسی، بهره گیری از اجرای نمادین و شبیه سازی دینامیک برای کاوش در مسیرهای اجرایی مبهم، و وصله زدن و اصلاح کد برای بازسازی نمودار جریان کنترل است.

نتایج نشان می دهند که مقابله با مهندسی معکوس یک «مسابقه تسلیحاتی» دائمی میان روش های دفاعی AD و روش های تحلیلی AADA است. موفقیت در حفظ امنیت و تکامل نرم افزار، مستلزم درک عمیق از هر دو جبهه و استفاده هوشمندانه و ترکیبی از ابزارهای تحلیل استاتیک و دینامیک برای خنثی سازی مؤثر لایه های ابهام سازی است.

فهرست مطالب

۱	چکیده
۵	۱ فرایندهای مهندسی نرم افزار و چرخه های تکامل تا پیدایش
۵	۱.۱ روش های Anti Anti Disassembly
۵	۱.۱.۱ مقدمه ای بر ضد ضد دی اسمبل (Anti-Anti-Disassembly)
۵	۲.۱.۱ تعریف (AD) Anti-Disassembly:
۶	۳.۱.۱ ورود به Anti-Anti-Disassembly (AADA)
۶	۴.۱.۱ اهمیت در تکامل نرم افزار
۶	۵.۱.۱ تکنیک های کلیدی ضد ضد دی اسمبل (AADA)
۸	۲.۱ استفاده از تکنیک های تغییر ساختار کد
۹	۱.۲.۱ دسته بندی تکنیک های مبهم سازی کد
۱۱	۳.۱ استفاده از دستورات و ساختارهای غیرمعمول
۱۱	۱.۳.۱ معرفی دستورات و روش های غیرمعمول
۱۴	۲.۳.۱ انواع دستورات غیرمعمول
۱۴	۳.۳.۱ مکانیزم عملکرد ضد دی اسمبل
۱۵	۴.۳.۱ جداول مقایسه ای
۱۶	۴.۱ پنهان سازی از طریق تجزیه و تحلیل پویا
۱۷	۵.۱ معرفی کدهای خودتغییر (Self-Modifying Code)

۱۸	مثال ساده از کد خودتغییر (Self-Modifying Code)	۱.۵.۱
۱۹	معایب مثال ساده SMC	۲.۵.۱
	مثال علمی از SMC وابسته به زمان اجرا (Time-Dependent Self-Modifying Code)	۳.۵.۱
۲۰		
۲۱	مقاومت در برابر دیس اسمبلی با استفاده از تکنیک‌های رمزگذاری	۶.۱
۲۱	کد پلی‌مورفیک (Polymorphic Code)	۱.۶.۱
۲۲	کد متامورفیک (Metamorphic Code)	۲.۶.۱
۲۳	بسته‌سازی کد (Packers)	۳.۶.۱
۲۴	ابهام‌زایی مبتنی بر مجازی‌سازی (Virtualization-Based Obfuscation)	۴.۶.۱
۲۵	رمزگذاری کد در زمان اجرا (Runtime Code Encryption)	۵.۶.۱
۲۷	۲ مشکلات مطرح در چرخه‌های توسعه و تکامل نرم‌افزار	
۲۷	روش‌های Anti Anti Disassembly	۱.۲
۲۷	مقدمه ای بر ضد ضد دی‌اسمبل (Anti-Anti-Disassembly)	۱.۱.۲
۲۷	تعریف (AD) Anti-Disassembly:	۲.۱.۲
۲۸	ورود به Anti-Anti-Disassembly (AADA)	۳.۱.۲
۲۸	اهمیت در تکامل نرم‌افزار	۴.۱.۲
۲۸	تکنیک‌های کلیدی ضد ضد دی‌اسمبل (AADA)	۵.۱.۲
۳۰	استفاده از دیس اسمبلرهای پیشرفته	۲.۲
۳۰	دلیل نام‌گذاری پیشرفته برای این نوع اسمبلرها	۱.۲.۲
۳۱	استفاده از تحلیل رفتار اجرایی	۳.۲
۳۱	معماری آزمایشگاه و متدولوژی پایش	۱.۳.۲
۳۳	چالش‌های ضد دیباگ و راهکارهای عبور (Anti-Anti-Debugging)	۲.۳.۲
۳۴	تکنیک‌های رمزگشایی و تحلیل کدهای رمزگذاری‌شده (Decryption Techniques)	۴.۲
۳۴	فرآیند عمومی آنپک کردن (Generic Unpacking)	۱.۴.۲

۳۵	بازسازی جدول آدرس واردات (IAT Reconstruction)	۲.۴.۲
۳۶	رمزگشایی الگوریتم‌های اختصاصی و رشته‌ها	۳.۴.۲
۳۷	شبیه‌سازی و اجرای داینامیک	۵.۲
۳۸	استفاده از شبیه‌سازی‌های داینامیک برای تحلیل کدهای اجرایی	۱.۵.۲
۳۸	استفاده از روش‌های شکست کدهای خود تغییر	۶.۲
۳۹	تکنیک‌هایی که به شکستن کدهایی که خودشان را تغییر می‌دهند کمک می‌کند	۱.۶.۲
۴۰	مقاومت در برابر تحلیل‌های ضد ضد دیس اسمبلی	۷.۲
۴۰	لایه‌بندی چندمرحله‌ای ابهام‌سازی	۱.۷.۲
۴۱	رفتار آگاه از تحلیل	۲.۷.۲
۴۲	وابستگی به سخت‌افزار واقعی	۳.۷.۲
۴۲	جریان کنترل غیرقطعی	۴.۷.۲
۴۳	کد خودتغییردهنده چندمرحله‌ای	۵.۷.۲
۴۳	جمع‌بندی	۶.۷.۲

۴۴	نتیجه‌گیری	۳
----	------------	---

فصل ۱

فرایندهای مهندسی نرم افزار و چرخه های تکامل تا پیدایش

۱.۱ روش های Anti Anti Disassembly

۱.۱.۱ مقدمه ای بر ضد ضد دی اسمبل (Anti-Anti-Disassembly)

در دنیای مدرن نرم افزار، کد منبع (Source Code) هسته اصلی مالکیت فکری و ارزش تجاری هر محصولی است. با افزایش پیچیدگی سیستم ها و گسترش تهدیدات امنیتی، نیاز به جلوگیری از مهندسی معکوس (Reverse Engineering)، سرقت فکری، و تحلیل بدافزارها اهمیت حیاتی یافته است. این نیاز، توسعه دهندگان نرم افزار را به سمت استفاده از تکنیک هایی تحت عنوان اُبفوسکیشن (Obfuscation) و ضد دی اسمبل (Anti-Disassembly - AD) سوق داده است.

۲.۱.۱ تعریف Anti-Disassembly (AD):

Anti-Disassembly مجموعه ای از ترفندها و الگوهای برنامه نویسی است که عمداً برای گمراه کردن ابزارهای تحلیل استاتیک (مانند دی اسمبلرها و دکامپایلرها) طراحی شده اند. این تکنیک ها با هدف شکستن نمودار جریان کنترل (Control Flow Graph - CFG) و جلوگیری از تفسیر صحیح دستورات ماشین توسط ابزارهای خودکار به کار می روند. روش هایی مانند قرار دادن بایت های فیک، پرش های نامعتبر، یا پرش های داینامیک محاسبه شده، تحلیل کد را برای انسان بسیار دشوار و زمان بر می سازند.

۳.۱.۱ ورود به Anti-Anti-Disassembly (AADA)

ضد ضد دی اسمبل (Anti-Anti-Disassembly - AADA)، در پاسخ به تکنیک های AD، به وجود آمده است. AADA نه تنها یک ابزار، بلکه یک فلسفه مقابله ای در مهندسی معکوس است. این حوزه به مجموعه ای از روش های پیشرفته و هوشمندانه اشاره دارد که توسط متخصصان امنیت و تحلیلگران بدافزار برای شناسایی، تجزیه و خنثی کردن عمدی لایه های محافظتی و مبهم سازی کد ایجاد شده اند. هدف غایی AADA این است که به طور موفقیت آمیز شفافیت (Clarity) و دقت (Accuracy) تحلیل را به کد بازگرداند. این فرآیند اغلب شامل تحلیل دینامیک (اجرای کنترل شده کد)، اجرای نمادین (Symbolic Execution) و اصلاح خودکار فایل های باینری (Patching) است تا موانع AD برداشته شده و تحلیلگر بتواند به منطق اصلی برنامه دست یابد.

۴.۱.۱ اهمیت در تکامل نرم افزار

در چارچوب درس تکامل نرم افزار (Software Evolution)، AADA نقش حیاتی در فرآیندهای نگهداری پیشگیرانه (Preventive Maintenance) و ممیزی امنیتی (Security Auditing) ایفا می کند. دانش AADA به تیم های توسعه و امنیت این امکان را می دهد که:

- بدافزارها را تحلیل کنند: برای درک نحوه عملکرد کدهای مخرب که از AD برای پنهان شدن استفاده می کنند.
- اثربخشی محافظت را ارزیابی کنند: برای تست و ارزیابی مقاومت تکنیک های حفاظت از کد خود در برابر پیشرفته ترین روش های شکستن آن.

۵.۱.۱ تکنیک های کلیدی ضد ضد دی اسمبل (AADA)

تکنیک های AADA به طور مستقیم هدف گذاری شده اند تا موانعی را که روش های Anti-Disassembly (AD) برای گمراه کردن تحلیل استاتیک ایجاد کرده اند، از بین ببرند. موفقیت AADA به توانایی تحلیلگر در ترکیب ابزارهای استاتیک و دینامیک بستگی دارد.

۱. تحلیل دینامیک و رصد (Dynamic Analysis and Tracing)

بسیاری از تکنیک های AD به متغیرهای وابسته به زمان اجرا (Runtime) تکیه دارند (مانند پرش های محاسبه شده). تحلیل دینامیک مؤثرترین روش برای غلبه بر این موانع است.

- استفاده از اشکالزدها (Debuggers) اجرای کد در یک محیط کنترل شده (Sandbox) یا یک اشکالزدای سطح پایین مانند x64dbg یا GDB به تحلیلگر اجازه می دهد تا مقادیر رجیسترها و حافظه را در لحظه اجرای دستورات حیاتی مشاهده کند.
 - ردیابی دستورات (Instruction Tracing) در این روش توالی دقیق دستوراتی که در طول اجرای بخش مبهم سازی شده اجرا می شوند ثبت می گردد. این ردیابی می تواند آدرس های پرش های دینامیک را آشکار سازد و مسیر واقعی جریان کنترل را نشان دهد.
 - استخراج آدرس های مقصد (Destination Address Extraction) در مورد پرش های محاسباتی مانند JMP [Register] یا CALL [Register]، تحلیل دینامیک مقدار نهایی رجیستر را در لحظه پرش کشف می کند و آدرس مقصد واقعی را به صورت صریح به دست می دهد.
۲. وصله زدن و اصلاح کد (Patching and Code Modification) هنگامی که مقصد واقعی جریان کنترل کشف می شود، تحلیلگر اغلب بایت های اصلی فایل باینری را برای تسهیل تحلیل های بعدی اصلاح می کند.
- NOPing دستورات فیک دستوراتی که توسط AD برای خراب کردن دی اسمبلر استفاده شده اند، با دستور NOP (No Operation) جایگزین می شوند. این کار دستورات را غیرفعال می کند اما اندازه کد را ثابت نگه می دارد و دی اسمبلر می تواند به درستی از دستورات فیک عبور کند.
 - تبدیل پرش دینامیک به پرش استاتیک پس از کشف آدرس مقصد پرش دینامیک، تحلیلگر می تواند دستورات پیچیده را با یک دستور پرش استاتیک و مستقیم (JMP یا CALL) به آدرس کشف شده جایگزین کند. این کار به ابزارهای تحلیل استاتیک اجازه می دهد تا نمودار جریان کنترل (CFG) را به درستی بازسازی کنند.
 - خنثی سازی تکنیک های ضد دی باگ حذف دستوراتی که عمداً برای شناسایی اشکالزدا و توقف برنامه (Anti-Debugging) طراحی شده اند.
۳. اجرای نمادین (Symbolic Execution) اجرای نمادین یک تکنیک پیشرفته است که به جای مقادیر واقعی ورودی، از نمادها استفاده می کند. این روش برای شکستن ابفوسکیشن های سنگین که شامل عملیات ریاضی پیچیده هستند، بسیار قدرتمند است.
- تحلیل نمادین پرش ها در سناریوهایی که آدرس پرش با عملیات ریاضی پیچیده ای بر روی متغیرهای داخلی محاسبه می شود، اجرای نمادین می تواند فرمول ریاضی محاسبه آدرس را به دست آورد و تمام مقادیر ممکن را تعیین کند.

• ابزارهای پیشرفته ابزاریهایی مانند Angr و Triton از اجرای نمادین برای کاوش خودکار در مسیرهای اجرایی کد مبهم سازی شده و کشف مقاصد پنهان استفاده می کنند.

۴. اسکریپت نویسی خودکار (Automated Scripting) برای مقابله با حجم بالای اُبفوسکیشن و تکنیک های تکراری AD، تحلیلگران از قابلیت اسکریپت نویسی دی اسمبلرها استفاده می کنند.

• اسکریپت های تخصصی نوشتن اسکریپت های (مانند IDAPython) که یک الگوی خاص AD را در کد شناسایی می کنند و به طور خودکار آن را خنثی می کنند (به عنوان مثال، اسکریپتی که به دنبال توالی بایت های خاص برای یک دستور نامعتبر می گردد و بلافاصله آن را NOP می کند).

• بازسازی CFG این اسکریپت ها اطلاعاتی را که از تحلیل دینامیک به دست آمده اند، در تحلیل استاتیک دی اسمبلر وارد می کنند تا نمودار جریان کنترل مجدداً با دقت بالا ترسیم شود.

۲.۱ استفاده از تکنیک های تغییر ساختار کد

مهندسی معکوس، به مجموعه ای از تکنیک ها اطلاق می شود که به تحلیل گر اجازه می دهد با استفاده از ابزارهایی مانند دیس اسمبلرها و دی کامپایلرها، فایل اجرایی را به یک نمایش سطح پایین تر یا حتی یک بازسازی نزدیک به کد منبع اصلی تبدیل کند. این امر درهای سوءاستفاده را به روی افراد مخرب باز می کند؛ از جمله دور زدن مکانیزم های حفاظتی نرم افزار، سرقت مالکیت معنوی برای ساخت محصولات رقیب، و شناسایی آسیب پذیری های امنیتی برای بهره برداری های بعدی.

به منظور ایجاد یک مانع جدی در برابر این تهدیدات، از رویکردی به نام مبهم سازی کد استفاده می شود. مبهم سازی فرآیندی است که طی آن، کد برنامه به نسخه ای دیگر تبدیل می شود که از نظر عملکردی کاملاً با نسخه اصلی یکسان است، اما درک، تحلیل و دنبال کردن منطق آن برای یک انسان یا حتی ابزارهای خودکار، به شدت دشوار و پیچیده می گردد. هدف نهایی این تکنیک، غیرممکن ساختن مطلق مهندسی معکوس نیست، بلکه افزایش هزینه، زمان و سطح تخصص مورد نیاز برای آن است، تا جایی که این فرآیند برای مهاجم از نظر عملی و اقتصادی فاقد صرفه باشد. تکنیک های مبهم سازی را می توان در دسته بندی های مختلفی مورد بررسی قرار داد.

۱.۲.۱ دسته بندی تکنیک های مبهم سازی کد

این تکنیک ها بر اساس جنبه ای از برنامه که هدف قرار می دهند، به سه دسته اصلی تقسیم می شوند: مبهم سازی واژگانی، مبهم سازی داده ها و مبهم سازی جریان کنترل.

۱. مبهم سازی واژگانی

این دسته از تکنیک ها بر روی ساختار سطحی و نوشتاری کد و اطلاعات مرتبط با آن تمرکز دارند و به عنوان اولین لایه دفاعی عمل می کنند.

تغییر نام شناسه ها: این روش یکی از اساسی ترین و در عین حال مؤثرترین تکنیک ها است. توسعه دهندگان به صورت طبیعی از اسامی معنادار برای اجزای مختلف کد مانند متغیرها، توابع و کلاس ها استفاده می کنند تا خوانایی و قابلیت نگهداری کد را تضمین کنند. این اسامی، سرخ های معنایی بسیار ارزشمندی را در اختیار تحلیل گر قرار می دهند. تکنیک تغییر نام، به صورت خودکار این اسامی با مفهوم را با شناسه های کوتاه، بی معنی و غیرقابل پیش بینی جایگزین می کند. در نتیجه، کدی که توسط دی کامپایلر بازسازی می شود، فاقد هرگونه زمینه معنایی است و تحلیل گر را مجبور می سازد تا برای درک عملکرد هر بخش، با صرف زمان و انرژی بسیار زیاد، جریان اجرای برنامه را به صورت دستی ردیابی کند و نقش هر عنصر را حدس بزند.

حذف اطلاعات فراداده ای و اشکال زدایی: فایل های اجرایی، به خصوص در پلتفرم های مدیریت شده، اغلب حاوی اطلاعات اضافی به نام فراداده هستند. این اطلاعات شامل ساختار کلی برنامه، نام کلاس ها، فضا های نام و همچنین اطلاعات اشکال زدایی است که می تواند مسیر تحلیل را برای مهندس معکوس بسیار هموار کند. حذف این اطلاعات غیرضروری، تحلیل گر را از یک دید کلی و ساختاریافته نسبت به برنامه محروم کرده و او را مجبور به تحلیل در سطح پایین تر و جزئی تر می کند.

۲. مبهم سازی داده ها

این تکنیک ها بر نحوه ذخیره سازی، نمایش و ساختار داده ها در برنامه تمرکز دارند تا فهم مقادیر و روابط بین آن ها را مختل کنند.

رمزنگاری رشته ها: رشته های متنی ثابت که در کد برنامه وجود دارند، اهداف اولیه ی تحلیل گران هستند. این تکنیک، تمام این رشته ها را در زمان کامپایل به فرمتی غیرقابل خواندن تبدیل کرده و در فایل اجرایی ذخیره می کند. در زمان اجرا، تنها در لحظه ای که برنامه به یک رشته نیاز دارد، یک روتین مخصوص آن را به حالت اولیه بازمی گرداند. این فرآیند باعث می شود که تحلیل گر نتواند با یک جستجوی ساده به اطلاعات حساس دست یابد و ابتدا باید مکانیزم رمزگشایی را شناسایی و مهندسی معکوس کند.

تبدیل ساختار داده ها: ساختارهای داده منطقی مانند کلاس ها، داده های مرتبط را به صورت یکپارچه سازماندهی می کنند. این تکنیک این پیوندهای منطقی را از هم می گسلد. به عنوان مثال، یک ساختار داده ای منسجم می تواند به چندین آرایه جداگانه و نامرتب تبدیل شود. این امر درک مدل داده ای برنامه و نحوه ارتباط اجزای مختلف داده با یکدیگر را برای تحلیل گر فوق العاده دشوار می سازد.

کدگذاری متغیرها: به جای ذخیره سازی مقدار یک متغیر به صورت مستقیم، مقدار آن تحت یک تبدیل منطقی یا حسابی قرار گرفته و به آن شکل ذخیره می شود. هر زمان که برنامه نیاز به خواندن یا نوشتن مقدار واقعی متغیر داشته باشد، ابتدا باید تبدیل معکوس آن را اجرا کند. این کار باعث می شود که مشاهده مستقیم مقادیر متغیرها در حافظه در حین فرآیند دیباگ، اطلاعات مفیدی را در اختیار تحلیل گر قرار ندهد، زیرا مقادیر مشاهده شده، مقادیر واقعی نیستند.

۳. مبهم سازی جریان کنترل

این دسته از تکنیک ها، که از پیچیده ترین و قوی ترین روش ها هستند، منطق و مسیر اجرای دستورات برنامه را هدف قرار می دهند.

درج کدهای بی اثر و گزاره های مبهم: در این روش، دستورات و بلوک های کدی به برنامه اضافه می شوند که هیچ تأثیر واقعی بر خروجی نهایی آن ندارند. شکل پیشرفته تر این تکنیک، استفاده از گزاره های مبهم است. یک گزاره مبهم، یک عبارت شرطی است که نتیجه آن همواره ثابت است، اما به گونه ای طراحی شده که ابزارهای تحلیل استاتیک قادر به تشخیص این ماهیت ثابت نباشند. این گزاره ها برای ایجاد انشعاب های کنترلی جعلی در برنامه استفاده می شوند که تحلیل گر انسانی و ابزارهای خودکار را به مسیرهای تحلیلی اشتباه و بی نتیجه هدایت می کنند.

صاف سازی جریان کنترل: این یک تکنیک بسیار قدرتمند است که ساختارهای منطقی و قابل فهم برنامه مانند حلقه ها و دستورات شرطی را به طور کامل از بین می برد. در این روش، بدنه یک تابع به بلوک های کد کوچکتر تقسیم می شود. سپس تمام این بلوک ها در داخل یک ساختار کنترلی مرکزی قرار می گیرند. یک متغیر حالت تعیین می کند که در هر لحظه کدام بلوک کد باید اجرا شود و هر بلوک پس از اجرا، این متغیر حالت را برای تعیین بلوک بعدی به روزرسانی می کند. نتیجه، یک ساختار اجرایی شبیه به کلاف سردرگم است که فاقد هرگونه جریان منطقی قابل تشخیص بوده و دنبال کردن آن برای تحلیل گر تقریباً غیرممکن است.

۳.۱ استفاده از دستورات و ساختارهای غیرمعمول

دستورات و ساختارهای غیرمعمول یکی از تکنیک های مؤثر در Anti-Disassembly هستند که با بهره گیری از ویژگی های خاص معماری پردازنده، تحلیل استاتیک کد را برای دیس اسمبلرها با چالش مواجه می کنند.

۱.۳.۱ معرفی دستورات و روش های غیرمعمول

دستورات غیرمعمول به دستوراتی اطلاق می شود که یا به ندرت در کدهای عادی استفاده می شوند یا رفتار پیچیده ای دارند که پردازش آن ها برای دیس اسمبلرها دشوار است.

دستورات شرطی با شرایط پیچیده

مکانیزم عملکرد: در این تکنیک، از ترکیب دستورات شرطی و محاسبات پیچیده برای ایجاد مسیرهای اجرایی مبهم استفاده می شود. دیس اسمبلرها که معمولاً بر پایه تحلیل خطی کد کار می کنند، در تشخیص مسیر صحیح اجرا دچار اشتباه می شوند. برای مثال:

```
; Example of complex conditional jump
cmp eax, ebx
jz normal_path
jmp unusual_path
```

```

unusual_path:
; Unusual instructions sequence
pushf
pop ax
and ax, 0x0FFF
jmp complex_calc

complex_calc:
; Complex calculation that confuses disassemblers
mov ecx, eax
ror ecx, 3
xor ecx, 0xDEADBEEF

```

شرح کد: در این کد، دیس اسمبلر در تشخیص مسیر اجرای واقعی دچار سردرگمی می شود. بخش ابتدایی با مقایسه رجیسترها و پرش شرطی یک تصمیم گیری ساده نشان می دهد، اما مسیر unusual-path با خواندن فلگ های پردازنده (pushf/pop) و ایجاد محاسبات پیچیده (ror/xor)، شرایط اجرایی مبهمی ایجاد می کند که وابسته به داده های پویای زمان اجراست. این پیچیدگی باعث می شود دیس اسمبلر نتواند مسیرهای اجرایی را به درستی شناسایی کند.

دستورات FPU غیرمعمول

دستورات واحد محاسبات ممیز شناور (FPU) به دلیل پیچیدگی ذاتی، گزینه مناسبی برای ایجاد سد در برابر دیس اسمبلرها هستند:

```

; Unusual FPU instructions
fldz                ; Load +0.0
fchs                ; Change sign
fstp st(1)          ; Unusual stack operation
fcomip st(0), st(1) ; Compare and pop

```

شرح کد: این دنباله دستورات واحد ممیز شناور (FPU) با ایجاد عملیات غیرمعمول روی پشته FPU، دیس اسمبلرها را گمراه می کند. مکانیزم کار به این صورت است که ابتدا مقدار صفر به پشته بارگذاری شده، سپس علامت آن تغییر می کند که یک عمل غیرمعمول محسوب می شود. در ادامه با ذخیره و خارج کردن غیراستاندارد از پشته و مقایسه همراه با حذف عناصر، یک الگوی استفاده پیچیده از پشته FPU ایجاد می شود که بسیاری از دیس اسمبلرها در تحلیل صحیح آن دچار مشکل می شوند.

استفاده از دستورات خود-تغییردهنده کد

یکی از پیچیده ترین تکنیک های ضد دیس اسمبل، استفاده از کد خود-تغییردهنده است که در آن برنامه در حین اجرا، بخشی از کد خود را تغییر می دهد. در این تکنیک، برنامه در زمان اجرا دستورات خود را اصلاح می کند که این کار تحلیل استاتیک را غیرممکن می سازد. دیس اسمبلرها که معمولاً کد را به صورت استاتیک تحلیل می کنند، قادر به پیش بینی تغییرات پویای کد نیستند.

```
; Self-modifying code example
section .data
code_buffer db 0x90, 0x90, 0x90 ; NOP instructions

section .text
mov esi, code_buffer
mov byte [esi], 0xB8 ; MOV EAX, immediate
mov dword [esi+1], 0x12345678 ; Immediate value
mov byte [esi+5], 0xC3 ; RET instruction
jmp code_buffer ; Execute modified code
```

شرح کد: این کد با تغییر پویای دستورات در حافظه، دیس اسمبلرها را به طور کامل گمراه می کند. در ابتدا بافر کد حاوی دستورات بی اثر NOP است، اما در حین اجرا به دنباله ای از دستورات MOV EAX و RET تبدیل می شود. دیس اسمبلر که تنها محتوای اولیه بافر را می بیند، قادر به تشخیص کد نهایی اجرا شده نخواهد بود.

۲.۳.۱ انواع دستورات غیرمعمول

دستورات با رفتار وابسته به حالت

این دستورات رفتار متفاوتی بر اساس حالت فعلی پردازنده دارند:

- ENTER و LEAVE با پارامترهای غیرمعمول

- دستورات BOUND برای بررسی محدوده آرایه

- دستورات SALC (Set AL on Carry)

دستورات مدیریت رشته ها

دستورات رشته ای به دلیل وابستگی به رجیسترها می توانند پیچیدگی ایجاد کنند:

- REP MOVSB با تنظیمات غیرمعمول رجیسترها

- SCASB با مقادیر جستجوی پیچیده

- CMPSW با اندازه های غیراستاندارد

دستورات سیستم و ممیزی

استفاده از دستورات سطح سیستم می تواند دیس اسمبلرها را دچار خطا کند:

- IN و OUT برای دسترسی به پورت ها

- دستورات SLDT, SIDT, SGDT

- دستورات VERW, VERR

۳.۳.۱ مکانیزم عملکرد ضد دیس اسمبل

دستورات غیرمعمول از چند طریق بر دیس اسمبلرها تأثیر می گذارند:

ایجاد خطا در تحلیل جریان کنترل

- شکستن تحلیل جریان خطی: با ایجاد پرش های غیرقابل پیش بینی
- ایجاد بلوک های کد مبهم: با ساختارهای شرطی پیچیده
- مختل کردن تحلیل داده: با دستوراتی که وابستگی داده ای پیچیده ایجاد می کنند

سوءاستفاده از محدودیت های دیس اسمبلر

- عدم پشتیبانی از دستورات نادر: برخی دیس اسمبلرها دستورات کم کاربرد را نمی شناسند
- خطا در تحلیل حالت پردازنده: دستورات وابسته به حالت را به درستی پردازش نمی کنند
- مشکل در تحلیل دستورات FPU: پیچیدگی محاسبات ممیز شناور باعث خطا می شود

۴.۳.۱ جداول مقایسه ای

جدول مقایسه انواع دستورات غیرمعمول

جدول ۱ انواع مختلف دستورات غیرمعمول و تأثیر آنها بر دیس اسمبلرها را مقایسه می کند.

جدول ۱.۱: مقایسه انواع دستورات غیرمعمول در Anti-Disassembly

نوع دستور	مثال ها	تأثیر بر دیس اسمبلر
دستورات شرطی پیچیده	JZ, JNZ, JC	شکستن تحلیل جریان کنترل
دستورات FPU	FCOMIP, FCHS, FLDZ	ایجاد خطا در تحلیل پشته FPU
دستورات سیستم	IN, OUT, SGDT	مشکل در شبیه سازی محیط
دستورات رشته ای	REP MOVSB, SCASB	پیچیدگی در تحلیل حلقه ها
دستورات حالت وابسته	ENTER, BOUND	وابستگی به حالت اجرا
دستورات محاسباتی پیچیده	ROR, ROL, BTC	مبهم کردن محاسبات

جدول مقایسه میزان اثرگذاری

جدول ۲ میزان اثرگذاری تکنیک های مختلف مبتنی بر دستورات غیرمعمول را نشان می دهد.

جدول ۲.۱: مقایسه میزان اثرگذاری تکنیک های دستورات غیرمعمول

تکنیک	میزان پیچیدگی	تأثیر بر عملکرد	کارایی
دستورات شرطی پیچیده	متوسط	کم	بالا
دستورات FPU غیرمعمول	بالا	متوسط	بسیار بالا
دستورات سیستم	بسیار بالا	بالا	متوسط
دستورات رشته ای پیچیده	متوسط	کم	بالا
دستورات حالت وابسته	بالا	متوسط	بالا
ترکیب چندین تکنیک	بسیار بالا	بالا	بسیار بالا

۴.۱ پنهان سازی از طریق تجزیه و تحلیل پویا

پنهان سازی از طریق تجزیه و تحلیل پویا یکی از پیشرفته ترین و کارآمدترین شیوه های مقابله با مهندسی معکوس است؛ زیرا بخش هایی از منطق برنامه تنها در زمان اجرا و تحت شرایط مشخص قابل مشاهده یا تحلیل هستند. این ویژگی باعث می شود ابزارهای تحلیل استاتیک نتوانند ساختار واقعی برنامه را بازسازی کنند و تحلیل گر مجبور به استفاده از روش های پویا گردد [۱].

یکی از مهم ترین تکنیک ها در این حوزه، رمزگشایی زمان اجرا (Runtime Decryption) است. در این روش بخش های حساس برنامه به صورت رمزگذاری شده ذخیره شده و تنها هنگام نیاز رمزگشایی می شوند. پس از اجرای تابع نیز این بخش ها مجدداً پاک شده یا دوباره رمزگذاری می شوند، که این موضوع دسترسی تحلیل گر به نسخه کامل و پایدار کد را دشوار می کند [۲]. این روش در بدافزارهای پیچیده و نرم افزارهای تجاری با حساسیت بالا بسیار رایج است.

روش دیگری که در پنهان سازی پویا کاربرد گسترده دارد، بارگذاری پویا (Dynamic Code Loading) است. در این تکنیک، بخش هایی از برنامه در فایل اجرایی وجود ندارند و در زمان اجرا از حافظه رمزگذاری شده، منابع بیرونی یا حتی شبکه بارگذاری می شوند. این ویژگی باعث بی اثر شدن بخش قابل توجهی از تحلیل های استاتیک می شود و تحلیل گر را مجبور به تحلیل پویا و ثبت رفتار اجرای زنده می کند [۳].

از سوی دیگر، تکنیک های تغییر مسیر کنترل در زمان اجرا (Dynamic Control Flow Modification) نیز با ایجاد وابستگی مسیر اجرای برنامه به شرایط محیطی مانند مقدار رجیسترها، وجود دیباگر، زمان بندی اجرای دستورها یا وضعیت سخت افزار، فرآیند تحلیل را برای مهاجم پیچیده می کنند. این وابستگی باعث می شود مسیر واقعی اجرا تنها در بستر واقعی قابل مشاهده باشد و ابزارهای تحلیلی قادر به بازسازی آن نباشند [۱].

بخش مهمی از پنهان سازی پویا به کارگیری تکنیک های ضد دیباگ پویا (Dynamic Anti-Debugging) است. روش هایی مانند اندازه گیری تأخیر اجرای دستورها، بررسی Breakpoint ها، شناسایی Emulator، یا فراخوانی توابع خاص سیستم عامل جهت تشخیص شرایط غیرعادی اجرا-همگی برای جلوگیری از تحلیل کد توسط ابزارهای دیباگ استفاده می شوند. در صورت تشخیص چنین شرایطی، برنامه می تواند منطق اصلی خود را پنهان کرده یا مسیر اجرا را تغییر دهد و تحلیل گر را گمراه کند [۴].

با وجود پتانسیل بالا، پنهان سازی پویا محدودیت هایی نیز دارد. ابزارهای تحلیل پیشرفته با استفاده از تکنیک هایی مانند حافظه برداری (Memory Dumping) و ابزارهای درج پویا (Dynamic Instrumentation) می توانند نسخه هایی از کد رمزگشایی شده در حافظه را استخراج کنند [۵]. با این حال، بهره گیری همزمان از چندین روش پویا و ایستا همچنان یکی از مقاوم ترین راهبردهای مقابله با مهندسی معکوس محسوب می شود.

۵.۱ معرفی کدهای خودتغییر (Self-Modifying Code)

کدهای خودتغییر به دسته ای از برنامه ها اشاره دارند که در زمان اجرا ساختار اجرایی خود را بازنویسی می کنند. در معماری رایج سامانه های امروزی-که دستور و داده هر دو در یک فضای حافظه ذخیره می شوند-برنامه می تواند همانند داده ها به ناحیه کد دسترسی پیدا کند و بخش هایی از آن را تغییر دهد. این ویژگی امکان آن را فراهم می کند که نسخه واقعی اجرا شده با نسخه ای که تحلیل گر در فایل دودویی مشاهده می کند یکسان نباشد. همین فاصله میان «کد ثابت» و «کد اجرایی» یکی از مهم ترین چالش ها در حوزه مهندسی معکوس و تحلیل استاتیک به شمار می رود [۶].

در این الگو، برنامه هنگام اجرا مجموعه ای از نوشتن ها (self-writes) بر روی ناحیه کد انجام می دهد. این نوشتن ها می توانند به صورت ساده تنها یک دستور را با نسخه معادل اما متفاوت جایگزین کنند، یا در قالب الگوهای پیشرفته تر، مسیرهای کنترلی و بلاک های اجرایی جدیدی را ایجاد کنند که فقط در حالت خاصی از اجرای برنامه فعال می شوند. به همین دلیل، کد خودتغییر الزاماً به معنای تغییر «منطق برنامه» نیست؛ بلکه بیشتر ابزاری برای پنهان سازی ساختار واقعی کد و پیچیده کردن تحلیل

است. رفتار نهایی برنامه ثابت می ماند، اما شکل داخلی آن در هر اجرا می تواند متفاوت باشد [۶].

از دید فنی، مهم ترین پیامد وجود SMC این است که بسیاری از دیس اسمبلرها و ابزارهای تحلیل استاتیک بر فرض «ثابت بودن کد» تکیه دارند. هنگامی که این فرض نقض می شود، ابزار قادر نخواهد بود ساختار واقعی جریان کنترل را به درستی بازبایی کند. مطالعات موجود نشان می دهد که این تغییرپذیری باعث می شود نمایش های معمولی مانند Control Flow Graph نتوانند وضعیت واقعی برنامه را پوشش دهند، زیرا هر بازنویسی در زمان اجرا می تواند نسخه جدیدی از یک بلاک اجرایی ایجاد کند و مسیرهای کنترلی متفاوتی را شکل دهد [۶].

بنابراین، Self-Modifying Code نه تنها یک تکنیک مخفی سازی است، بلکه مسئله ای بنیادین برای تحلیل گر ایجاد می کند: کدی که باید تحلیل شود، در فایل حضور ندارد و تنها در زمان اجرا به وجود می آید. این ویژگی، SMC را به یکی از پرکاربردترین سازوکارهای ضد دیس اسمبلی در بدافزارها، محافظ های نرم افزاری و ابزارهای ضد مهندسی معکوس تبدیل کرده است [۶].

۱.۵.۱ مثال ساده از کد خودتغییر (Self-Modifying Code)

فرض کن برنامه زیر قرار است در نهایت مقدار 10 را در رجیستر EAX قرار دهد، اما نویسنده برنامه می خواهد مسیر اصلی کد مخفی بماند؛ بنابراین از SMC استفاده می کند.

مرحله ۱: کد اولیه (نسخه قابل مشاهده در فایل)

در فایل دودویی (روی دیسک)، کد به این شکل دیده می شود:

```
MOV EAX, 5
MOV EBX, 5
ADD EAX, EBX      ;      5 + 5
JMP END
```

برای تحلیلگر، این یک کد کاملاً معمولی است.

مرحله ۲: بخش خودتغییر (self-write)

اما بخشی از برنامه پیش از اجرای دستور ADD این تغییر را اعمال می کند:

```
MOV BYTE PTR [address_of_ADD], 0x90
MOV BYTE PTR [address_of_ADD+1], 0x90
MOV BYTE PTR [address_of_ADD+2], 0x90
```

دستور ۰x۹۰ برابر NOP است (هیچ کاری انجام نمی دهد). یعنی برنامه قبل از اجرا دستور اصلی را پاک کرده است.

مرحله ۳: نسخه واقعی که اجرا می شود

وقتی CPU به خط مربوط به ADD می رسد، دیگر این دستور وجود ندارد. نسخه دیده شده توسط تحلیلگر:

```
ADD EAX, EBX
```

اما نسخه واقعی در حافظه تبدیل شده به:

```
NOP
NOP
NOP
```

مقدار نهایی:

```
EAX = 5
```

که برابر ۱۰ نیست.

۲.۵.۱ معایب مثال ساده SMC

مثال ساده ای که در آن یک دستور ADD در زمان اجرا با چند دستور NOP جایگزین می شود، اگرچه برای معرفی مفهوم Self-Modifying Code مفید است، اما از نظر فنی و تحلیلی محدودیت های مهمی دارد... [۶]

۳.۵.۱ مثال علمی از SMC وابسته به زمان اجرا (Time-Dependent Self-Modifying Code)

در این مثال، ساختار کد به گونه ای طراحی شده است که **دستور واقعی** در زمان اجرا و بر اساس مقدار زمان تولید می شود [۶].

مرحله ۱: کد روی دیسک

```
MOV EAX, [SystemTime]
```

```
AND EAX, 1
```

```
JZ EvenPath
```

```
JMP OddPath
```

EvenPath:

```
; encrypted even block
```

```
DB 0xA1, 0x29, 0xF3, 0xC2, 0x18
```

OddPath:

```
; encrypted odd block
```

```
DB 0x9B, 0x11, 0x84, 0xCC, 0x72
```

نسخه های رمز شده اند .

مرحله ۲: انتخاب مسیر

در زمان اجرا مسیر بر اساس ثانیه انتخاب می شود.

مرحله ۳: تولید کد پویا

DecryptStage:

```
MOV ESI, SelectedEncryptedBlock
```

```
MOV EDI, ExecBuffer
```

```
MOV ECX, 5
```

```
DecryptLoop:
```

```
XOR BYTE PTR [EDI], BYTE PTR [ESI]
```

```
INC ESI
```

```
INC EDI
```

```
LOOP DecryptLoop
```

```
JMP ExecBuffer
```

مرحله ۴: اجرای نسخه تولیدشده

```
MOV EAX, 100
```

```
ADD EAX, 20
```

```
RET
```

۶.۱ مقاومت در برابر دیس اسمبلی با استفاده از تکنیک های رمزگذاری

با تبدیل فایل اجرایی به کد اسمبلی، تحلیل گران می توانند منطق داخلی، باگ ها و بدافزارهای احتمالی را شناسایی کنند [۷]. به عبارت دیگر، دیس اسمبلی امکان افشای عملکرد برنامه و پیدا کردن آسیب پذیری ها یا کدهای مخرب را فراهم می کند. به همین دلیل، نویسندگان نرم افزارهای محافظت شده تلاش می کنند با روش هایی مانند رمزگذاری و ابهام زایی، دیس اسمبلی را دشوار کنند.

۱.۶.۱ کد پلی مورفیک (Polymorphic Code)

تعریف و مکانیسم:

کد پلی مورفیک نوعی کد خودتغییرده است که در هر نسخه ظاهر باینری متفاوتی دارد، ولی منطق و الگوریتم اصلی اش حفظ می شود [۸]. معمولاً در این روش، بخش اصلی کد («payload») با یک کلید

تصادفی رمزگذاری می شود و یک بخش کوتاه کوچک در ابتدای برنامه (بخش رمزگشا) وجود دارد که با کلید مربوطه، payload را در حافظه رمزگشایی می کند [۸، ۹]. به عنوان مثال، ویروس های پلی مورفیک مشهور، از رمزنگاری XOR با یک کلید تصادفی برای مخفی کردن بدنه اصلی خود استفاده می کنند، به طوری که هر نمونه از لحاظ باینری کاملاً با نمونه های دیگر متفاوت است [۹].

مزایا و کاربردها:

مزیت اصلی پلی مورفیسم مقاومت در برابر شناسایی مبتنی بر امضا است؛ چون هر بار که کد اجرا می شود، الگویی یکتا ایجاد می کند و امضای ثابتی ندارد [۹]. این روش عمدتاً در ویروس ها، کرم ها و بدافزارها به کار می رود تا برنامه های ضد ویروس را فریب دهد. همچنین در برخی محافظت کننده های تجاری کد (پکرها) دیده می شود که با تولید کلیدهای مختلف یا تغییر الگوریتم رمزگشایی، کد را در هر توزیع متفاوت می کنند. پلی مورفیسم نسبتاً ساده است و تأثیر زیادی بر عملکرد اجرایی برنامه ندارد.

نحوه عملکرد:

معمولاً هر نمونه پلی مورفیک شامل دو بخش است: یک لایه رمزنگاری (برای پنهان سازی payload) و یک بخش رمزگشای کوتاه. در زمان اجرا، ابتدا بخش رمزگشا فعال شده و بدنه اصلی را در حافظه بازمی کند، سپس کنترل به کد اصلی داده می شود [۸]. به خاطر این ساختار، دیس اسمبلی ایستا مؤثر نیست زیرا کد اصلی در فایل رمز یافته و فقط با اجرای برنامه آشکار می شود [۸].

معایب و روش های تحلیل:

گرچه پلی مورفیسم شناسایی ساده مبتنی بر امضا را دشوار می کند، اما شکستن آن آسان است. تحلیل گران می توانند برنامه را اجرا کنند و با قرار دادن نقطه نگاری روی بخش رمزگشا یا استفاده از دیباگر، بدنه اصلی را در زمان اجرای برنامه مشاهده و استخراج نمایند [۸]. افزون بر این، چون الگوریتم رمزگشایی معمولاً ثابت یا ساده است، می توان با نوشتن اسکریپت های کوچک یا استفاده از ابزارهای خودکار، کد را رمزگشایی کرد. به همین علت، اکثر روش های مقابله با کد پلی مورفیک بر تحلیل پویا (اجرا در محیط کنترل شده و خاموش کردن رمزنگاری) و یا نرمال سازی کد بر مبنای الگوریتم های آماری متمرکزند.

۲.۶.۱ کد متامورفیک (Metamorphic Code)

تعریف و مکانیسم:

کد پلی مورفیک نوعی کد خودتغییرده است که در هر نسخه ظاهر باینری متفاوتی دارد، ولی منطق و الگوریتم اصلی اش حفظ می شود [۸]. معمولاً در این روش، بخش اصلی کد («payload») با یک کلید

تصادفی رمزگذاری می شود و یک بخش کوتاه کوچک در ابتدای برنامه (بخش رمزگشا) وجود دارد که با کلید مربوطه، payload را در حافظه رمزگشایی می کند [۹][۸]. به عنوان مثال، ویروس های پلی مورفیک مشهور، از رمزنگاری XOR با یک کلید تصادفی برای مخفی کردن بدنه ی اصلی خود استفاده می کنند، به طوری که هر نمونه از لحاظ باینری کاملاً با نمونه های دیگر متفاوت است [۹].

مزایا و کاربردها:

مزیت اصلی پلی مورفیسم مقاومت در برابر شناسایی مبتنی بر امضاست؛ چون هر بار که کد اجرا می شود، الگویی یکتا ایجاد می کند و امضای ثابتی ندارد [۹]. این روش عمدتاً در ویروس ها، کرم ها و بدافزارها به کار می رود تا برنامه های ضد ویروس را فریب دهد. همچنین در برخی محافظت کننده های تجاری کد (پکرها) دیده می شود که با تولید کلیدهای مختلف یا تغییر الگوریتم رمزگشایی، کد را در هر توزیع متفاوت می کنند. پلی مورفیسم نسبتاً ساده است و تأثیر زیادی بر عملکرد اجرایی برنامه ندارد.

معایب و روش های تحلیل:

گرچه پلی مورفیسم شناسایی ساده مبتنی بر امضا را دشوار می کند، اما شکستن آن آسان است. تحلیل گران می توانند برنامه را اجرا کنند و با قرار دادن نقطه نگاری روی بخش رمزگشا یا استفاده از دیباگر، بدنه اصلی را در زمان اجرای برنامه مشاهده و استخراج نمایند [۸]. افزون بر این، چون الگوریتم رمزگشایی معمولاً ثابت یا ساده است، می توان با نوشتن اسکریپت های کوچک یا استفاده از ابزارهای خودکار، کد را رمزگشایی کرد. به همین علت، اکثر روش های مقابله با کد پلی مورفیک بر تحلیل پویا (اجرا در محیط کنترل شده و خاموش کردن رمزنگاری) و یا نرمال سازی کد بر مبنای الگوریتم های آماری متمرکزند.

۳.۶.۱ بسته سازی کد (Packers)

تعریف و مکانیسم:

کد پلی مورفیک نوعی کد خودتغییرده است که در هر نسخه ظاهر باینری متفاوتی دارد، ولی منطق و الگوریتم اصلی اش حفظ می شود [۸]. معمولاً در این روش، بخش اصلی کد («payload») با یک کلید تصادفی رمزگذاری می شود و یک بخش کوتاه کوچک در ابتدای برنامه (بخش رمزگشا) وجود دارد که با کلید مربوطه، payload را در حافظه رمزگشایی می کند [۹][۸]. به عنوان مثال، ویروس های پلی مورفیک مشهور، از رمزنگاری XOR با یک کلید تصادفی برای مخفی کردن بدنه ی اصلی خود استفاده می کنند، به طوری که هر نمونه از لحاظ باینری کاملاً با نمونه های دیگر متفاوت است [۹].

مزایا و کاربردها:

مزیت اصلی پلی مورفیسم مقاومت در برابر شناسایی مبتنی بر امضاست؛ چون هر بار که کد اجرا می شود، الگویی یکتا ایجاد می کند و امضای ثابتی ندارد [۹]. این روش عمدتاً در ویروس ها، کرم ها و بدافزارها به کار می رود تا برنامه های ضد ویروس را فریب دهد. همچنین در برخی محافظت کننده های تجاری کد (پکرها) دیده می شود که با تولید کلیدهای مختلف یا تغییر الگوریتم رمزگشایی، کد را در هر توزیع متفاوت می کنند. پلی مورفیسم نسبتاً ساده است و تأثیر زیادی بر عملکرد اجرایی برنامه ندارد.

معایب و روش های تحلیل:

گرچه پلی مورفیسم شناسایی ساده مبتنی بر امضا را دشوار می کند، اما شکستن آن آسان است. تحلیل گران می توانند برنامه را اجرا کنند و با قرار دادن نقطه نگاری روی بخش رمزگشا یا استفاده از دیباگر، بدنه اصلی را در زمان اجرای برنامه مشاهده و استخراج نمایند [۸]. افزون بر این، چون الگوریتم رمزگشایی معمولاً ثابت یا ساده است، می توان با نوشتن اسکریپت های کوچک یا استفاده از ابزارهای خودکار، کد را رمزگشایی کرد. به همین علت، اکثر روش های مقابله با کد پلی مورفیک بر تحلیل پویا (اجرا در محیط کنترل شده و خاموش کردن رمزنگاری) و یا نرمال سازی کد بر مبنای الگوریتم های آماری متمرکزند.

۴.۶.۱ ابهام زایی مبتنی بر مجازی سازی (Virtualization-Based Obfuscation)

تعریف و مکانیسم:

کد پلی مورفیک نوعی کد خودتغییرده است که در هر نسخه ظاهر باینری متفاوتی دارد، ولی منطق و الگوریتم اصلی اش حفظ می شود [۸]. معمولاً در این روش، بخش اصلی کد («payload») با یک کلید تصادفی رمزگذاری می شود و یک بخش کوتاه کوچک در ابتدای برنامه (بخش رمزگشا) وجود دارد که با کلید مربوطه، payload را در حافظه رمزگشایی می کند [۹][۸]. به عنوان مثال، ویروس های پلی مورفیک مشهور، از رمزنگاری XOR با یک کلید تصادفی برای مخفی کردن بدنه اصلی خود استفاده می کنند، به طوری که هر نمونه از لحاظ باینری کاملاً با نمونه های دیگر متفاوت است [۹].

مزایا و کاربردها:

مزیت اصلی پلی مورفیسم مقاومت در برابر شناسایی مبتنی بر امضاست؛ چون هر بار که کد اجرا می شود، الگویی یکتا ایجاد می کند و امضای ثابتی ندارد [۹]. این روش عمدتاً در ویروس ها، کرم ها و بدافزارها به کار می رود تا برنامه های ضد ویروس را فریب دهد. همچنین در برخی محافظت کننده های تجاری کد (پکرها) دیده می شود که با تولید کلیدهای مختلف یا تغییر الگوریتم رمزگشایی، کد را در هر توزیع متفاوت می کنند. پلی مورفیسم نسبتاً ساده است و تأثیر زیادی بر عملکرد اجرایی برنامه ندارد.

معایب و روش های تحلیل:

گرچه پلی مورفیسم شناسایی ساده مبتنی بر امضا را دشوار می کند، اما شکستن آن آسان است. تحلیل گران می توانند برنامه را اجرا کنند و با قرار دادن نقطه نگاری روی بخش رمزگشا یا استفاده از دیباگر، بدنه اصلی را در زمان اجرای برنامه مشاهده و استخراج نمایند [۸]. افزون بر این، چون الگوریتم رمزگشایی معمولاً ثابت یا ساده است، می توان با نوشتن اسکریپت های کوچک یا استفاده از ابزارهای خودکار، کد را رمزگشایی کرد. به همین علت، اکثر روش های مقابله با کد پلی مورفیک بر تحلیل پویا (اجرا در محیط کنترل شده و خاموش کردن رمزنگاری) و یا نرمال سازی کد بر مبنای الگوریتم های آماری متمرکزند.

۵.۶.۱ رمزگذاری کد در زمان اجرا (Runtime Code Encryption)

تعریف و مکانیسم:

کد پلی مورفیک نوعی کد خودتغییرده است که در هر نسخه ظاهر باینری متفاوتی دارد، ولی منطق و الگوریتم اصلی اش حفظ می شود [۸]. معمولاً در این روش، بخش اصلی کد («payload») با یک کلید تصادفی رمزگذاری می شود و یک بخش کوتاه کوچک در ابتدای برنامه (بخش رمزگشا) وجود دارد که با کلید مربوطه، payload را در حافظه رمزگشایی می کند [۸، ۹]. به عنوان مثال، ویروس های پلی مورفیک مشهور، از رمزنگاری XOR با یک کلید تصادفی برای مخفی کردن بدنه ی اصلی خود استفاده می کنند، به طوری که هر نمونه از لحاظ باینری کاملاً با نمونه های دیگر متفاوت است [۹].

مزایا و کاربردها:

مزیت اصلی پلی مورفیسم مقاومت در برابر شناسایی مبتنی بر امضا است؛ چون هر بار که کد اجرا می شود، الگویی یکتا ایجاد می کند و امضای ثابتی ندارد [۹]. این روش عمدتاً در ویروس ها، کرم ها و بدافزارها به کار می رود تا برنامه های ضد ویروس را فریب دهد. همچنین در برخی محافظت کننده های تجاری کد (پکرها) دیده می شود که با تولید کلیدهای مختلف یا تغییر الگوریتم رمزگشایی، کد را در هر توزیع متفاوت می کنند. پلی مورفیسم نسبتاً ساده است و تأثیر زیادی بر عملکرد اجرایی برنامه ندارد.

معایب و روش های تحلیل:

گرچه پلی مورفیسم شناسایی ساده مبتنی بر امضا را دشوار می کند، اما شکستن آن آسان است. تحلیل گران می توانند برنامه را اجرا کنند و با قرار دادن نقطه نگاری روی بخش رمزگشا یا استفاده از دیباگر، بدنه اصلی را در زمان اجرای برنامه مشاهده و استخراج نمایند [۸]. افزون بر این، چون الگوریتم رمزگشایی معمولاً ثابت یا ساده است، می توان با نوشتن اسکریپت های کوچک یا استفاده از ابزارهای

خودکار، کد را رمزگشایی کرد. به همین علت، اکثر روش های مقابله با کد پلی مورفیک بر تحلیل پویا (اجرا در محیط کنترل شده و خاموش کردن رمزنگاری) و یا نرمال سازی کد بر مبنای الگوریتم های آماری متمرکزند.

فصل ۲

مشکلات مطرح در چرخه‌های توسعه و تکامل نرم‌افزار

۱.۲ روش‌های Anti Anti Disassembly

۱.۱.۲ مقدمه ای بر ضد ضد دی‌اسمبل (Anti-Anti-Disassembly)

در دنیای مدرن نرم‌افزار، کد منبع (Source Code) هسته اصلی مالکیت فکری و ارزش تجاری هر محصولی است. با افزایش پیچیدگی سیستم‌ها و گسترش تهدیدات امنیتی، نیاز به جلوگیری از مهندسی معکوس (Reverse Engineering)، سرقت فکری، و تحلیل بدافزارها اهمیت حیاتی یافته است. این نیاز، توسعه‌دهندگان نرم‌افزار را به سمت استفاده از تکنیک‌هایی تحت عنوان اُبفوسکیشن (Obfuscation) و ضد دی‌اسمبل (Anti-Disassembly - AD) سوق داده است.

۲.۱.۲ تعریف (AD) Anti-Disassembly:

Anti-Disassembly مجموعه‌ای از ترفندها و الگوهای برنامه‌نویسی است که عمدهً برای گمراه کردن ابزارهای تحلیل استاتیک (مانند دی‌اسمبلرها و دکامپایلرها) طراحی شده‌اند. این تکنیک‌ها با هدف شکستن نمودار جریان کنترل (Control Flow Graph - CFG) و جلوگیری از تفسیر صحیح دستورات ماشین توسط ابزارهای خودکار به کار می‌روند. روش‌هایی مانند قرار دادن بایت‌های فیک، پرش‌های نامعتبر، یا پرش‌های داینامیک محاسبه‌شده، تحلیل کد را برای انسان بسیار دشوار و زمان‌بر می‌سازند.

۳.۱.۲ ورود به Anti-Anti-Disassembly (AADA)

ضد ضد دی‌اسمبل (Anti-Anti-Disassembly - AADA)، در پاسخ به تکنیک‌های AD، به وجود آمده است. AADA نه تنها یک ابزار، بلکه یک فلسفه مقابله‌ای در مهندسی معکوس است. این حوزه به مجموعه‌ای از روش‌های پیشرفته و هوشمندانه اشاره دارد که توسط متخصصان امنیت و تحلیلگران بدافزار برای شناسایی، تجزیه و خنثی کردن عمدی لایه‌های محافظتی و مبهم‌سازی کد ایجاد شده‌اند. هدف غایی AADA این است که به‌طور موفقیت‌آمیز شفافیت (Clarity) و دقت (Accuracy) تحلیل را به کد بازگرداند. این فرآیند اغلب شامل تحلیل دینامیک (اجرای کنترل‌شده کد)، اجرای نمادین (Symbolic Execution) و اصلاح خودکار فایل‌های باینری (Patching) است تا موانع AD برداشته شده و تحلیلگر بتواند به منطق اصلی برنامه دست یابد.

۴.۱.۲ اهمیت در تکامل نرم‌افزار

در چارچوب درس تکامل نرم‌افزار (Software Evolution)، AADA نقش حیاتی در فرآیندهای نگهداری پیشگیرانه (Preventive Maintenance) و ممیزی امنیتی (Security Auditing) ایفا می‌کند. دانش AADA به تیم‌های توسعه و امنیت این امکان را می‌دهد که:

- بدافزارها را تحلیل کنند: برای درک نحوه عملکرد کدهای مخرب که از AD برای پنهان شدن استفاده می‌کنند.
- اثربخشی محافظت را ارزیابی کنند: برای تست و ارزیابی مقاومت تکنیک‌های حفاظت از کد خود در برابر پیشرفته‌ترین روش‌های شکستن آن.

۵.۱.۲ تکنیک‌های کلیدی ضد ضد دی‌اسمبل (AADA)

تکنیک‌های AADA به‌طور مستقیم هدف‌گذاری شده‌اند تا موانعی را که روش‌های Anti-Disassembly (AD) برای گمراه کردن تحلیل استاتیک ایجاد کرده‌اند، از بین ببرند. موفقیت AADA به توانایی تحلیلگر در ترکیب ابزارهای استاتیک و دینامیک بستگی دارد.

۱. تحلیل دینامیک و رصد (Dynamic Analysis and Tracing)

بسیاری از تکنیک‌های AD به متغیرهای وابسته به زمان اجرا (Runtime) تکیه دارند (مانند پرش‌های محاسبه‌شده). تحلیل دینامیک مؤثرترین روش برای غلبه بر این موانع است.

- استفاده از اشکال‌زدها (Debuggers) اجرای کد در یک محیط کنترل‌شده (Sandbox) یا یک اشکال‌زدای سطح پایین مانند x64dbg یا GDB به تحلیلگر اجازه می‌دهد تا مقادیر رجیسترها و حافظه را در لحظه اجرای دستورات حیاتی مشاهده کند.
 - ردیابی دستورات (Instruction Tracing) در این روش توالی دقیق دستوراتی که در طول اجرای بخش مبهم‌سازی شده اجرا می‌شوند ثبت می‌گردد. این ردیابی می‌تواند آدرس‌های پرش‌های دینامیک را آشکار سازد و مسیر واقعی جریان کنترل را نشان دهد.
 - استخراج آدرس‌های مقصد (Destination Address Extraction) در مورد پرش‌های محاسباتی مانند JMP [Register] یا CALL [Register]، تحلیل دینامیک مقدار نهایی رجیستر را در لحظه پرش کشف می‌کند و آدرس مقصد واقعی را به صورت صریح به دست می‌دهد.
۲. وصله زدن و اصلاح کد (Patching and Code Modification) هنگامی که مقصد واقعی جریان کنترل کشف می‌شود، تحلیلگر اغلب بایت‌های اصلی فایل باینری را برای تسهیل تحلیل‌های بعدی اصلاح می‌کند.
- NOPing دستورات فیک دستوراتی که توسط AD برای خراب کردن دی‌اسمبلر استفاده شده‌اند، با دستور NOP (No Operation) جایگزین می‌شوند. این کار دستورات را غیرفعال می‌کند اما اندازه کد را ثابت نگه می‌دارد و دی‌اسمبلر می‌تواند به درستی از دستورات فیک عبور کند.
 - تبدیل پرش دینامیک به پرش استاتیک پس از کشف آدرس مقصد پرش دینامیک، تحلیلگر می‌تواند دستورات پیچیده را با یک دستور پرش استاتیک و مستقیم (JMP یا CALL) به آدرس کشف شده جایگزین کند. این کار به ابزارهای تحلیل استاتیک اجازه می‌دهد تا نمودار جریان کنترل (CFG) را به درستی بازسازی کنند.
 - خنثی‌سازی تکنیک‌های ضد دی‌باگ حذف دستوراتی که عمداً برای شناسایی اشکال‌زدا و توقف برنامه (Anti-Debugging) طراحی شده‌اند.
۳. اجرای نمادین (Symbolic Execution) اجرای نمادین یک تکنیک پیشرفته است که به جای مقادیر واقعی ورودی، از نمادها استفاده می‌کند. این روش برای شکستن ابفوسکیشن‌های سنگین که شامل عملیات ریاضی پیچیده هستند، بسیار قدرتمند است.
- تحلیل نمادین پرش‌ها در سناریوهایی که آدرس پرش با عملیات ریاضی پیچیده‌ای بر روی متغیرهای داخلی محاسبه می‌شود، اجرای نمادین می‌تواند فرمول ریاضی محاسبه آدرس را به دست آورد و تمام مقادیر ممکن را تعیین کند.

- ابزارهای پیشرفته ابزاریهایی مانند Angr و Triton از اجرای نمادین برای کاوش خودکار در مسیرهای اجرایی کد مبهم‌سازی شده و کشف مقاصد پنهان استفاده می‌کنند.
- ۴. اسکریپت‌نویسی خودکار (Automated Scripting) برای مقابله با حجم بالای اُبفوسکیشن و تکنیک‌های تکراری AD، تحلیلگران از قابلیت اسکریپت‌نویسی دی‌اسمبلرها استفاده می‌کنند.
- اسکریپت‌های تخصصی نوشتن اسکریپت‌هایی (مانند IDAPython) که یک الگوی خاص AD را در کد شناسایی می‌کنند و به طور خودکار آن را خنثی می‌کنند (به عنوان مثال، اسکریپتی که به دنبال توالی بایت‌های خاص برای یک دستور نامعتبر می‌گردد و بلافاصله آن را NOP می‌کند).
- بازسازی CFG این اسکریپت‌ها اطلاعاتی را که از تحلیل دینامیک به دست آمده‌اند، در تحلیل استاتیک دی‌اسمبلر وارد می‌کنند تا نمودار جریان کنترل مجدداً با دقت بالا ترسیم شود.

۲.۲ استفاده از دیس اسمبلرهای پیشرفته

۱.۲.۲ دلیل نام‌گذاری پیشرفته برای این نوع اسمبلرها

دی‌اسمبلرهای مدرن (مانند IDA Pro و Ghidra) فراتر از یک ابزار ساده برای ترجمه کد ماشین به اسمبلی عمل می‌کنند و به همین دلیل عنوان "پیشرفته" را به خود اختصاص داده‌اند. این برتری به دلیل سه قابلیت کلیدی زیر است:

۱. تحلیل هوشمند و سطح بالا (High-Level Analysis)

- **دکامپایلر داخلی** مهم‌ترین تفاوت، وجود دکامپایلر است. این قابلیت به ابزار اجازه می‌دهد کد اسمبلی سطح پایین را به یک شبه کد سطح بالا (شبه زبان C/C++) تبدیل کند، که درک منطق برنامه را ده‌ها برابر سریع‌تر و آسان‌تر می‌کند.
- **ترسیم CFG** این ابزارها قادر به ترسیم دقیق نمودار جریان کنترل (Control Flow Graph) هستند.

۲. قابلیت مقابله با موانع امنیتی (AADA Integration)

- **اسکریپت‌نویسی قوی** دی‌اسمبلرهای پیشرفته دارای API برنامه‌نویسی (مانند IDAPython) هستند که به تحلیلگر اجازه می‌دهد کارهای تکراری را خودکار کند.
- **خنثی‌سازی** (AADA) Anti-Disassembly اسکریپت‌ها برای شناسایی الگوهای مبهم‌سازی (Obfuscation) و اعمال تصحیحات خودکار (مانند Patching) در کد استفاده می‌شوند تا موانع امنیتی را خنثی کنند.

۳. یکپارچگی و انعطاف‌پذیری

- **پشتیبانی چند معماری** آن‌ها از ده‌ها معماری پردازنده (ARM، MIPS، x86/x64) پشتیبانی می‌کنند، برخلاف ابزارهای قدیمی که محدود بودند.
- **مدیریت پروژه** تمامی نتایج تحلیل، نام‌گذاری‌ها، و تصحیحات انجام شده در یک پایگاه داده پروژه ذخیره می‌شود و نیازی به تحلیل مجدد در هر بار بازگشایی نیست.

به طور خلاصه، دی‌اسمبلرهای پیشرفته به دلیل تبدیل شدن به محیط‌های هوشمند، خودکار و یکپارچه برای تحلیل عمیق کد، و توانایی آن‌ها در غلبه بر دفاعیات Anti-Disassembly، ابزارهای قدیمی را پشت سر گذاشته‌اند.

۳.۲ استفاده از تحلیل رفتار اجرایی

تحلیل رفتار اجرایی، که در ادبیات امنیت سایبری به عنوان تحلیل پویا (Dynamic Analysis) نیز شناخته می‌شود، رویکردی است که بر مشاهده عملکرد برنامه در حین اجرا تمرکز دارد. این روش به عنوان پاسخی مستقیم به محدودیت‌های تحلیل ایستا (Static Analysis) توسعه یافته است. در حالی که تکنیک‌های ضد دیس‌اسمبلی با دستکاری جریان کنترل (Control Flow) و تزریق دستورات انحرافی سعی در گمراه کردن ابزارهای دیس‌اسمبلر و تحلیل‌گرانی دارند که کد را روی دیسک بررسی می‌کنند، تحلیل رفتاری با نادیده گرفتن ساختار داخلی کد و تمرکز بر «نیت» و «اثر» برنامه، این موانع را دور می‌زند. [۱۰] [۱۱]

۱.۳.۲ معماری آزمایشگاه و متدولوژی پایش

برای انجام یک تحلیل رفتاری دقیق که بتواند لایه‌های ضد تحلیل را خنثی کند، محیط آزمایشگاه باید به گونه‌ای پیکربندی شود که تمامی آثار دیجیتال (Digital Footprints) ثبت گردند و در عین حال، بدافزار

قادر به تشخیص محیط مجازی نباشد. استفاده از ماشین‌های مجازی ایزوله مانند Flare VM (مبتنی بر ویندوز) و Remnux (مبتنی بر لینوکس) به عنوان استاندارد صنعتی پذیرفته شده است. [۱۲]

فرآیند تحلیل رفتاری معمولاً شامل مراحل زیر است که هر یک لایه‌ای از فعالیت‌های بدافزار را آشکار می‌سازد:

۱. **تصویربرداری از وضعیت سیستم (System State Snapshot):** قبل از اجرای نمونه مشکوک،

ضروری است که وضعیت پایه سیستم ثبت شود. ابزارهایی مانند Regshot یا Process Hacker برای گرفتن تصویر از وضعیت رجیستری و فایل‌سیستم استفاده می‌شوند. پس از اجرای بدافزار و اتمام دوره فعالیت آن، تصویر دوم گرفته شده و با تصویر اول مقایسه می‌شود. این مقایسه تغییرات پایداری (Persistence Mechanisms) مانند ایجاد کلیدهای رجیستری برای اجرای خودکار (Run Keys) یا تغییرات در تنظیمات امنیتی سیستم را آشکار می‌سازد. [۱۲]

۲. **پایش فعالیت‌های سیستمی (System Activity Monitoring):** در حین اجرای بدافزار، ابزار

Process Monitor (ProcMon) نقشی حیاتی ایفا می‌کند. این ابزار تمامی فعالیت‌های فایل‌سیستم، رجیستری و پروسه‌ها را در زمان واقعی ضبط می‌کند. تحلیل‌گران با استفاده از فیلترهای اختصاصی، نویزهای سیستم‌عامل را حذف کرده و تمرکز خود را بر روی پروسه مخرب قرار می‌دهند. برای مثال، مشاهده تلاش‌های مکرر برای دسترسی به فایل‌های خاص یا بارگذاری کتابخانه‌های دینامیک (DLLs) مشکوک می‌تواند نشان‌دهنده تلاش برای تزریق کد (Code Injection) یا سرقت اطلاعات باشد. [۱۲] [۱۳]

۳. **تحلیل ترافیک شبکه (Network Traffic Analysis):** بسیاری از بدافزارها برای دریافت دستورات

یا ارسال اطلاعات سرقت شده با سرورهای کنترل و فرمان (C2) ارتباط برقرار می‌کنند. برای تحلیل این ارتباطات بدون به خطر انداختن شبکه واقعی، از ابزارهایی مانند Wireshark برای شنود پکت‌ها و Fiddler برای تحلیل ترافیک HTTP/HTTPS استفاده می‌شود. در یک محیط آزمایشگاهی پیشرفته، ماشین Remnux به عنوان دروازه اینترنت (Gateway) برای ماشین Flare VM عمل کرده و با ابزارهایی مانند FakeDNS و Inetsim سرویس‌های اینترنتی را شبیه‌سازی می‌کند. این امر بدافزار را فریب می‌دهد تا درخواست‌های خود را ارسال کند، حتی اگر دسترسی واقعی به اینترنت وجود نداشته باشد. [۱۲] [۱۳]

۲.۳.۲ چالش‌های ضد دیباگ و راهکارهای عبور (Anti-Anti-Debugging)

یکی از چالش‌های اصلی در تحلیل رفتاری، مکانیزم‌های «ضد دیباگ» (Anti-Debugging) است. نویسندگان بدافزار کد خود را به گونه‌ای طراحی می‌کنند که محیط تحلیل را شناسایی کرده و در صورت تشخیص دیباگر یا ماشین مجازی، رفتار خود را تغییر داده یا اجرا را متوقف کنند. این تکنیک‌ها تحلیل‌گر را مجبور می‌کنند تا از روش‌های «ضد ضد دیباگ» استفاده کند. [۱۴] [۱۵]

تکنیک‌های شناسایی دیباگر

بدافزارها از روش‌های متنوعی برای کشف حضور دیباگر استفاده می‌کنند:

بررسی پرچم‌های سیستم استفاده از توابع API ویندوز مانند `IsDebuggerPresent` یا بررسی مستقیم ساختار `PEB (Process Environment Block)` برای یافتن پرچم‌های `BeingDebugged` و `NtGlobalFlag`. [۱۴] [۱۵]

بررسی‌های زمانی (Timing Checks) استفاده از دستوراتی مانند `RDTSC` برای اندازه‌گیری تعداد چرخه‌های پردازنده. از آنجا که اجرای کد در دیباگر (به خصوص در حالت `Single-step`) بسیار کندتر از اجرای عادی است، اختلاف زمانی زیاد نشان‌دهنده حضور تحلیل‌گر است. [۱۰] [۱۶]

تزریق کد و اسکن وقفه جستجو برای دستورات وقفه نرم‌افزاری (`INT 3`) که توسط دیباگرها برای ایجاد نقاط توقف (`Breakpoints`) در کد تزریق می‌شوند. [۱۶]

راهکارهای مقابله (Anti-Anti-Debugging)

در چارچوب «ضد ضد دیس‌اسمبلی»، تحلیل‌گران از استراتژی‌های زیر برای پنهان‌سازی ابزارهای خود و خنثی‌سازی تکنیک‌های شناسایی استفاده می‌کنند:

۱. **تحلیل مبتنی بر هایپروایزر (Hypervisor-based Analysis):** استفاده از ابزارهایی مانند `HyperDbg` یا تکنیک‌های درون‌نگری ماشین مجازی (`VMI`) که خارج از سیستم‌عامل مهمان اجرا می‌شوند، یکی از پیشرفته‌ترین روش‌هاست. در این معماری، ابزار تحلیل در سطح هایپروایزر (`Ring -1`) قرار دارد و سیستم‌عامل مهمان و بدافزار که در سطح کرنل (`Ring 0`) یا کاربر (`Ring 3`) اجرا می‌شوند، قادر به تشخیص یا دستکاری آن نیستند. این روش به تحلیل‌گر اجازه می‌دهد تا بدون تغییر در ساختارهای حافظه سیستم عامل مهمان (که بدافزار آن‌ها را بررسی می‌کند)، بر اجرا نظارت داشته باشد و شفافیت تحلیل را به حداکثر برساند. [۱۷]

۲. **پنهان‌سازی مصنوعات (Artifact Hiding):** برای مقابله با بدافزارهایی که محیط‌های مجازی مانند VMware یا VirtualBox را شناسایی می‌کنند، تحلیل‌گران باید مصنوعات این محیط‌ها را پنهان کنند. این شامل تغییر نام درایورهای دستگاه، اصلاح جداول ACPI، تغییر آدرس‌های MAC کارت شبکه و استفاده از پلاگین‌هایی مانند ScyllaHide است. پلاگین ScyllaHide می‌تواند توابع API مربوط به تشخیص دیباگر (مانند IsDebuggerPresent) را در حافظه هوک کرده و همواره مقدار "False" را برگرداند، بدین ترتیب بدافزار فریب خورده و به اجرای مخرب خود ادامه می‌دهد. [۱۵] [۱۷]

۳. **دستکاری محیط اجرا (Environment Tampering):** گاهی اوقات تحلیل‌گر ناچار است کد بدافزار را در حین اجرا وصله (Patch) کند. برای مثال، اگر بدافزار از دستور RDTSC برای بررسی زمان اجرا استفاده کند، تحلیل‌گر می‌تواند با تغییر درایورهای سیستم یا تنظیمات ماشین مجازی، کاری کند که این دستور همواره مقداری ثابت یا منطقی را برگرداند، یا به سادگی دستور شرطی که بعد از بررسی زمان قرار دارد را در حافظه به یک دستور پرش بی‌قید و شرط (JMP) تغییر دهد تا بررسی امنیتی دور زده شود. [۱۶]

۴.۲ تکنیک‌های رمزگشایی و تحلیل کدهای رمزگذاری شده (Decryp- tion Techniques)

بخش اعظمی از بدافزارهای مدرن برای جلوگیری از تحلیل ایستا و دیس‌اسمبلی، از «پکرها» (Packers) و «کریپترها» (Crypters) استفاده می‌کنند. در این حالت، فایل اجرایی روی دیسک ترکیبی از داده‌های رمزگذاری شده و یک قطعه کد کوچک برای بازگردانی (Stub) است. هدف از تکنیک‌های رمزگشایی در تحلیل بدافزار، شکستن این پوسته محافظ و دستیابی به کد اصلی (Payload) است. [۱۸] [۱۹]

۱.۴.۲ فرآیند عمومی آنپک کردن (Generic Unpacking)

فرآیند آنپک کردن بر یک اصل ساده استوار است: بدافزار برای اجرا شدن باید در نقطه‌ای از زمان، کد خود را رمزگشایی کرده و در حافظه بنویسد. تحلیل‌گران از این رفتار اجباری برای شکار کد اصلی استفاده می‌کنند.

شناسایی نقطه ورود اصلی (Original Entry Point - OEP)

هدف نهایی در آنپک کردن، یافتن OEP است؛ آدرسی که پس از پایان عملیات رمزگشایی توسط Stub، کنترل اجرا به آن منتقل می‌شود و کد اصلی بدافزار آغاز می‌گردد. روش‌های هیوریستیک متعددی برای یافتن OEP وجود دارد: [۲۰]

۱. **رهگیری نوشتن در حافظه (Memory Write Detection):** با استفاده از ابزارهای دیباگ (مانند x64dbg یا DBI، نقاطی که برنامه در آن‌ها اقدام به نوشتن داده می‌کند (توابعی مانند VirtualAlloc و WriteProcessMemory) نظارت می‌شوند. بخشی از حافظه که پس از نوشتن، دستورات اجرایی در آن قرار می‌گیرند و مجوز اجرا (Execute Permission) دریافت می‌کنند، کاندیدای اصلی OEP است.

۲. **تکنیک آنتروپی (Entropy Heuristics):** کدهای پک شده به دلیل فشردگی یا رمزنگاری، آنتروپی (بی‌نظمی) بسیار بالایی دارند. تحلیلگران با پایش تغییرات آنتروپی بلوک‌های حافظه در زمان اجرا، لحظه‌ای را که آنتروپی یک بلوک حافظه به طور ناگهانی کاهش می‌یابد (نشان‌دهنده تبدیل داده‌های رمزگذاری شده به کد اسمبلی معنادار)، به عنوان لحظه آنپک شدن شناسایی می‌کنند.

۳. **بازسازی رجیسترها (Stack Balancing):** بسیاری از پکرها (مانند UPX) در ابتدای اجرا وضعیت تمام رجیسترها را در پشته ذخیره می‌کنند (دستور PUSHAD) و دقیقاً قبل از پرش به OEP، آن‌ها را بازیابی می‌کنند (POPAD). قرار دادن یک نقطه توقف سخت‌افزاری (Hardware Breakpoint) روی آدرس پشته‌ای که رجیسترها در آن ذخیره شده‌اند، می‌تواند تحلیلگر را مستقیماً به لحظه پرش به OEP هدایت کند.

۲.۴.۲ بازسازی جدول آدرس واردات (IAT Reconstruction)

یکی از پیچیده‌ترین مراحل پس از استخراج کد آنپک شده از حافظه، بازسازی جدول آدرس واردات (Import Address Table - IAT) است. IAT جدولی است که آدرس توابع API مورد نیاز برنامه (مانند توابع Kernel32.dll یا User32.dll) در آن قرار دارد. پکرها برای سخت‌تر کردن تحلیل، IAT اصلی را تخریب کرده و تمام فراخوانی‌های تابع را به روتین‌های داخلی خود هدایت می‌کنند.

اگر فایل دامپ شده از حافظه دارای IAT بازسازی شده نباشد، اجرا نخواهد شد. فرآیند بازسازی شامل مراحل زیر است:

۱. **کشف الگوی فراخوانی:** تحلیلگر باید کدهای آتپک شده را بررسی کند تا ببیند فراخوانی‌های غیرمستقیم (Indirect Calls) به کجا اشاره می‌کنند.

۲. **استفاده از ابزارهای بازسازی (Scylla / ImpRec):** ابزارهایی مانند Scylla به پردازش معلق شده متصل می‌شوند، جدول IAT موجود در حافظه را اسکن می‌کنند، توابع نامعتبر یا تغییر مسیر داده شده توسط پکر را شناسایی کرده و آن‌ها را با آدرس‌های صحیح API‌های سیستم‌عامل جایگزین می‌کنند.

۳. **مدیریت بایت‌های دزدیده شده (Stolen Bytes):** برخی پکرها تکنیک پیشرفته‌ای به نام "Stolen Bytes" را به کار می‌برند؛ آن‌ها چند دستور اول توابع API یا کد اصلی را حذف کرده و آن‌ها را در فضای حافظه تخصیص یافته خود اجرا می‌کنند. بازسازی این موارد نیازمند تحلیل دستی دقیق و کپی بازگشتی این دستورات به فایل اصلی است تا فایل قابلیت اجرای مجدد (Re-runnable) پیدا کند.

۳.۴.۲ رمزگشایی الگوریتم‌های اختصاصی و رشته‌ها

علاوه بر پکینگ کلی فایل، بدافزارها اغلب رشته‌های حساس (مانند آدرس C2، نام فایل‌ها و دستورات) را با الگوریتم‌های اختصاصی رمزگذاری می‌کنند. تحلیلگران برای شکستن این لایه از روش‌های زیر استفاده می‌کنند:

- **شناسایی حلقه‌های رمزگشایی (Decryption Loops):** در سطح اسمبلی، روتین‌های رمزگشایی معمولاً با الگوی خاصی ظاهر می‌شوند: یک حلقه تکرار (Loop) که روی بافری از داده‌ها عملیات منطقی (مانند XOR, NOT) یا شیفت (ROL/ROR) انجام می‌دهد. با استفاده از تحلیل گراف جریان کنترل (CFG) در دیس‌اسمبلرها، این ساختارها به سرعت قابل شناسایی هستند.
- **تشخیص ثابت‌های رمزنگاری (Cryptographic Constants):** بسیاری از الگوریتم‌های استاندارد (مانند AES, MD5, SHA) از مقادیر ثابت منحصر به فردی (S-Boxes یا مقادیر اولیه بردار) استفاده می‌کنند. ابزارهایی مانند Krypto Analyzer در دیس‌اسمبلرها یا اسکریپت‌های FindCrypt می‌توانند با جستجوی این ثابت‌ها در کد باینری، نوع الگوریتم رمزنگاری را تشخیص دهند.
- **استخراج کلید با اجرای نمادین (Symbolic Execution) و DBI:** به جای تلاش برای مهندسی معکوس کامل الگوریتم ریاضی که زمان‌بر است، تحلیلگران از DBI استفاده می‌کنند تا آرگومان‌های توابع رمزنگاری را در لحظه فراخوانی شنود کنند. با هوک کردن توابع قبل از ورود به روتین

رمزنگاری، می‌توان «کلید» و «متن آشکار» (Plaintext) را مستقیماً از حافظه استخراج کرد. این روش به ویژه در برابر بدافزارهای باج‌گیر (Ransomware) برای استخراج کلیدهای رمزنگاری فایل‌ها حیاتی است.

۵.۲ شبیه‌سازی و اجرای دینامیک

شبیه‌سازی و اجرای دینامیک یکی از تکنیک‌های مهم در تحلیل و تکامل نرم‌افزار است که در آن کد اجرایی برنامه در یک محیط کنترل‌شده و مجازی اجرا می‌شود. برخلاف اجرای معمولی که برنامه مستقیماً روی سخت‌افزار یا سیستم‌عامل میزبان اجرا می‌شود، در شبیه‌سازی دینامیک تمام دستورات عمل‌ها توسط یک لایه نرم‌افزاری شبیه‌ساز دریافت، تفسیر و سپس اجرا می‌شوند. این لایه می‌تواند رفتار برنامه را به‌طور کامل رصد کند و اطلاعات دقیقی در مورد روند اجرا ارائه دهد.

هدف اصلی از این روش، فراهم‌کردن دید عمیق و قابل‌کنترلی از عملکرد برنامه است؛ دیدی که در اجرای مستقیم به دلیل محدودیت‌های سخت‌افزاری، امنیتی و طراحی سیستم معمولاً قابل دستیابی نیست. قدرت شبیه‌سازهای دینامیک در این است که تحلیل‌گر می‌تواند اجرای برنامه را در هر لحظه متوقف، تغییر مسیر داده، ورودی‌ها را دستکاری کرده و حتی وضعیت حافظه یا رجیسترها را مشاهده یا تغییر دهد. به همین دلیل این روش در سیستم‌هایی که نیاز به بررسی دقیق رفتار دارند، اهمیت ویژه‌ای پیدا می‌کند.

از دیگر مزایای اجرای دینامیک این است که برای تحلیل رفتار برنامه نیازی به سورس‌کد وجود ندارد. بسیاری از نرم‌افزارهای قدیمی، بدافزارها یا سیستم‌هایی که تحت شرایط خاص اجرا می‌شوند، تنها در قالب باینری قابل دسترس‌اند. اجرای آن‌ها در محیط واقعی، به‌خصوص زمانی که احتمال وجود رفتارهای مخرب یا باگ‌های غیرقابل‌پیش‌بینی وجود دارد، می‌تواند خطرناک باشد. اما با استفاده از شبیه‌سازی، این خطر کاملاً برطرف می‌شود، زیرا برنامه در محیطی جداسازی شده و ایزوله اجرا می‌گردد.

علاوه بر این، شبیه‌سازی دینامیک امکان بررسی اجرای برنامه روی معماری‌ها، سیستم‌عامل‌ها یا سخت‌افزارهایی را فراهم می‌کند که در دسترس تحلیل‌گر نیستند. این موضوع در حوزه‌هایی مانند توسعه سیستم‌های توکار، تحلیل سیستم‌های قدیمی یا شبیه‌سازی معماری‌های خاص پردازنده اهمیت زیادی دارد. ابزارهایی مانند QEMU نمونه‌ای از این شبیه‌سازها هستند که معماری‌های مختلف را بدون نیاز به دستگاه واقعی اجرا می‌کنند.

۱.۵.۲ استفاده از شبیه‌سازی‌های دینامیک برای تحلیل کدهای اجرایی

یکی از مهم‌ترین کاربردهای شبیه‌سازی دینامیک، تحلیل دقیق کدهای اجرایی است. در این روش، شبیه‌ساز مسیرهای اجرایی برنامه را تحت ورودی‌های مختلف دنبال می‌کند و تمامی رویدادهای مهم شامل دسترسی به حافظه، تغییر در رجیسترها، فراخوانی توابع، فراخوانی‌های سیستمی، زمان‌بندی اجرا، شاخه‌زنی‌های شرطی و حتی تعاملات شبکه‌ای را ثبت می‌کند. این سطح از جزئیات برای تحلیل‌گر ارزشمند است، زیرا امکان پایش رفتار واقعی برنامه بدون دخالت محیط خارجی فراهم می‌شود.

در حوزه امنیت، این تکنیک برای شناسایی رفتارهای مخرب به‌کار می‌رود. تحلیل‌گران بدافزار معمولاً از شبیه‌سازی دینامیک برای مشاهده فعالیت‌های پنهان یا زمان‌بندی‌شده بدافزار استفاده می‌کنند؛ فعالیت‌هایی که ممکن است در محیط واقعی قابل تشخیص نباشد. همچنین، امکان تغییر شرایط اجرای برنامه مانند شبیه‌سازی تعداد هسته‌ها، میزان حافظه یا رفتار شبکه، کمک می‌کند که تحلیل‌گر به رفتارهای پنهان دسترسی پیدا کند.

در حوزه تست نرم‌افزار نیز این روش کاربرد دارد. تست دینامیک مبتنی بر شبیه‌سازی این امکان را فراهم می‌کند که برنامه تحت سناریوهای غیرعادی یا شرایطی که بازسازی آن‌ها در دنیای واقعی دشوار است، آزمایش شود. برنامه‌نویسان می‌توانند تعیین کنند که کدام مسیرهای کد اجرا شده‌اند، کدام بخش‌ها هنوز پوشش داده نشده‌اند و رفتار برنامه در شرایط خاص چگونه است.

به‌طور کلی، استفاده از شبیه‌سازی دینامیک برای تحلیل کدهای اجرایی یکی از ابزارهای کلیدی در تکامل نرم‌افزار محسوب می‌شود. این روش امکان مشاهده رفتار واقعی برنامه را بدون خطر، بدون نیاز به سخت‌افزار واقعی و با سطح کنترلی بسیار بالا فراهم می‌کند. در نتیجه، تحلیل‌گران می‌توانند ایرادات، رفتارهای ناخواسته، وابستگی‌ها و ویژگی‌های مهم اجرایی برنامه را شناسایی و برای بهبود یا اصلاح نرم‌افزار تصمیم‌گیری کنند.

۶.۲ استفاده از روش‌های شکست کدهای خود تغییر

کدهای خودتغییر (Self-Modifying Code یا SMC) یکی از چالش‌برانگیزترین بخش‌ها در تحلیل و تکامل نرم‌افزار محسوب می‌شوند. در این نوع از برنامه‌ها، کد اجرایی در زمان اجرا اقدام به تغییر بخش‌هایی از خود می‌کند؛ تغییری که می‌تواند شامل بازنویسی دستورالعمل‌ها، جابه‌جایی بخش‌های کد، رمزگذاری و رمزگشایی پویا یا تولید ساختارهای جدید اجرایی باشد. دلیل اصلی استفاده از این تکنیک، جلوگیری از تحلیل معکوس، پیچیده‌سازی روند اشکال‌زدایی یا بهبود عملکرد در وضعیت‌های خاص است. به همین دلیل، این روش بیشتر در بدافزارها، سیستم‌های حفاظتی نرم‌افزار، مفسرها،

موتورهای مجازی و برخی سیستم‌های بهینه‌سازی دیده می‌شود.

از آنجا که رفتار کد در هر مرحله می‌تواند تغییر کند، تحلیل آن با روش‌های سنتی بسیار دشوار است. نه تنها مسیر اجرای برنامه قابل پیش‌بینی نیست، بلکه حتی ساختار کلی باینری نیز در طول اجرا ثابت نمی‌ماند. بنابراین نیاز به مجموعه‌ای از تکنیک‌ها و ابزارهای تخصصی وجود دارد که بتوانند این نوع رفتار پویا را شناسایی، ردیابی و تحلیل کنند. هدف اصلی روش‌های شکست کدهای خودتغییر، شفاف‌سازی روند تغییرات و ارائه‌ی تصویری ثابت یا حداقل قابل‌درک از ساختار نهایی و مسیرهای اجرایی برنامه است.

۱.۶.۲ تکنیک‌هایی که به شکستن کدهایی که خودشان را تغییر می‌دهند کمک می‌کند

برای مقابله با کدهای خودتغییر، تحلیل‌گران از مجموعه‌ای از تکنیک‌های دقیق، ترکیبی و گاهی چندمرحله‌ای استفاده می‌کنند. یکی از مهم‌ترین روش‌ها Controlled Dynamic Execution است؛ جایی که برنامه در یک شبیه‌ساز یا ابزار چندلایه اجرا شده و هر تغییر در حافظه، رجیسترها یا بخش‌های اجرایی ضبط می‌شود. این روش به تحلیل‌گر اجازه می‌دهد لحظه‌ای که کد دچار تغییر می‌شود را شناسایی کرده و نسخه‌ی قبل و بعد از تغییر را استخراج کند.

روش رایج دیگر، Snapshot-Based Analysis است. در این روش، وضعیت کامل حافظه در بازه‌های زمانی مختلف ذخیره شده و تفاوت آن‌ها مقایسه می‌شود. این مقایسه می‌تواند به سرعت نقاطی از حافظه که در حال بازنویسی هستند را مشخص کند. ابزارهایی مانند QEMU، PANDA یا Intel PIN در این حوزه بسیار کاربرد دارند.

Deobfuscation و Dynamic Unpacking پویا از دیگر تکنیک‌های مهم محسوب می‌شود. بسیاری از کدهای خودتغییر ابتدا نسخه‌ی رمزگذاری شده یا فشرده‌ی خود را اجرا می‌کنند و در زمان اجرا نسخه‌ی واقعی را تولید می‌کنند. تحلیل‌گران با دنبال کردن این فرآیند و استخراج نسخه‌ی «بازشده»، تلاش می‌کنند کد را به وضعیت پایدار برگردانند.

تکنیک مهم دیگر، نظارت بر نواحی حافظه‌ای است که هم قابل‌نوشتن و هم قابل‌اجرا هستند (Write-Execute Regions). در این نواحی احتمال وجود رفتار خودتغییر بالا است. تحلیل‌گران با مانیتور کردن این بخش‌ها یا با فعال کردن محافظت‌هایی مانند WX می‌توانند فعالیت‌های مشکوک را شناسایی کنند.

در موارد پیشرفته‌تر، ترکیبی از Static Slicing و Dynamic Taint Analysis استفاده می‌شود

تا مشخص شود چه بخش‌هایی از برنامه باعث تغییر بقیه بخش‌ها شده‌اند. این روش به تحلیل‌گر کمک می‌کند منبع تغییرات و مسیرهای داده‌ای که منجر به بازنویسی کد شده‌اند را شناسایی کند.

در نهایت، نتیجه‌ی این تکنیک‌ها ایجاد یک نسخه‌ی شبه‌پایدار از برنامه است که برای تحلیل، مهندسی معکوس یا ارزیابی امنیتی قابل استفاده است. این فرآیند اگرچه چالش‌برانگیز است، اما یکی از مهم‌ترین بخش‌های تحلیل کدهای سطح پایین و بدافزارهای پیشرفته محسوب می‌شود و نقش مهمی در تکامل نرم‌افزار، امنیت و درک بهتر رفتار سیستم‌های پیچیده ایفا می‌کند.

۷.۲ مقاومت در برابر تحلیل‌های ضد ضد دیس‌اسمبلی

با پیشرفت گسترده ابزارهای تحلیل معکوس، تکنیک‌های کلاسیک ضد دیس‌اسمبلی دیگر به‌تنهایی نمی‌توانند امنیت کافی فراهم کنند. ابزارهای مدرنی همچون Binary Ninja، Ghidra، IDA Pro، Radare2 و چارچوب‌های تحلیل پویا مانند DynamoRIO و Intel PIN قادرند بسیاری از تکنیک‌های قدیمی را شناسایی، نرمال‌سازی یا خنثی کنند [۲۱]. علاوه بر این، موتورهای تحلیل نمادین (Symbolic Execution) مانند angr و KLEE نیز توانایی عبور از برخی موانع جریان کنترل را فراهم آورده‌اند.

در چنین شرایطی، نیاز به روش‌هایی برای ایجاد «لایه‌های مقاومت علیه ابزارهای Anti-Anti-Disassembly» به‌وجود آمده است. این لایه‌ها نه تنها تحلیل استاتیک را دشوار می‌سازند، بلکه تحلیل پویا، اجرای نمادین و روش‌های خودکار مدرن را نیز با شکست مواجه می‌کنند. این بخش به بررسی جامع تکنیک‌هایی می‌پردازد که با هدف ایجاد مقاومت پایدار در برابر تحلیلگر انسانی و ابزارهای پیشرفته AADA طراحی شده‌اند.

۱.۷.۲ لایه‌بندی چندمرحله‌ای ابهام‌سازی

در روش‌های سنتی، ابهام‌سازی معمولاً در یک سطح و با تکنیک‌هایی قابل پیش‌بینی اجرا می‌شد؛ اما ابزارهای ضد مبهم‌سازی قادرند بسیاری از آن‌ها را بازسازی کنند. روش چندمرحله‌ای جدید بر پایه وابستگی میان لایه‌ها طراحی می‌شود؛ به‌گونه‌ای که حذف یک لایه باعث فعال شدن لایه‌های دیگر یا ایجاد اجرای اشتباه می‌شود [۲۲].

این وابستگی‌ها معمولاً شامل موارد زیر است:

- ابهام‌سازی جریان کنترل (Control Flow Obfuscation)

- درهم‌سازی داده (Data Obfuscation)
- رمزگذاری پویا (Dynamic Code Encryption)
- رشته‌های رمزگذاری‌شده (Encrypted Strings)
- قطعه‌قطعه کردن کد (Code Splitting)
- بازسازی مرحله‌ای کد (Staged Reconstruction)

چنین ساختاری پیچیدگی زیادی ایجاد می‌کند و حتی ابزارهایی مانند Ghidra تنها می‌توانند بخش‌هایی از CFG را بازسازی کنند و تحلیل کامل برنامه غیرممکن می‌شود [۲۱، ۲۳].

۲.۷.۲ رفتار آگاه از تحلیل

این تکنیک بر اساس این اصل عمل می‌کند که برنامه بتواند وجود تحلیلگر یا ابزار تحلیل را تشخیص دهد و مسیر اجرای خود را تغییر دهد. این رفتارها معمولاً شامل موارد زیر هستند:

شناسایی Breakpoint ابزارهایی مانند IDA Pro و x64dbg هنگام قرار دادن نقطه توقف تغییراتی در حافظه ایجاد می‌کنند که قابل شناسایی است [۲۱].

تحلیل زمان‌بندی (Timing Analysis) اجرای برنامه در VM یا Emulator باعث تغییر رفتار زمانی می‌شود. یک حلقه ساده با شمارنده می‌تواند اختلاف را آشکار کند.

تشخیص محیط سندباکس سندباکس‌هایی مانند Cuckoo Sandbox دارای نشانه‌هایی همچون:

- پردازش‌های تکراری،
- اندازه ثابت حافظه،
- MAC Addressهای غیرواقعی،
- فایل‌های سیستم شبیه‌سازی‌شده

هستند و برنامه می‌تواند با چند بررسی ساده آنها را شناسایی کند [۲۴].

واکنش به تحلیل در صورت شناسایی تحلیلگر، برنامه ممکن است:

- مسیر اجرای واقعی را پنهان کند،
- اجرای جعلی نمایش دهد،
- بخش‌هایی از کد را بازنویسی کند،
- یا اجرای خود را متوقف سازد.

۳.۷.۲ وابستگی به سخت‌افزار واقعی

بخش‌های حساس برنامه می‌توانند به ویژگی‌های سخت‌افزاری واقعی وابسته شوند که در VM یا Emulator قابل شبیه‌سازی دقیق نیستند:

- شمارنده‌های سخت‌افزاری (Performance Monitoring Unit)
- رفتار دقیق Cache L1/L2
- وقفه‌های سیستم‌عامل
- زمان‌بندی چرخه CPU (RDTSC)

این وابستگی‌ها باعث می‌شوند تحلیل دقیق در محیط شبیه‌سازی شده غیرممکن باشد [۲۳].

۴.۷.۲ جریان کنترل غیرقطعی

در این روش، جریان کنترل برنامه در اجرای‌های مختلف تغییر می‌کند. این تغییرات ممکن است ناشی از:

- داده‌های تصادفی،
- زمان دقیق اجرا،
- وقفه‌های خارجی،
- تخصیص حافظه متفاوت

نتیجه:

- CFG ناپایدار می‌شود،
- اجرای نمادین (Symbolic Execution) شکست می‌خورد،
- مسیر اجرای برنامه قابل بازتولید نیست.

۵.۷.۲ کد خودتغییردهنده چندمرحله‌ای

در نسخه چندمرحله‌ای، هر مرحله فقط بخش کوچکی از کد را بازسازی می‌کند، مرحله بعد فقط پس از اجرای مرحله قبل قابل دستیابی است، هیچ Dump کاملی از کد وجود ندارد و خروجی برنامه در هر اجرا متفاوت است [۲۲، ۲۵].

۶.۷.۲ جمع‌بندی

مقاومت در برابر Anti-Anti-Disassembly یعنی ایجاد تکنیک‌هایی که حتی پس از عبور تحلیلگر از موانع اولیه، روند تحلیل را کند، ناقص یا غیرقابل اعتماد می‌سازند. ترکیب روش‌هایی مانند:

- ابهام‌سازی لایه‌ای،
- رفتار آگاه از تحلیل،
- وابستگی سخت‌افزاری،
- جریان کنترل غیرقطعی،
- بازنویسی چندمرحله‌ای کد

سطحی از محافظت ایجاد می‌کند که حتی ابزارهای پیشرفته نیز قادر به تحلیل پایدار آن نیستند.

فصل ۳

نتیجه‌گیری

گزارش حاضر تصویری جامع از مجموعه‌ای گسترده از روش‌ها و تکنیک‌های مقابله با تحلیل معکوس، ضد دیس‌اسمبلی و روش‌های پیشرفته تحلیل بدافزار ارائه داد. با بررسی جنبه‌های مختلف این حوزه، می‌توان نتیجه گرفت که ماهیت تحلیل معکوس به‌گونه‌ای است که همواره در حال تکامل و پیچیده‌تر شدن است و به همان میزان، تکنیک‌های دفاعی نیز باید عمیق‌تر، پویاتر و هوشمندانه‌تر طراحی شوند. از مدل‌های ساده‌ی جلوگیری از دیس‌اسمبلی تا روش‌های پیچیده‌ی رفتارآگاه، شبیه‌سازی پویا و جریان کنترل غیرقطعی، همگی نشان‌دهنده‌ی ضرورت ایجاد لایه‌های ترکیبی و چندمرحله‌ای دفاع برای حفاظت از نرم‌افزارهای حساس هستند.

در بخش‌های ابتدایی گزارش، تکنیک‌های کلاسیک ضد تحلیل همچون جلوگیری از دیس‌اسمبلی، استفاده از دستورالعمل‌های غیرمعمول و مبهم‌سازی کد بررسی شد. این روش‌ها معمولاً با ایجاد اختلال در اسکنرهای استاتیک یا ابزارهای بازسازی کد، اولین لایه‌ی دفاعی یک برنامه را تشکیل می‌دهند. با وجود سادگی ظاهری، این تکنیک‌ها همچنان در بسیاری از نرم‌افزارهای تجاری و بدافزارها استفاده می‌شوند و نقش مهمی در افزایش هزینه و زمان تحلیل برای مهاجم دارند.

در ادامه، تکنیک‌های پیچیده‌تری مانند مبهم‌سازی پویا، کد خودتغییردهنده و رمزگذاری کد مورد بررسی قرار گرفتند. ویژگی مشترک این روش‌ها، وابستگی شدید به زمان اجرا و تغییر ساختار برنامه به‌صورت پویا است. این موضوع باعث می‌شود تحلیلگر نتواند با تحلیل استاتیک یا بازسازی باینری، تصویر کاملی از برنامه به دست آورد و ناچار شود به تحلیل پویا روی بیاورد – تحلیلی که بسیار زمان‌برتر، کندتر و شکننده‌تر است. وجود رمزگذاری کد و بازسازی مرحله‌ای نیز باعث می‌شود تنها بخش کوچکی از کد در هر لحظه در حالت قابل تحلیل قرار گیرد، که این خود یک مانع بسیار مؤثر در برابر ابزارهای تحلیل نیمه‌خودکار به‌شمار می‌رود.

در بخش‌های مربوط به تحلیل رفتاری و تحلیل پویا نیز مشخص شد که حتی زمانی که ابزار تحلیل قادر به عبور از لایه‌های اولیه شود، می‌توان با استفاده از رفتارهای آگاه از محیط و تحلیل، مانند شناسایی دیباگر، تشخیص ماشین مجازی، یا بررسی ناهنجاری‌های زمانی، مسیر اجرای برنامه را تغییر داد یا کد واقعی را در اختیار تحلیلگر قرار نداد. این دسته از تکنیک‌ها باعث می‌شوند حتی ابزارهای پیشرفته‌ی اجرای نمادین، شبیه‌سازی پویا یا سیستم‌های خودکار تحلیل بدافزار نیز در استخراج منطق واقعی برنامه شکست بخورند.

در نهایت، در بخش مقاومت در برابر تحلیل‌های Anti-Anti-Disassembly روشن شد که تنها استفاده از تکنیک‌های کلاسیک کافی نیست، زیرا ابزارهای جدید قادرند بسیاری از این موانع را دور بزنند. بنابراین، استفاده از لایه‌بندی چندگانه ابهام‌سازی، وابستگی به سخت‌افزار واقعی، کنترل جریان غیرقطعی و بازنویسی پویا ضرورت پیدا می‌کند. این روش‌ها نه تنها عبور از مرحله‌ی ضدتحلیل اولیه را دشوار می‌کنند، بلکه حتی پس از عبور تحلیلگر از برخی لایه‌ها نیز همچنان تحلیل را غیرقابل اعتماد، ناقص یا کاملاً غیرممکن می‌سازند.

جمع‌بندی کلی نشان می‌دهد که دفاع مؤثر در برابر تحلیل معکوس، نه در یک تکنیک منفرد بلکه در ترکیبی از روش‌ها، لایه‌ها و رفتارهای وابسته به محیط نهفته است. به عبارت دیگر، امنیت واقعی از هم‌افزایی تکنیک‌ها به وجود می‌آید؛ یعنی جایی که هر لایه نقطه‌ی ضعف لایه‌ی دیگر را پوشش می‌دهد و یک ساختار دفاعی پویا و تطبیق‌پذیر را شکل می‌دهد. همچنین روشن شد که این حوزه به‌طور دائم در حال تکامل است و برای مواجهه با ابزارهای تحلیل پیشرفته‌ی نسل جدید، نیاز به به‌کارگیری تکنیک‌های نوین مبتنی بر سخت‌افزار، ویژگی‌های ریزمعماری و حتی روش‌های یادگیری ماشین وجود خواهد داشت.

به‌طور کلی، گزارش نشان می‌دهد که مقابله با مهندسی معکوس یک نبرد مداوم میان تحلیل‌گران و توسعه‌دهندگان است – نبردی که در آن برنده کسی است که بتواند پیچیدگی بیشتری ایجاد کند، رفتار نرم‌افزار را غیرقابل پیش‌بینی‌تر سازد و مسیر تحلیل را تا حد ممکن طولانی و پرهزینه کند. آینده‌ی این حوزه بدون شک ترکیبی از دفاع چندلایه، تحلیل مبتنی بر داده، و تکنیک‌های وابسته به محیط واقعی خواهد بود؛ مسیری که هم‌اکنون نیز نشانه‌های آن در بسیاری از نرم‌افزارهای پیشرفته و بدافزارهای پیچیده دیده می‌شود.

- [١] Christopher Kruegel, William Robertson, and Giovanni Vigna. "Detecting Kernel-Level Rootkits Through Binary Analysis". In: Proceedings of the USENIX Security Symposium. ٢٠٠٦.
- [٢] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Limits of Static Analysis for Malware Detection". In: Proceedings of the ٢٠٠٧ Annual Computer Security Applications Conference (ACSAC). ٢٠٠٧.
- [٣] Manuel Egele et al. "A Survey on Automated Dynamic Malware Analysis Techniques and Tools". In: ACM Computing Surveys ٤٤.٢ (٢٠٠٨).
- [٤] Dae Young Kwon and Seunghoon Y. Kim. "Anti-Debugging Techniques and Dynamic Analysis Evasion". In: IEEE Security & Privacy ١٢.٢ (٢٠١٤), pp. ٧٧–٨٣.
- [٥] Brendan Dolan-Gavitt et al. "Dynamic Binary Instrumentation: A Detailed Analysis". In: IEEE Transactions on Software Engineering (٢٠١٦).
- [٦] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. "A Model for Self-Modifying Code". In: Information Hiding, ٨th International Workshop, IH ٢٠٠٦, Revised Selected Papers. Vol. ٤٤٣٧. Lecture Notes in Computer Science. Springer, ٢٠٠٦, pp. ٢٣٢–٢٤٨. DOI: [10.1007/978-3-540-74124-4_16](https://doi.org/10.1007/978-3-540-74124-4_16).
- [٧] Twingate Team. What is Disassembly? ٢٠٢٤. URL: <https://www.twingate.com/blog/glossary/disassembly>.
- [٨] Liviu PETREAN. POLYMORPHIC AND METAMORPHIC CODE APPLICATIONS IN PORTABLE EXECUTABLE FILES PROTECTION. ٢٠١٠. URL: https://users.utcluj.ro/~atn/papers/ATN%5C_1%5C_2010%5C_1.pdf.

- [٩] What is Polymorphic Malware? URL: <https://www.threatdown.com/glossary/what-is-polymorphic-malware/>.
- [١٠] fiveable. Anti-reverse engineering. ٢٠٢٥. URL: <https://fiveable.me/network-security-and-forensics/unit-4/anti-reverse-engineering/study-guide/2yna2P93Bt4M4uSd>.
- [١١] paloaltonetworks. Why You Need Static Analysis, Dynamic Analysis, and Machine Learning? ٢٠٢٤. URL: <https://www.paloaltonetworks.com/cyberpedia/why-you-need-static-analysis-dynamic-analysis-machine-learning>.
- [١٢] Tho Le. Malware Analysis Lab and Behavioral Analysis Steps. ٢٠٢١. URL: <https://tho-le.medium.com/malware-analysis-lab-and-behavioral-analysis-steps-229cf8c15d5e>.
- [١٣] Neil Fox. How to Unpack Malware with x64dbg. ٢٠٢٥. URL: <https://www.varonis.com/blog/x64dbg-unpack-malware>.
- [١٤] Tony K Rodgers. Anti-Reverse Engineering. ٢٠٢٥. URL: https://medium.com/@Tony_K_Rodgers/anti-reverse-engineering-e6adcf1ded04.
- [١٥] apriorit. Anti Debugging Protection Techniques with Examples. ٢٠٢٤. URL: <https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software>.
- [١٦] Michael Sikorski and Andrew Honig. Practical Malware Analysis. oreilly, ٢٠١٢.
- [١٧] Omer Sezgin Uğurlu. "STEALTH SANDBOX ANALYSIS OF MALWARE". Master's thesis. bilkent university, ٢٠٠٩.
- [١٨] Mark Vincent Yason. The Art of Unpacking. ٢٠٠٧. URL: <https://blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf>.
- [١٩] Nicole Fishbein. Beginner's guide to malware analysis and reverse engineering. ٢٠٢٥. URL: <https://intezer.com/beginners-guide-to-malware-analysis-and-reverse-engineering/>.
- [٢٠] Julien Lenoir. Implementing your own generic unpacker. ٢٠١٥. URL: <https://airbus-seclab.github.io/gunpack/HITB-GSEC-SG2015-Lenoir-Gunpack.pdf>.
- [٢١] Chris Eagle. The IDA Pro Book. No Starch Press, ٢٠٢٠.

-
- [٢٢] Bruce Dang, Alexandre Gazet, and Elias Bachaalany. Practical Reverse Engineering. Wiley, ٢٠١٤.
- [٢٣] Dennis Andriesse. Practical Binary Analysis. No Starch Press, ٢٠١٨.
- [٢٤] Manuel Egele et al. "A Survey on Automated Dynamic Malware Analysis Techniques". In: ACM Computing Surveys (٢٠١٢).
- [٢٥] Mikel Ugarte and Juan Caballero. "On the Resilience of Obfuscation Techniques against Symbolic Execution". In: IEEE Security and Privacy Workshops. ٢٠١٩.