دانشگاه صنعتی خواجه نصیرالدین طوسی

دانشکده مهندسی کامپیوتر

گزارش گروهی دوم درس تکامل نرم افزار

نویسندگان:

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

استاد راهنما: دکتر محمدهادی علائیان

۲۲ آذر ۱۴۰۴

چکیده

گزارش حاضر با هدف تحلیل معماری دفاعی نرم‌افزار، به بررسی روش‌های متقابل در برابر تحلیل کدهای اجرایی می‌پردازد. با توجه به اهمیت حیاتی جلوگیری از مهندسی معکوس و سوءاستفاده از مالکیت فکری و آسیب‌پذیری‌های امنیتی، توسعه‌دهندگان به سمت استفاده از تکنیک‌های ضد دیس‌اسمبلی (AD) سوق داده شده‌اند. در این راستا، تکنیک‌های اصلی AD شامل ابهام‌سازی کد در سطوح واژگانی، داده‌ای و جریان کنترل، بهره‌گیری از دستورات و ساختارهای غیرمعمول و وابسته به حالت پردازنده برای شکستن تحلیل استاتیک، روش‌های پنهان‌سازی مبتنی بر تجزیه و تحلیل پویا نظیر رمزگشایی زمان اجرا و تکنیک‌های ضد دیباگ، و همچنین تکنیک‌های رمزگذاری مانند کدهای پلی‌مورفیک و متامورفیک مورد بررسی قرار می‌گیرند.

این گزارش به فلسفه و تکنیک‌های ضد ضد دیس‌اسمبلی (AADA) می‌پردازد که در پاسخ به پیچیدگی‌های AD توسط تحلیل‌گران امنیتی ایجاد شده‌اند. این تکنیک‌ها شامل تحلیل دینامیک و رصد کد برای کشف مسیر اجرای واقعی، استفاده از دیس‌اسمبلرهای پیشرفته با قابلیت‌های تحلیل هوشمند و اسکریپت‌نویسی، بهره‌گیری از اجرای نمادین و شبیه‌سازی داینامیک برای کاوش در مسیرهای اجرایی مبهم، و وصله‌زدن و اصلاح کد برای بازسازی نمودار جریان کنترل است.

نتایج نشان می‌دهند که مقابله با مهندسی معکوس یک «مسابقه تسلیحاتی» دائمی میان روش‌های دفاعی AD و روش‌های تحلیلی AADA است. موفقیت در حفظ امنیت و تکامل نرم‌افزار، مستلزم درک عمیق از هر دو جبهه و استفاده هوشمندانه و ترکیبی از ابزارهای تحلیل استاتیک و دینامیک برای خنثی‌سازی مؤثر لایه‌های ابهام‌سازی است.

فهرست مطالب

فهرست تصاویر

فصل ۱

فرایندهای مهندسی نرم‌افزار و چرخه‌های تکامل تا پیدایش

۱.۱ روش‌های Anti Anti Disassembly

۱.۱.۱ 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.

۲.۱.۱ تعریف (AD) Anti-Disassembly:

Anti-Disassembly مجموعه‌ای از ترفندها و الگوهای برنامه‌نویسی است که عمداً 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.

۳.۱.۱ 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.

۴.۱.۱ 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.

۵.۱.۱ تکنیک‌های کلیدی ضد ضد دی‌اسمبل (AADA)

تکنیک‌های AADA به طور 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 اشکال‌زداها (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 لحظهٔ اجرای دستورات حیاتی مشاهده کند.
- ردیابی دستورات (Instruction Tracing) در این روش توالی دقیق دستوراتی که در طول اجرای بخش مبهم‌سازی‌شده اجرا می‌شوند ثبت می‌گردد. این ردیابی می‌تواند آدرس‌های پرش‌های داینامیک 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 لحظهٔ پرش کشف می‌کند و آدرس مقصد واقعی را به‌صورت 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 داینامیک به پرش استاتیک پس از کشف آدرس مقصد پرش داینامیک، 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 اُبفوسکیشن‌های 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 ابزارهایی 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.

۲.۱ استفاده از تکنیک‌های تغییر ساختار کد

مهندسی معکوس، به مجموعه‌ای از تکنیک‌ها اطلاق می‌شود که به تحلیل‌گر اجازه می‌دهد با استفاده از ابزارهایی مانند دیس‌اسمبلرها و دی‌کامپایلرها، فایل اجرایی را به یک نمایش سطح پایین‌تر یا حتی یک بازسازی نزدیک به کد منبع اصلی تبدیل کند. این امر درهای سوءاستفاده را به روی افراد مخرب باز می‌کند؛ از جمله دور زدن مکانیزم‌های حفاظتی نرم‌افزار، سرقت مالکیت معنوی برای ساخت محصولات رقیب، و شناسایی آسیب‌پذیری‌های امنیتی برای بهره‌برداری‌های بعدی.

به منظور ایجاد یک مانع جدی در برابر این تهدیدات، از رویکردی به نام مبهم‌سازی کد استفاده می‌شود. مبهم‌سازی فرآیندی است که طی آن، کد برنامه به نسخه‌ای دیگر تبدیل می‌شود 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 می‌گردد. 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. تکنیک‌های مبهم‌سازی را می‌توان در دسته‌بندی‌های مختلفی مورد بررسی قرار داد.

۱.۲.۱ دسته‌بندی تکنیک‌های مبهم‌سازی کد

این تکنیک‌ها بر اساس جنبه‌ای از برنامه که هدف قرار می‌دهند، به سه دسته اصلی تقسیم می‌شوند: مبهم‌سازی واژگانی، مبهم‌سازی داده‌ها و مبهم‌سازی جریان کنترل.

۱. مبهم‌سازی واژگانی

این دسته از تکنیک‌ها بر روی ساختار سطحی و نوشتاری کد و اطلاعات مرتبط با آن تمرکز دارند و به عنوان اولین لایه دفاعی عمل می‌کنند.

تغییر نام شناسه‌ها: این روش یکی از اساسی‌ترین و در عین حال مؤثرترین تکنیک‌ها است. توسعه‌دهندگان به صورت طبیعی از اسامی معنادار برای اجزای مختلف کد مانند متغیرها، توابع و کلاس‌ها استفاده می‌کنند تا خوانایی و قابلیت نگهداری کد را تضمین کنند. این اسامی، سرنخ‌های معنایی بسیار ارزشمندی را در اختیار تحلیل‌گر قرار می‌دهند. تکنیک تغییر نام، به صورت خودکار این اسامی بامفهوم را با شناسه‌های کوتاه، بی‌معنی و غیرقابل پیش‌بینی جایگزین می‌کند. در نتیجه، کدی که توسط دی‌کامپایلر بازسازی می‌شود، فاقد هرگونه زمینه معنایی است و تحلیل‌گر را مجبور می‌سازد 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 فراداده‌ای و اشکال‌زدایی: فایل‌های اجرایی، به خصوص در پلتفرم‌های مدیریت‌شده، اغلب حاوی اطلاعات اضافی به نام فراداده هستند. این اطلاعات شامل ساختار کلی برنامه، نام کلاس‌ها، فضاهای نام و همچنین اطلاعات اشکال‌زدایی است که می‌تواند مسیر تحلیل را برای مهندس معکوس بسیار هموار کند. حذف این اطلاعات غیرضروری، تحلیل‌گر را از یک دید کلی و ساختاریافته نسبت به برنامه محروم کرده و او را مجبور به تحلیل در سطح پایین‌تر و جزئی‌تر می‌کند.

۲. مبهم‌سازی داده‌ها

این تکنیک‌ها بر نحوه ذخیره‌سازی، نمایش و ساختار داده‌ها در برنامه تمرکز دارند تا فهم مقادیر و روابط بین آن‌ها را مختل کنند.

رمزنگاری رشته‌ها: رشته‌های متنی ثابت که در کد برنامه وجود دارند، اهداف اولیه‌ی تحلیل‌گران هستند. این تکنیک، تمام این رشته‌ها را در زمان کامپایل به فرمتی غیرقابل خواندن تبدیل کرده و در فایل اجرایی ذخیره می‌کند. در زمان اجرا، تنها در لحظه‌ای که برنامه به یک رشته نیاز دارد، یک روتین مخصوص آن را به حالت اولیه بازمی‌گرداند. این فرآیند باعث می‌شود که تحلیل‌گر نتواند با یک جستجوی ساده به اطلاعات حساس دست یابد و ابتدا باید مکانیزم رمزگشایی را شناسایی و مهندسی معکوس کند.

تبدیل ساختار داده‌ها: ساختارهای داده منطقی مانند کلاس‌ها، داده‌های مرتبط را به صورت یکپارچه سازماندهی می‌کنند. این تکنیک این پیوندهای منطقی را از هم می‌گسلد. به عنوان مثال، یک ساختار داده‌ای منسجم می‌تواند به چندین آرایه جداگانه و نامرتبط تبدیل شود. این امر درک مدل داده‌ای برنامه و نحوه ارتباط اجزای مختلف داده با یکدیگر را برای تحلیل‌گر فوق‌العاده دشوار می‌سازد.

کدگذاری متغیرها: به جای ذخیره‌سازی مقدار یک متغیر به صورت مستقیم، مقدار آن تحت یک تبدیل منطقی یا حسابی قرار گرفته و به آن شکل ذخیره می‌شود. 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 می‌شود که مشاهده مستقیم مقادیر متغیرها در حافظه در حین فرآیند دیباگ، اطلاعات مفیدی را در اختیار تحلیل‌گر قرار ندهد، زیرا مقادیر مشاهده شده، مقادیر واقعی نیستند.

۳. مبهم‌سازی جریان کنترل

این دسته از تکنیک‌ها، که از پیچیده‌ترین و قوی‌ترین روش‌ها هستند، منطق و مسیر اجرای دستورات برنامه را هدف قرار می‌دهند.

درج کدهای بی‌اثر و گزاره‌های مبهم: در این روش، دستورات و بلوک‌های کدی به برنامه اضافه می‌شوند که هیچ تأثیر واقعی بر خروجی نهایی آن ندارند. شکل پیشرفته‌تر این تکنیک، استفاده از گزاره‌های مبهم است. یک گزاره مبهم، یک عبارت شرطی است که نتیجه آن همواره ثابت است، اما به گونه‌ای طراحی شده که ابزارهای تحلیل استاتیک قادر به تشخیص این ماهیت ثابت نباشند. این گزاره‌ها برای ایجاد انشعاب‌های کنترلی جعلی در برنامه استفاده می‌شوند که تحلیل‌گر انسانی و ابزارهای خودکار را به مسیرهای تحلیلی اشتباه و بی‌نتیجه هدایت می‌کنند.

صاف‌سازی جریان کنترل: این یک تکنیک بسیار قدرتمند است که ساختارهای منطقی و قابل فهم برنامه مانند حلقه‌ها و دستورات شرطی را به طور کامل از بین می‌برد. در این روش، بدنه یک تابع به بلوک‌های کد کوچکتر تقسیم می‌شود. سپس تمام این بلوک‌ها در داخل یک ساختار کنترلی مرکزی قرار می‌گیرند. یک متغیر حالت تعیین می‌کند که در هر لحظه کدام بلوک کد باید اجرا شود و هر بلوک پس از اجرا، این متغیر حالت را برای تعیین بلوک بعدی به‌روزرسانی می‌کند. نتیجه، یک ساختار اجرایی شبیه به کلاف سردرگم است که فاقد هرگونه جریان منطقی قابل تشخیص بوده و دنبال کردن آن برای تحلیل‌گر تقریباً غیرممکن است.

۳.۱ استفاده از دستورات و ساختارهای غیرمعمول

دستورات و ساختارهای غیرمعمول یکی از تکنیک‌های مؤثر در Anti-Disassembly هستند که با بهره‌گیری از ویژگی‌های خاص معماری پردازنده، تحلیل استاتیک کد را برای دیس‌اسمبلرها با چالش مواجه می‌کنند.

۱.۳.۱ معرفی دستورات و روش های غیرمعمول

دستورات غیرمعمول به دستوراتی اطلاق می‌شود که یا به ندرت در کدهای عادی استفاده می‌شوند یا رفتار پیچیده‌ای دارند که پردازش آن‌ها برای دیس‌اسمبلرها 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 می‌شود. دیس‌اسمبلرها که معمولاً بر پایه تحلیل خطی کد کار می‌کنند، در تشخیص مسیر صحیح اجرا دچار اشتباه می‌شوند. 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، دیس‌اسمبلر در تشخیص مسیر اجرای واقعی دچار سردرگمی می‌شود. بخش ابتدایی با مقایسه رجیسترها و پرش شرطی یک تصمیم‌گیری ساده نشان می‌دهد، اما مسیر unusual-path با خواندن فلگ‌های پردازنده (pushf/pop) و ایجاد محاسبات پیچیده (ror/xor)، شرایط اجرایی مبهمی ایجاد می‌کند که وابسته به داده‌های پویای زمان اجراست. این پیچیدگی باعث می‌شود دیس‌اسمبلر 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 دیس‌اسمبلرها 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، دیس‌اسمبلرها را گمراه می‌کند. مکانیزم کار به این صورت است که ابتدا مقدار صفر به پشته بارگذاری شده، سپس علامت آن تغییر می‌کند که یک عمل غیرمعمول محسوب می‌شود. در ادامه با ذخیره و خارج کردن غیراستاندارد از پشته و مقایسه همراه با حذف عناصر، یک الگوی استفاده پیچیده از پشته FPU ایجاد می‌شود که بسیاری از دیس‌اسمبلرها در تحلیل صحیح آن دچار مشکل می‌شوند.

استفاده از دستورات خود-تغییردهنده کد

یکی از پیچیده‌ترین تکنیک‌های ضد دیس‌اسمبل، استفاده از کد خود-تغییردهنده است که در آن برنامه در حین اجرا، بخشی از کد خود را تغییر می‌دهد. در این تکنیک، برنامه در زمان اجرا دستورات خود را اصلاح می‌کند که این کار تحلیل استاتیک را غیرممکن می‌سازد. دیس‌اسمبلرها که معمولاً کد را به صورت استاتیک تحلیل می‌کنند، قادر به پیش‌بینی 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، دیس‌اسمبلرها را به طور کامل گمراه می‌کند. در ابتدا بافر کد حاوی دستورات بی‌اثر NOP است، اما در حین اجرا به دنباله‌ای از دستورات MOV EAX و RET تبدیل می‌شود. دیس‌اسمبلر که تنها محتوای اولیهٔ بافر را می‌بیند، قادر به تشخیص کد نهاییِ اجراشده 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 رشته‌ها

دستورات رشته‌ای به دلیل وابستگی به رجیسترها می‌توانند پیچیدگی ایجاد کنند:

- REP MOVSB با تنظیمات غیرمعمول رجیسترها
- SCASB با مقادیر جستجوی پیچیده
- CMPSW با اندازه‌های غیراستاندارد

دستورات سیستم و ممیزی

استفاده از دستورات سطح سیستم می‌تواند دیس‌اسمبلرها را دچار خطا کند:

- IN و OUT برای دسترسی به پورت‌ها
- دستورات SGDT، SIDT، SLDT
- دستورات VERR، VERW

۳.۳.۱ مکانیزم عملکرد ضد دیس‌اسمبل

دستورات غیرمعمول از چند طریق بر دیس‌اسمبلرها تأثیر می‌گذارند:

ایجاد خطا در تحلیل جریان کنترل

- **شکستن تحلیل جریان خطی**: با ایجاد پرش‌های غیرقابل پیش‌بینی
- **ایجاد بلوک‌های کد مبهم**: با ساختارهای شرطی پیچیده
- **مختل کردن تحلیل داده**: با دستوراتی که وابستگی داده‌ای پیچیده ایجاد می‌کنند

سوءاستفاده از محدودیت‌های دیس‌اسمبلر

- **عدم پشتیبانی از دستورات نادر**: برخی دیس‌اسمبلرها دستورات کم‌کاربرد را نمی‌شناسند
- **خطا در تحلیل حالت پردازنده**: دستورات وابسته به حالت را به درستی پردازش نمی‌کنند
- **مشکل در تحلیل دستورات** FPU: پیچیدگی محاسبات ممیز شناور باعث خطا می‌شود

۴.۳.۱ جداول مقایسه‌ای

جدول مقایسه انواع دستورات غیرمعمول

جدول ۱ انواع مختلف دستورات غیرمعمول و تأثیر آن‌ها بر دیس‌اسمبلرها را مقایسه می‌کند.

جدول ۱.۱: مقایسه انواع دستورات غیرمعمول در Anti-Disassembly

نوع دستور	مثال‌ها	تأثیر بر دیس‌اسمبلر
دستورات شرطی پیچیده	JZ, JNZ, JC با شرایط پیچیده	شکستن تحلیل جریان کنترل
دستورات FPU	FCOMIP, FCHS, FLDZ	ایجاد خطا در تحلیل پشته FPU
دستورات سیستم	IN, OUT, SGDT	مشکل در شبیه‌سازی محیط
دستورات رشته‌ای	REP MOVSB, SCASB	پیچیدگی در تحلیل حلقه‌ها
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 تکنیک‌های مختلف مبتنی بر دستورات غیرمعمول را نشان می‌دهد.

جدول ۲.۱: مقایسه میزان اثرگذاری تکنیک‌های 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 رشته‌ای پیچیده	متوسط	کم	بالا
دستورات حالت وابسته	بالا	متوسط	بالا
ترکیب چندین تکنیک	بسیار بالا	بالا	بسیار بالا

۴.۱ پنهان‌سازی از طریق تجزیه و تحلیل پویا

پنهان‌سازی از طریق تجزیه و تحلیل پویا یکی از پیشرفته‌ترین و کارآمدترین شیوه‌های مقابله با مهندسی معکوس است؛ زیرا بخش‌هایی از منطق برنامه تنها در زمان اجرا و تحت شرایط مشخص قابل مشاهده یا تحلیل هستند. این ویژگی باعث می‌شود ابزارهای تحلیل استاتیک نتوانند ساختار واقعی برنامه را بازسازی کنند و تحلیل‌گر مجبور به استفاده از روش‌های پویا گردد [۱].

یکی از مهم‌ترین تکنیک‌ها در این حوزه، رمزگشایی زمان اجرا (Runtime Decryption) است. در این روش بخش‌های حساس برنامه به‌صورت رمزگذاری‌شده ذخیره شده و تنها هنگام نیاز رمزگشایی می‌شوند. پس از اجرای تابع نیز این بخش‌ها مجدداً پاک شده یا دوباره رمزگذاری می‌شوند، که این موضوع دسترسی تحلیل‌گر به نسخهٔ کامل و پایدار کد را دشوار می‌کند [۲]. این روش در بدافزارهای پیچیده و نرم‌افزارهای تجاری با حساسیت بالا بسیار رایج است.

روش دیگری که در پنهان‌سازی پویا کاربرد گسترده دارد، بارگذاری پویا (Dynamic Code Loading) است. در این تکنیک، بخش‌هایی از برنامه در فایل اجرایی وجود ندارند و در زمان اجرا از حافظهٔ رمزگذاری‌شده، منابع بیرونی یا حتی شبکه بارگذاری می‌شوند. این ویژگی باعث بی‌اثر شدن بخش قابل توجهی از تحلیل‌های استاتیک می‌شود و تحلیل‌گر را مجبور به تحلیل پویا و ثبت رفتار اجرای زنده می‌کند [۳].

از سوی دیگر، تکنیک‌های 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، زمان‌بندی اجرای دستورها یا وضعیت سخت‌افزار، فرآیند تحلیل را برای مهاجم پیچیده می‌کنند. این وابستگی باعث می‌شود مسیر واقعی اجرا تنها در بستر واقعی قابل مشاهده باشد و ابزارهای تحلیلی قادر به بازسازی آن نباشند [۱].

بخش مهمی از پنهان‌سازی پویا به‌کارگیری تکنیک‌های ضد دیباگ پویا (Dynamic Anti-Debugging) است. روش‌هایی مانند اندازه‌گیری تأخیر اجرای دستورها، بررسی Breakpointها، شناسایی Emulator، یا فراخوانی توابع خاص سیستم‌عامل جهت تشخیص شرایط غیرعادی اجرا—همگی برای جلوگیری از تحلیل کد توسط ابزارهای دیباگ استفاده می‌شوند. در صورت تشخیص چنین شرایطی، برنامه می‌تواند منطق اصلی خود را پنهان کرده یا مسیر اجرا را تغییر دهد و تحلیل‌گر را گمراه کند [۴].

با وجود پتانسیل بالا، پنهان‌سازی پویا محدودیت‌هایی نیز دارد. ابزارهای تحلیل پیشرفته با استفاده از تکنیک‌هایی مانند حافظه‌برداری (Memory Dumping) و ابزارهای درج پویا (Dynamic Instrumentation) می‌توانند نسخه‌هایی از کد رمزگشایی‌شده در حافظه را استخراج کنند [۵]. با این حال، بهره‌گیری هم‌زمان از چندین روش پویا و ایستا همچنان یکی از مقاوم‌ترین راهبردهای مقابله با مهندسی معکوس محسوب می‌شود.

۵.۱ معرفی کدهای خودتغییر (Self-Modifying Code)

کدهای خودتغییر به دسته‌ای از برنامه‌ها اشاره دارند که در زمان اجرا ساختار اجرایی خود را بازنویسی می‌کنند. در معماری رایج سامانه‌های امروزی—که دستور و داده هر دو در یک فضای حافظه ذخیره می‌شوند—برنامه می‌تواند همانند داده‌ها به ناحیهٔ کد دسترسی پیدا کند و بخش‌هایی از آن را تغییر دهد. این ویژگی امکان آن را فراهم می‌کند که نسخهٔ واقعیِ اجرا شده با نسخه‌ای که تحلیل‌گر در فایل دودویی مشاهده می‌کند یکسان نباشد. همین فاصله میان «کد ثابت» و «کد اجرایی» یکی از مهم‌ترین چالش‌ها در حوزهٔ مهندسی معکوس و تحلیل استاتیک به‌شمار می‌رود [۶].

در این الگو، برنامه هنگام اجرا مجموعه‌ای از نوشتن‌ها (self-writes) بر روی ناحیهٔ کد انجام می‌دهد. این نوشتن‌ها می‌توانند به‌صورت ساده تنها یک دستور را با نسخهٔ معادل اما متفاوت جایگزین کنند، یا در قالب الگوهای پیشرفته‌تر، مسیرهای کنترلی و بلاک‌های اجرایی جدیدی را ایجاد کنند که فقط در حالت خاصی از اجرای برنامه فعال می‌شوند. به همین دلیل، کد خودتغییر الزاماً به‌معنای تغییر «منطق برنامه» نیست؛ بلکه بیشتر ابزاری برای پنهان‌سازی ساختار واقعی کد و پیچیده‌کردن تحلیل

است. رفتار نهایی برنامه ثابت می‌ماند، اما شکل داخلی آن در هر اجرا می‌تواند متفاوت باشد [۶].

از دید فنی، مهم‌ترین پیامد وجود SMC این است که بسیاری از دیس‌اسمبلرها و ابزارهای تحلیل استاتیک بر فرض «ثابت بودن کد» تکیه دارند. هنگامی که این فرض نقض می‌شود، ابزار قادر نخواهد بود ساختار واقعی جریان کنترل را به‌درستی بازیابی کند. مطالعات موجود نشان می‌دهد که این تغییرپذیری باعث می‌شود نمایش‌های معمولی مانند Control Flow Graph نتوانند وضعیت واقعی برنامه را پوشش دهند، زیرا هر بازنویسی در زمان اجرا می‌تواند نسخهٔ جدیدی از یک بلاک اجرایی ایجاد کند و مسیرهای کنترلی متفاوتی را شکل دهد [۶].

بنابراین، Self-Modifying Code نه‌تنها یک تکنیک مخفی‌سازی است، بلکه مسئله‌ای بنیادین برای تحلیل‌گر ایجاد می‌کند: کدی که باید تحلیل شود، در فایل حضور ندارد و تنها در زمان اجرا به‌وجود می‌آید. این ویژگی، SMC را به یکی از پرکاربردترین سازوکارهای ضد دیس‌اسمبلی در بدافزارها، محافظ‌های نرم‌افزاری و ابزارهای ضد مهندسی معکوس تبدیل کرده است [۶].

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

فرض کن برنامهٔ زیر قرار است در نهایت مقدار 10 را در رجیستر EAX قرار دهد، اما نویسندهٔ برنامه می‌خواهد مسیر اصلی کد مخفی بماند؛ بنابراین از SMC استفاده می‌کند.

مرحلهٔ ۱: کد اولیه (نسخهٔ قابل‌مشاهده در فایل)

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

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

برای تحلیلگر، این یک کد کاملاً معمولی است.

مرحلهٔ ۲: بخش خودتغییر (self-write)

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

```
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 نمی‌دهد). یعنی برنامه قبل از اجرا دستور اصلی را پاک کرده است.

مرحلهٔ ۳: نسخهٔ واقعی که اجرا می‌شود

وقتی CPU به خط مربوط به ADD می‌رسد، دیگر این دستور وجود ندارد. نسخهٔ دیده‌شده توسط تحلیلگر:

```
ADD EAX, EBX
```

اما نسخهٔ واقعی در حافظه تبدیل شده به:

```
NOP
NOP
NOP
```

مقدار نهایی:

```
EAX = 5
```

که برابر ۱۰ نیست.

۲.۵.۱ معایب مثال سادهٔ SMC

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

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

در این مثال، ساختار کد به‌گونه‌ای طراحی شده است که **دستور واقعی** در زمان اجرا و بر اساس مقدار زمان تولید می‌شود [۶].

مرحلهٔ 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
```

نسخه‌های رمز شده‌اند .

مرحلهٔ ۲: انتخاب مسیر

در زمان اجرا مسیر بر اساس ثانیه انتخاب می‌شود.

مرحلهٔ 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
```

مرحلهٔ ۴: اجرای نسخهٔ تولیدشده

```
MOV EAX, 100
ADD EAX, 20
RET
```

۶.۱ مقاومت در برابر دیس اسمبلی با استفاده از تکنیک‌های رمزگذاری

با تبدیل فایل اجرایی به کد اسمبلی، تحلیل‌گران می‌توانند منطق داخلی، باگ‌ها و بدافزارهای احتمالی را شناسایی کنند[۷]. به‌عبارت دیگر، دیس‌اسمبلی امکان افشای عملکرد برنامه و پیدا کردن آسیب‌پذیری‌ها یا کدهای مخرب را فراهم می‌کند. به همین دلیل، نویسندگان نرم‌افزارهای محافظت‌شده تلاش می‌کنند با روش‌هایی مانند رمزگذاری و ابهام‌زایی، دیس‌اسمبلی را دشوار کنند.

۱.۶.۱ کد پلی‌مورفیک (Polymorphic Code)

تعریف و مکانیسم:

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

تصادفی رمزگذاری می‌شود و یک بخش کوتاه کوچک در ابتدای برنامه (بخش رمزگشا) وجود دارد که با کلید مربوطه، payload را در حافظه رمزگشایی می‌کند[۸، ۹]. به‌عنوان مثال، ویروس‌های پلی‌مورفیک مشهور، از رمزنگاری XOR با یک کلید تصادفی برای مخفی کردن بدنه‌ی اصلی خود استفاده می‌کنند، به‌طوری که هر نمونه از لحاظ باینری کاملاً با نمونه‌های دیگر متفاوت است[۹].

مزایا و کاربردها:

مزیت اصلی پلی‌مورفیسم مقاومت در برابر شناسایی مبتنی بر امضاست؛ چون هر بار که کد اجرا می‌شود، الگویی یکتا ایجاد می‌کند و امضای ثابتی ندارد[۹]. این روش عمدتاً در ویروس‌ها، کرم‌ها و بدافزارها به کار می‌رود تا برنامه‌های ضدویروس را فریب دهد. همچنین در برخی محافظت‌کننده‌های تجاری کد (پکرها) دیده می‌شود که با تولید کلیدهای مختلف یا تغییر الگوریتم رمزگشایی، کد را در هر توزیع متفاوت می‌کنند. پلی‌مورفیسم نسبتاً ساده است و تأثیر زیادی بر عملکرد اجرایی برنامه ندارد.

نحوه عملکرد:

معمولاً هر نمونه پلی‌مورفیک شامل دو بخش است: یک لایه رمزنگاری (برای پنهان‌سازی payload) و یک بخش رمزگشای کوتاه. در زمان اجرا، ابتدا بخش رمزگشا فعال شده و بدنه اصلی را در حافظه بازمی‌کند، سپس کنترل به کد اصلی داده می‌شود[۸]. به خاطر این ساختار، دیس‌اسمبلی ایستا مؤثر نیست زیرا کد اصلی در فایل رمزیافته و فقط با اجرای برنامه آشکار می‌شود[۸].

معایب و روش‌های تحلیل:

گرچه پلی‌مورفیسم شناسایی ساده مبتنی بر امضا را دشوار می‌کند، اما شکستن آن آسان است. تحلیل‌گران می‌توانند برنامه را اجرا کنند و با قرار دادن نقطه‌نگاری 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، می‌توان با نوشتن اسکریپت‌های کوچک یا استفاده از ابزارهای خودکار، کد را رمزگشایی کرد. به همین علت، اکثر روش‌های مقابله با کد پلی‌مورفیک بر تحلیل پویا (اجرا در محیط کنترل‌شده و خاموش کردن رمزنگاری) و یا نرمال‌سازی کد بر مبنای الگوریتم‌های آماری متمرکزند.

۲.۶.۱ کد متامورفیک (Metamorphic Code)

تعریف و مکانیسم:

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

تصادفی رمزگذاری می‌شود و یک بخش کوتاه کوچک در ابتدای برنامه (بخش رمزگشا) وجود دارد که با کلید مربوطه، payload را در حافظه رمزگشایی می‌کند[۹][۸]. به‌عنوان مثال، ویروس‌های پلی‌مورفیک مشهور، از رمزنگاری XOR با یک کلید تصادفی برای مخفی کردن بدنه‌ی اصلی خود استفاده می‌کنند، به‌طوری که هر نمونه از لحاظ باینری کاملاً با نمونه‌های دیگر متفاوت است[۹].

مزایا و کاربردها:

مزیت اصلی پلی‌مورفیسم مقاومت در برابر شناسایی مبتنی بر امضاست؛ چون هر بار که کد اجرا می‌شود، الگویی یکتا ایجاد می‌کند و امضای ثابتی ندارد[۹]. این روش عمدتاً در ویروس‌ها، کرم‌ها و بدافزارها به کار می‌رود تا برنامه‌های ضدویروس را فریب دهد. همچنین در برخی محافظت‌کننده‌های تجاری کد (پکرها) دیده می‌شود که با تولید کلیدهای مختلف یا تغییر الگوریتم رمزگشایی، کد را در هر توزیع متفاوت می‌کنند. پلی‌مورفیسم نسبتاً ساده است و تأثیر زیادی بر عملکرد اجرایی برنامه ندارد.

معایب و روش‌های تحلیل:

گرچه پلی‌مورفیسم شناسایی ساده مبتنی بر امضا را دشوار می‌کند، اما شکستن آن آسان است. تحلیل‌گران می‌توانند برنامه را اجرا کنند و با قرار دادن نقطه‌نگاری 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، می‌توان با نوشتن اسکریپت‌های کوچک یا استفاده از ابزارهای خودکار، کد را رمزگشایی کرد. به همین علت، اکثر روش‌های مقابله با کد پلی‌مورفیک بر تحلیل پویا (اجرا در محیط کنترل‌شده و خاموش کردن رمزنگاری) و یا نرمال‌سازی کد بر مبنای الگوریتم‌های آماری متمرکزند.

۳.۶.۱ بسته‌سازی کد (Packers)

تعریف و مکانیسم:

کد پلی‌مورفیک نوعی کد خودتغییرده است که در هر نسخه ظاهر باینری متفاوتی دارد، ولی منطق و الگوریتم اصلی‌اش حفظ می‌شود[۸]. معمولاً در این روش، بخش اصلی کد («payload») با یک کلید تصادفی رمزگذاری می‌شود و یک بخش کوتاه کوچک در ابتدای برنامه (بخش رمزگشا) وجود دارد که با کلید مربوطه، payload را در حافظه رمزگشایی می‌کند[۹][۸]. به‌عنوان مثال، ویروس‌های پلی‌مورفیک مشهور، از رمزنگاری XOR با یک کلید تصادفی برای مخفی کردن بدنه‌ی اصلی خود استفاده می‌کنند، به‌طوری که هر نمونه از لحاظ باینری کاملاً با نمونه‌های دیگر متفاوت است[۹].

مزایا و کاربردها:

مزیت اصلی پلی‌مورفیسم مقاومت در برابر شناسایی مبتنی بر امضاست؛ چون هر بار که کد اجرا می‌شود، الگویی یکتا ایجاد می‌کند و امضای ثابتی ندارد[۹]. این روش عمدتاً در ویروس‌ها، کرم‌ها و بدافزارها به کار می‌رود تا برنامه‌های ضدویروس را فریب دهد. همچنین در برخی محافظت‌کننده‌های تجاری کد (پکرها) دیده می‌شود که با تولید کلیدهای مختلف یا تغییر الگوریتم رمزگشایی، کد را در هر توزیع متفاوت می‌کنند. پلی‌مورفیسم نسبتاً ساده است و تأثیر زیادی بر عملکرد اجرایی برنامه ندارد.

معایب و روش‌های تحلیل:

گرچه پلی‌مورفیسم شناسایی ساده مبتنی بر امضا را دشوار می‌کند، اما شکستن آن آسان است. تحلیل‌گران می‌توانند برنامه را اجرا کنند و با قرار دادن نقطه‌نگاری 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، می‌توان با نوشتن اسکریپت‌های کوچک یا استفاده از ابزارهای خودکار، کد را رمزگشایی کرد. به همین علت، اکثر روش‌های مقابله با کد پلی‌مورفیک بر تحلیل پویا (اجرا در محیط کنترل‌شده و خاموش کردن رمزنگاری) و یا نرمال‌سازی کد بر مبنای الگوریتم‌های آماری متمرکزند.

۴.۶.۱ ابهام‌زایی مبتنی بر مجازی‌سازی (Virtualization-Based Obfuscation)

تعریف و مکانیسم:

کد پلی‌مورفیک نوعی کد خودتغییرده است که در هر نسخه ظاهر باینری متفاوتی دارد، ولی منطق و الگوریتم اصلی‌اش حفظ می‌شود[۸]. معمولاً در این روش، بخش اصلی کد («payload») با یک کلید تصادفی رمزگذاری می‌شود و یک بخش کوتاه کوچک در ابتدای برنامه (بخش رمزگشا) وجود دارد که با کلید مربوطه، payload را در حافظه رمزگشایی می‌کند[۹][۸]. به‌عنوان مثال، ویروس‌های پلی‌مورفیک مشهور، از رمزنگاری XOR با یک کلید تصادفی برای مخفی کردن بدنه‌ی اصلی خود استفاده می‌کنند، به‌طوری که هر نمونه از لحاظ باینری کاملاً با نمونه‌های دیگر متفاوت است[۹].

مزایا و کاربردها:

مزیت اصلی پلی‌مورفیسم مقاومت در برابر شناسایی مبتنی بر امضاست؛ چون هر بار که کد اجرا می‌شود، الگویی یکتا ایجاد می‌کند و امضای ثابتی ندارد[۹]. این روش عمدتاً در ویروس‌ها، کرم‌ها و بدافزارها به کار می‌رود تا برنامه‌های ضدویروس را فریب دهد. همچنین در برخی محافظت‌کننده‌های تجاری کد (پکرها) دیده می‌شود که با تولید کلیدهای مختلف یا تغییر الگوریتم رمزگشایی، کد را در هر توزیع متفاوت می‌کنند. پلی‌مورفیسم نسبتاً ساده است و تأثیر زیادی بر عملکرد اجرایی برنامه ندارد.

معایب و روش‌های تحلیل:

گرچه پلی‌مورفیسم شناسایی ساده مبتنی بر امضا را دشوار می‌کند، اما شکستن آن آسان است. تحلیل‌گران می‌توانند برنامه را اجرا کنند و با قرار دادن نقطه‌نگاری 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، می‌توان با نوشتن اسکریپت‌های کوچک یا استفاده از ابزارهای خودکار، کد را رمزگشایی کرد. به همین علت، اکثر روش‌های مقابله با کد پلی‌مورفیک بر تحلیل پویا (اجرا در محیط کنترل‌شده و خاموش کردن رمزنگاری) و یا نرمال‌سازی کد بر مبنای الگوریتم‌های آماری متمرکزند.

۵.۶.۱ رمزگذاری کد در زمان اجرا (Runtime Code Encryption)

تعریف و مکانیسم:

کد پلی‌مورفیک نوعی کد خودتغییرده است که در هر نسخه ظاهر باینری متفاوتی دارد، ولی منطق و الگوریتم اصلی‌اش حفظ می‌شود[۸]. معمولاً در این روش، بخش اصلی کد («payload») با یک کلید تصادفی رمزگذاری می‌شود و یک بخش کوتاه کوچک در ابتدای برنامه (بخش رمزگشا) وجود دارد که با کلید مربوطه، payload را در حافظه رمزگشایی می‌کند[۸، ۹]. به‌عنوان مثال، ویروس‌های پلی‌مورفیک مشهور، از رمزنگاری XOR با یک کلید تصادفی برای مخفی کردن بدنه‌ی اصلی خود استفاده می‌کنند، به‌طوری که هر نمونه از لحاظ باینری کاملاً با نمونه‌های دیگر متفاوت است[۹].

مزایا و کاربردها:

مزیت اصلی پلی‌مورفیسم مقاومت در برابر شناسایی مبتنی بر امضاست؛ چون هر بار که کد اجرا می‌شود، الگویی یکتا ایجاد می‌کند و امضای ثابتی ندارد[۹]. این روش عمدتاً در ویروس‌ها، کرم‌ها و بدافزارها به کار می‌رود تا برنامه‌های ضدویروس را فریب دهد. همچنین در برخی محافظت‌کننده‌های تجاری کد (پکرها) دیده می‌شود که با تولید کلیدهای مختلف یا تغییر الگوریتم رمزگشایی، کد را در هر توزیع متفاوت می‌کنند. پلی‌مورفیسم نسبتاً ساده است و تأثیر زیادی بر عملکرد اجرایی برنامه ندارد.

معایب و روش‌های تحلیل:

گرچه پلی‌مورفیسم شناسایی ساده مبتنی بر امضا را دشوار می‌کند، اما شکستن آن آسان است. تحلیل‌گران می‌توانند برنامه را اجرا کنند و با قرار دادن نقطه‌نگاری 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، می‌توان با نوشتن اسکریپت‌های کوچک یا استفاده از ابزارهای

خودکار، کد را رمزگشایی کرد. به همین علت، اکثر روش‌های مقابله با کد پلی‌مورفیک بر تحلیل پویا (اجرا در محیط کنترل‌شده و خاموش کردن رمزنگاری) و یا نرمال‌سازی کد بر مبنای الگوریتم‌های 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 (Anti-Disassembly (AD:

Anti-Disassembly مجموعه‌ای از ترفندها و الگوهای برنامه‌نویسی است که عمداً 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 به طور 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 اشکال‌زداها (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 لحظهٔ اجرای دستورات حیاتی مشاهده کند.
- ردیابی دستورات (Instruction Tracing) در این روش توالی دقیق دستوراتی که در طول اجرای بخش مبهم‌سازی‌شده اجرا می‌شوند ثبت می‌گردد. این ردیابی می‌تواند آدرس‌های پرش‌های داینامیک 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 لحظهٔ پرش کشف می‌کند و آدرس مقصد واقعی را به‌صورت 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 داینامیک به پرش استاتیک پس از کشف آدرس مقصد پرش داینامیک، 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 اُبفوسکیشن‌های 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 ابزارهایی 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 اسمبلر ها

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 (شبیه 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** Anti-Disassembly (AADA) 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 رمزگذاری‌شده (Decryption 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 فشرده‌سازی 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 آنپک 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 پردازه 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 داینامیک

شبیه‌سازی و اجرای داینامیک 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 داینامیک تمام دستورالعمل‌ها 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 داینامیک 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 داینامیک 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 جداشده و ایزوله اجرا می‌گردد.

علاوه بر این، شبیه‌سازی داینامیک 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 داینامیک برای تحلیل کدهای اجرایی

یکی از مهم‌ترین کاربردهای شبیه‌سازی داینامیک، 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 داینامیک 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 داینامیک 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 داینامیک 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. نه‌تنها 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 به‌سرعت نقاطی از حافظه که در حال بازنویسی هستند را مشخص کند. ابزارهایی مانند QEMU، PANDA یا Intel PIN در این حوزه بسیار کاربرد دارند.

Dynamic Unpacking و Deobfuscation پویا از دیگر تکنیک‌های مهم محسوب می‌شود. بسیاری از کدهای خودتغییر ابتدا نسخه‌ی رمزگذاری‌شده یا فشرده‌ی خود را اجرا می‌کنند و در زمان اجرا نسخه‌ی واقعی را تولید می‌کنند. تحلیل‌گران با دنبال‌کردن 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 مانیتورکردن این بخش‌ها یا با فعال‌کردن 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 دیس اسمبلی

با پیشرفت گسترده ابزارهای تحلیل معکوس، تکنیک‌های کلاسیک ضد دیس‌اسمبلی دیگر به‌تنهایی نمی‌توانند امنیت کافی فراهم کنند. ابزارهای مدرنی همچون IDA Pro، Ghidra، Binary Ninja، 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 نه‌تنها 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:

- پردازه‌های 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 شبیه‌سازی‌شده 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.

منابع

[۱] Christopher Kruegel, William Robertson, and Giovanni Vigna. “Detecting Kernel-Level Rootkits Through Binary Analysis”. In: Proceedings of the USENIX Security Symposium. ۲۰۰۶.

[۲] Andreas Moser, Christopher Kruegel, and Engin Kirda. “Limits of Static Analysis for Malware Detection”. In: Proceedings of the ۲۰۰۷ Annual Computer Security Applications Conference (ACSAC). ۲۰۰۷.

[۳] Manuel Egele et al. “A Survey on Automated Dynamic Malware Analysis Techniques and Tools”. In: ACM Computing Surveys ۴۴.۲ (۲۰۰۸).

[۴] Dae Young Kwon and Seunghoon Y. Kim. “Anti-Debugging Techniques and Dynamic Analysis Evasion”. In: IEEE Security & Privacy ۱۲.۲ (۲۰۱۴), pp. ۷۷–۸۳.

[۵] Brendan Dolan-Gavitt et al. “Dynamic Binary Instrumentation: A Detailed Analysis”. In: IEEE Transactions on Software Engineering (۲۰۱۶).

[۶] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. “A Model for Self-Modifying Code”. In: Information Hiding, ۸th International Workshop, IH ۲۰۰۶, Revised Selected Papers. Vol. ۴۴۳۷. Lecture Notes in Computer Science. Springer, ۲۰۰۶, pp. ۲۳۲–۲۴۸. DOI: 10.1007/978-3-540-74124-4_16.

[۷] Twingate Team. What is Disassembly? ۲۰۲۴. URL: https://www.twingate.com/blog/glossary/disassembly.

[۸] Liviu PETREAN. POLYMORPHIC AND METAMORPHIC CODE APPLICATIONS IN PORTABLE EXECUTABLE FILES PROTECTION. ۲۰۱۰. URL: https://users.utcluj.ro/~atn/papers/ATN%5C_1%5C_2010%5C_1.pdf.

[۹] What is Polymorphic Malware? URL: https://www.threatdown.com/glossary/what-is-polymorphic-malware/.

[۱۰] fiveable. Anti-reverse engineering. ۲۰۲۵. URL: https://fiveable.me/network-security-and-forensics/unit-4/anti-reverse-engineering/study-guide/2yna2P93Bt4M4uSd.

[۱۱] paloaltonetworks. Why You Need Static Analysis, Dynamic Analysis, and Machine Learning? ۲۰۲۴. URL: https://www.paloaltonetworks.com/cyberpedia/why-you-need-static-analysis-dynamic-analysis-machine-learning.

[۱۲] Tho Le. Malware Analysis Lab and Behavioral Analysis Steps. ۲۰۲۱. URL: https://tho-le.medium.com/malware-analysis-lab-and-behavioral-analysis-steps-229cf8c15d5e.

[۱۳] Neil Fox. How to Unpack Malware with x۶۴dbg. ۲۰۲۵. URL: https://www.varonis.com/blog/x64dbg-unpack-malware.

[۱۴] Tony K Rodgers. Anti-Reverse Engineering. ۲۰۲۵. URL: https://medium.com/@Tony_K_Rodgers/anti-reverse-engineering-e6adcf1ded04.

[۱۵] apriorit. Anti Debugging Protection Techniques with Examples. ۲۰۲۴. URL: https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software.

[۱۶] Michael Sikorski and Andrew Honig. Practical Malware Analysis. oreilly, ۲۰۱۲.

[۱۷] Omer Sezgin U˘gurlu. “STEALTH SANDBOX ANALYSIS OF MALWARE”. Master’s thesis. bilkent university, ۲۰۰۹.

[۱۸] Mark Vincent Yason. The Art of Unpacking. ۲۰۰۷. URL: https://blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf.

[۱۹] Nicole Fishbein. Beginner’s guide to malware analysis and reverse engineering. ۲۰۲۵. URL: https://intezer.com/beginners-guide-to-malware-analysis-and-reverse-engineering/.

[۲۰] Julien Lenoir. Implementing your own generic unpacker. ۲۰۱۵. URL: https://airbus-seclab.github.io/gunpack/HITB-GSEC-SG2015-Lenoir-Gunpack.pdf.

[۲۱] Chris Eagle. The IDA Pro Book. No Starch Press, ۲۰۲۰.

[۲۲] Bruce Dang, Alexandre Gazet, and Elias Bachaalany. Practical Reverse Engineering. Wiley, ۲۰۱۴.

[۲۳] Dennis Andriesse. Practical Binary Analysis. No Starch Press, ۲۰۱۸.

[۲۴] Manuel Egele et al. “A Survey on Automated Dynamic Malware Analysis Techniques”. In: ACM Computing Surveys (۲۰۱۲).

[۲۵] Mikel Ugarte and Juan Caballero. “On the Resilience of Obfuscation Techniques against Symbolic Execution”. In: IEEE Security and Privacy Workshops. ۲۰۱۹.