Ondanks alle sterke punten van .NET moeten de eindverantwoordelijken voor grootschalige softwareprojecten nog steeds rekening houden met de mogelijkheid van beveiligingslekken binnen het nieuwe raamwerk. Dit artikel zal wat opheldering geven over twee mogelijke beveiligingskwesties bij ontwikkeling in .NET: reverse engineering en ongewenste aanpassingen van code. Is het nodig dat u zich daar acuut zorgen over maakt? Voor het antwoord op deze vraag moeten we een kijkje achter de schermen nemen.

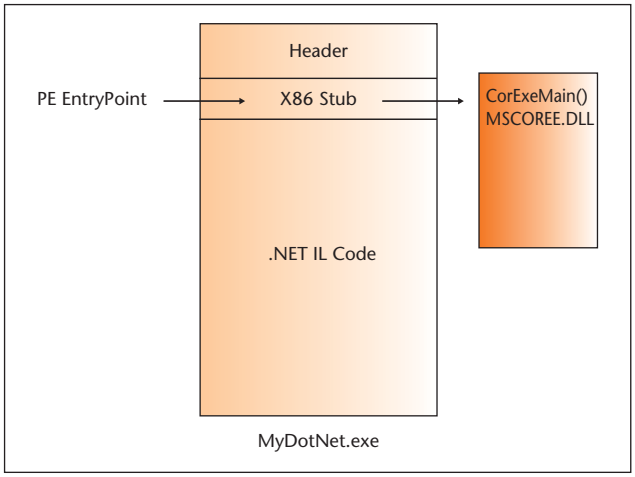
Ontwikkelen op eigen risico?

Beveiligingskwesties in .NET

In theorie zijn .NET-applicaties platformonafhankelijk. Gecompileerde programma's bevatten niet automatisch processorinstructies. In plaats daarvan genereert de compiler zogeheten 'intermediate language'(IL) code die wordt geladen en gecompileerd door de runtime-omgeving van het systeem dat de code gaat gebruiken, ofwel het common language runtime (CLR)-systeem.

HOE VEILIG IS UW CODE? Tot zover heeft dit proces alleen theoretische invloed op native code. Een diepgaander onderzoek naar de structuur van .NET-bestanden onthult veel meer, wellicht zelfs teveel. .NET komt misschien over als zwaar beveiligd, maar ontwikkelaars moeten zich bewust zijn van deze beveiligingsoverwegingen om te voorkomen dat ze door middel van code hun eigen ruiten ingooien. IL-bestanden gebruiken de welbekende PE (Portable Executable) bestandsstructuur die al wordt gebruikt door Win32-bestanden als transportmiddel. Waarschijnlijk is dit gedaan omdat Microsoft op deze manier eenvoudig een PE-conform mechanisme kon integreren. Dit activeerde automatisch de CLR in oudere versies van Windows als iemand de .NET-executable probeerde uit te voeren. Het is zelfs zo dat pure .NET-applicatiebestanden enige native Intel x86-code bevatten om ervoor te zorgen dat de CLR wordt ingeladen en de controle wordt overgedragen aan de kern-DLL (mscoree.dll).

METADATA Het ware .NET-deel binnen het bestand wordt beschreven met behulp van een 'nieuwe' tag,



AFBEELDING 1. de CLR wordt ingeladen en de controle wordt overgedragen aan de kern DLL (mscoree.dll).

bekend als MetaData. Of u het nu wilt of niet, de MetaData onthult talloze details over uw code. Net als de PE-tabellen in native Win32-applicaties bevatten deze data informatie over geëxporteerde interfaces en geïmporteerde items. Verder zijn alle namen van interne objecten, methoden, hun locaties, omvang, parameters, handtekeningen en alle strings hier te bekijken in een gerangschikte tabel. Deze MetaData zorgen ervoor dat de gebruiker geen informatie mist bij demontage van het systeem. Elke verwijzing naar andere objecten, zowel intern als extern, maakt gebruik van deze tabel.

Voor Win32-applicaties speelt het permanente

probleem dat de ontwikkelaar bij demontage vaak moet raden naar het startadres van een specifieke functie en uit moet vinden of het bewuste gebied code dan wel data bevat. Bij de analyse van .NET-applicaties is echter geen sprake van al deze onzekerheid. De MetaData bieden u alle benodigde details. Het is alsof u Win32-applicaties levert met alle debugging-informatie. U kunt zich nu gemakkelijk voorstellen wat concurrenten en crackers kunnen doen met uw software:

- Geheime, bedrijfskritische algoritmes in de software zijn te achterhalen.
- Verificatie-informatie van licentievoorwaarden is duidelijk zichtbaar.
- Alle delen van de code zijn te decompileren, aan te passen zodat ze zich anders gedragen en vervolgens opnieuw te integreren.

VERDUISTERING Hoe is dit te voorkomen? Een oude en welbekende benadering van Java is verduistering. Deze volgt twee hoofdregels:

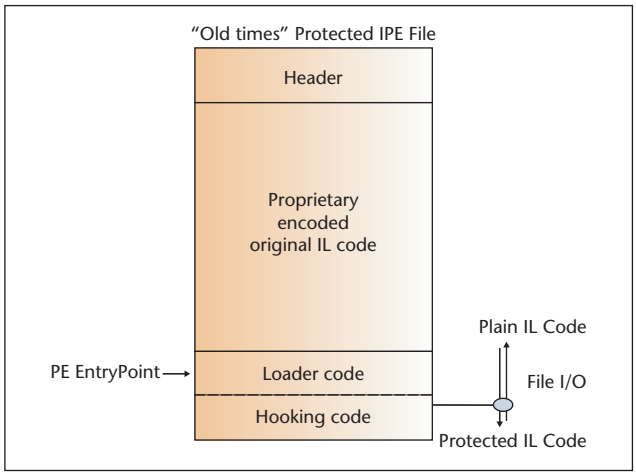
1. Computers geven niet om menselijke leesbaarheid. Nette, gemakkelijk te doorgronden object-namen zijn dus overbodig. Alle interne, niet-geëxporteerde objecten worden doorgaans voorzien van willekeurig gekozen herkenningspunten. In de meeste gevallen heeft de naam op zich geen enkele betekenis voor de uitvoering, zoals ook het geval is bij de meeste objecten.
2. Informatie die niet essentieel is voor correcte uitvoering wordt verwijderd.

Bij uitvoering geeft het besturingssysteem de controle door aan het PE entrypoint, de eigenlijke loader

Sommige verduisteraars voegen aanvullende ingrediënten toe, zoals:

- Willekeurige toevoeging van code die geen invloed heeft op het resultaat van de functie.
- Invoering van kleine fouten in de samengestelde structuur waardoor standaardtools, zoals ILDASM, incorrecte resultaten produceren of de weg kwijt raken.

CODE UITKLEDEN Al deze methoden verslechteren duidelijk de leesbaarheid van de code. Desondanks is de uitgevoerde code uiteindelijk onbeveiligd aanwezig in bestanden op de harde schijf. Alle informatie voor veilige demontage van de code met meer verfijnde tools zijn nog steeds aanwezig. Er zijn geen obstakels aanwezig om aanpassingen te voorkomen. Crackers hebben al bij Win32-applicaties bewezen dat de leesbaarheid van



AFBEELDING 2. Wanneer de code aanwezig was in het geheugen werd de loader compleet losgekoppeld van de applicatie.

variabelen geen belemmering is om toegang te krijgen tot de applicatiecode en deze aan te passen. Het is dan ook duidelijk dat geraffineerdere maatregelen nodig zijn om crack-pogingen succesvol te blokkeren.

De gangbare benadering is een erfenis van de vroegere Win32-beschermtools. Dit waren 'plain loaders'. Het originele applicatiebestand werd gebundeld in een vergrendeld formaat met een loader die wist hoe de applicatie te reconstrueren was. Bij uitvoering kreeg de loader eerst de controle en werd de originele applicatie in het geheugen geladen of werd een filter geïnstalleerd dat actief werd wanneer het besturingssysteem de applicatie aansprak zodat het gedecodeerde fragment leesbaar werd gepresenteerd. Wanneer de code aanwezig was in het geheugen werd de loader compleet losgekoppeld van de applicatie.

Bij .NET-applicaties is dit schema alleen van toepassing op uitvoerbare bestanden. Het is niet toepasbaar op compilaties. Tools die gebruik maken van deze techniek benutten het feit dat de .NET CLR aan te roepen is door de eerdergenoemde x86 code die aanwezig is in alle .NET executables. Zo wordt de originele .NET executable getransformeerd tot een regulier ogend Win32 PE-bestand. Bij uitvoering geeft het besturingssysteem de controle door aan het PE entry point, de eigenlijke loader. Deze installeert filters om aan dit proces gerelateerde file I/O's te onderscheppen en geeft het stokje uiteindelijk door aan de mscoree.dll. Als de CLR het image-bestand van de schijf haalt, onderscheppen de filters van de loader de datastroom. De loader presenteert het originele, gedecodeerde imagebestand aan de CLR en de applicatie zal normaal opstarten.

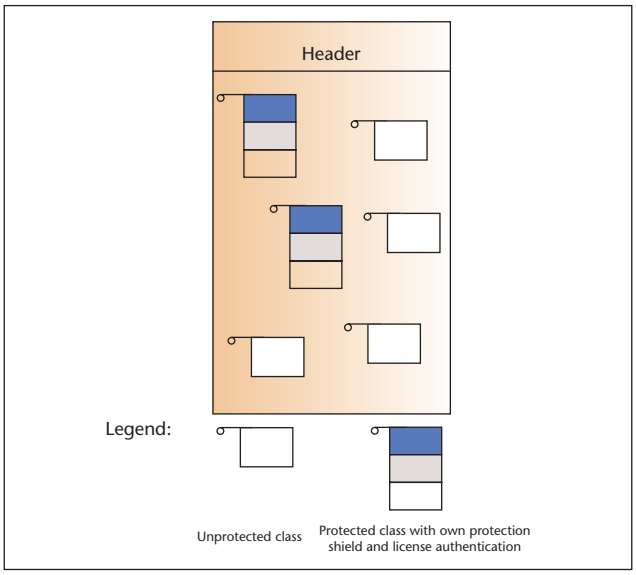
GECOMPILEERD De kracht van deze benadering is dat de IL-code nooit onbeschermd wordt gelaten op de

schijf, maar alleen aanwezig is in het tijdelijke geheugen tussen het laden en de eerste uitvoering. Bij de eerste uitvoering wordt de code 'just-in-time' (JIT) gecompileerd en aangepast. Als het formaat waarin de IL-code wordt opgeslagen op een schijf niet bekend is, zou de cracker het decoderingsproces volledig moeten 'reverse engineeren' om in staat te zijn een deel van de applicatie op noemenswaardige wijze aan te passen. De grootste nadelen hiervan zijn dat deze methode alleen is toe te passen op uitvoerbare bestanden en niet op compilaties, aangezien de x86 code zijn opstartfunctie niet kan vervullen. Als een cracker erin slaagt de aan het CLR gepresenteerde data te achterhalen, beschikt hij direct over het zuivere en onbeschermde origineel van het .NET-bestand. Bedrijven die beschermtools ontwikkelen volgens deze benadering moeten dan ook krachtige anti-debug maatregelen implementeren om een dergelijke aanval onmogelijk te maken.

De derde benadering van het probleem verschilt aanzienlijk van de eerdere. Deze methode is uitgevonden om de grootste zwakheden van verduistering en de loader-benadering op te lossen. Bij verduistering blijft het IL-codesysteem leesbaar. Loaders werken alleen bij uitvoerbare bestanden, zijn strikt gebonden aan de runtime omgeving en zijn niet geïntegreerd in het uitvoerbare deel van de applicatie.

CLASS LEVEL Werken op class level is dé oplossing. U kunt kiezen welke classes de individuele bescherming ontvangen die volledig zijn geïntegreerd in het gebruikelijke uitvoeringsproces (de execution flow, vergelijkbaar met de JIT-compilatie). Dit wordt een integraal deel van de concretisering van de class. Daardoor is elke class individueel te beveiligen.

De op schijf opgeslagen data van deze class zijn daardoor onbruikbaar zonder zijn eigen beveiligingsmechanisme. Dit beveiligingsmechanisme is onafhankelijk, waardoor het ook toe te passen is op .NET-compilaties. Het is zelfs aan te passen voor 'Mono'. In tegenstelling tot de loader-methode vergt deze technologie gedetailleerde kennis van de class-structuur en IL-code waarop de beveiliging van toepassing is. In vergelijking met verduistering is het voornaamste verschil dat een compleet nieuw .NET-bestand wordt gegenereerd dat in staat is tot hosting van zowel beschermde als onbeschermde classes. De mogelijkheid om de individuele class te selecteren houdt de laadtijden laag en maakt het automatische toevoegen van code mogelijk op hetzelfde niveau (Voor bijvoorbeeld bescherming van verschillende classes met afwijkende licentievoorwaarden, wat een ongekende flexibiliteit biedt voor licentiebeheer van software).



AFBEELDING 3. Elke class is individueel te beveiligen.

Moderne tools voor verduistering kunnen reverse engineering stukken moeilijker maken

BEVEILIGING VAN APPLICATIES Ontwikkelen voor .NET brengt nieuwe uitdagingen, met name bij het beschermen van uw applicatie tegen reverse engineering en illegaal gebruik door cracking. Moderne tools voor verduistering kunnen reverse engineering stukken moeilijker maken. Om de beveiliging tegen crackers te verbeteren moet u zorgvuldig analyseren wat u nodig heeft. Als u geen compilatiebescherming nodig hebt, of licensering op class-niveau, kan een eenvoudige loader al voldoende zijn. Maar als u op zoek bent naar volledige flexibiliteit is de class level-methode de beste keuze.

Michael Zunke, Aladdin Software DRM CTO.