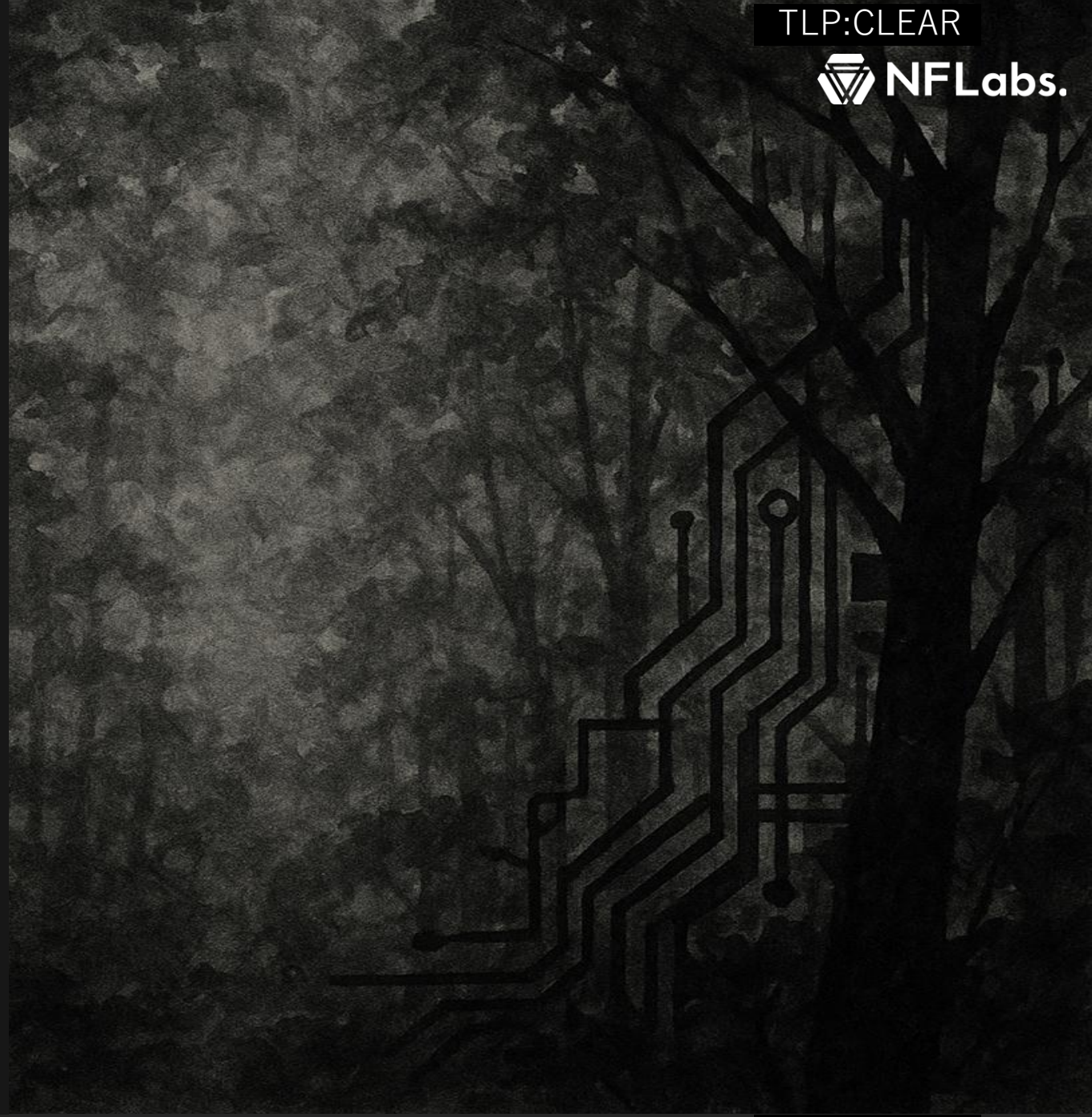
NFLabs.

Re:birth the fidb: Reverse Engineering the .NET AOT Malware

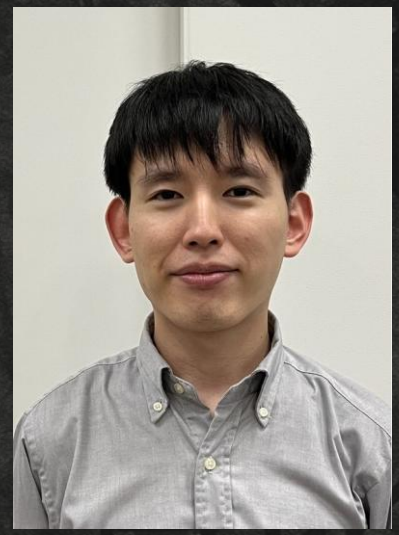


NFLabs.

About us

株式会社エヌ・エフ・ラボラトリーズ
リバースエンジニアリング、
マルウェア解析講師

株式会社エヌ・エフ・ラボラトリーズ
マルウェア解析講師



株式会社エヌ・エフ・ラボラトリーズ
オフェンシブ科目講師

NFLabs.

Key takeaway

- .NETのNative AOTでコンパイルされたプログラムの内部構造に関する基礎知識
- 静的解析を効率化するためのシグネチャの作成方法
- シグネチャやプラグインを活用し、膨大なコードから読むべきコードを絞り込む(トリアージ)手法

NFLabs.

本日の流れ

- 前半
  - .NETのマルウェア解析の基礎(Managed vs Native AOT)
  - GhidraやIDA Proを用いたシグネチャの作成とライブラリ関数の識別
- (休憩 10分)
- 後半
  - プラグインを使用してオブジェクトメタデータの復元

NFLabs.

ちょっとだけ質問です ✋

- マルウェア解析の経験がある方（趣味・実務問わず）
- CTF等でリバースエンジニアリング(Rev)問題をやったことのある人
- .NETの解析に触れたことのある方
- 解析自体が今回初めてという方
  - 経験のある方は、周囲にいる初学者の方をサポートしていただけると幸いです！

NFLabs.

Re:birth the fidb:
Reverse Engineering the .NET AOT Malware
section #1

NFLabs.

最近の事例

Native AOTでコンパイルされたマルウェアの事例が
少しずつ確認されるようになっている

IBM X-Force Threat Analysis: QuirkyLoader - A new malware loader delivering infostealers and RATs | IBM

Tags
Security Compute and servers Network

IBM X-Force Threat Analysis: QuirkyLoader - A new malware loader delivering infostealers and RATs

Booking.com Phishing Campaign Targeting Hotels and Customers - Sekoia.io Blog

THE STING OF FAKE KLING: FACEBOOK MALVERTISING LURES VICTIMS TO FAKE AI GENERATION WEBSITE

May 20, 2025

Research by: Jaromír Horejší (@JaromirHorejsi)

Impersonated GenAI Site Lures Victims to Infostealer Download - Check Point Research

最近の事例

Native AOTでコンパイルされたバイナリはステルス性が高く、解析が難しい

Users expecting to preview their generated video instead unknowingly launched a loader. In several instances, this executable utilized .NET Native AOT **(Ahead-Of-Time) Compilation** to complicate analysis and evade many traditional detection techniques. Once

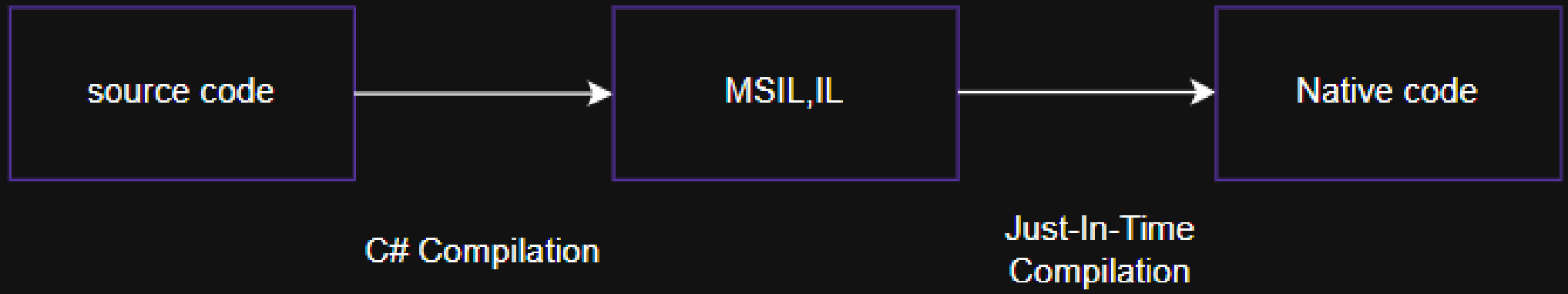
Impersonated GenAI Site Lures Victims to Infostealer Download - Check Point Research

NFLabs.

そもそも.NETのマルウェアはどうやって解析するの？

- .NET(C#言語など)で作成されたマルウェアは、コンパイルの方法によって解析のしやすさが変わる
- デフォルトでは、IL(中間言語)が含まれるため、dnSpy/dnSpyExやILSpyなどを使ってデコンパイルすることで、高級言語のレベルで解析することができる

NFLabs.



NFLabs.

```
IL_0000: nop
IL_0001: ldstr "Hello, World!"
IL_0006: call void [mscorlib]System.Console::WriteLine(string)
IL_000b: nop
IL_000c: ret
```

NFLabs.

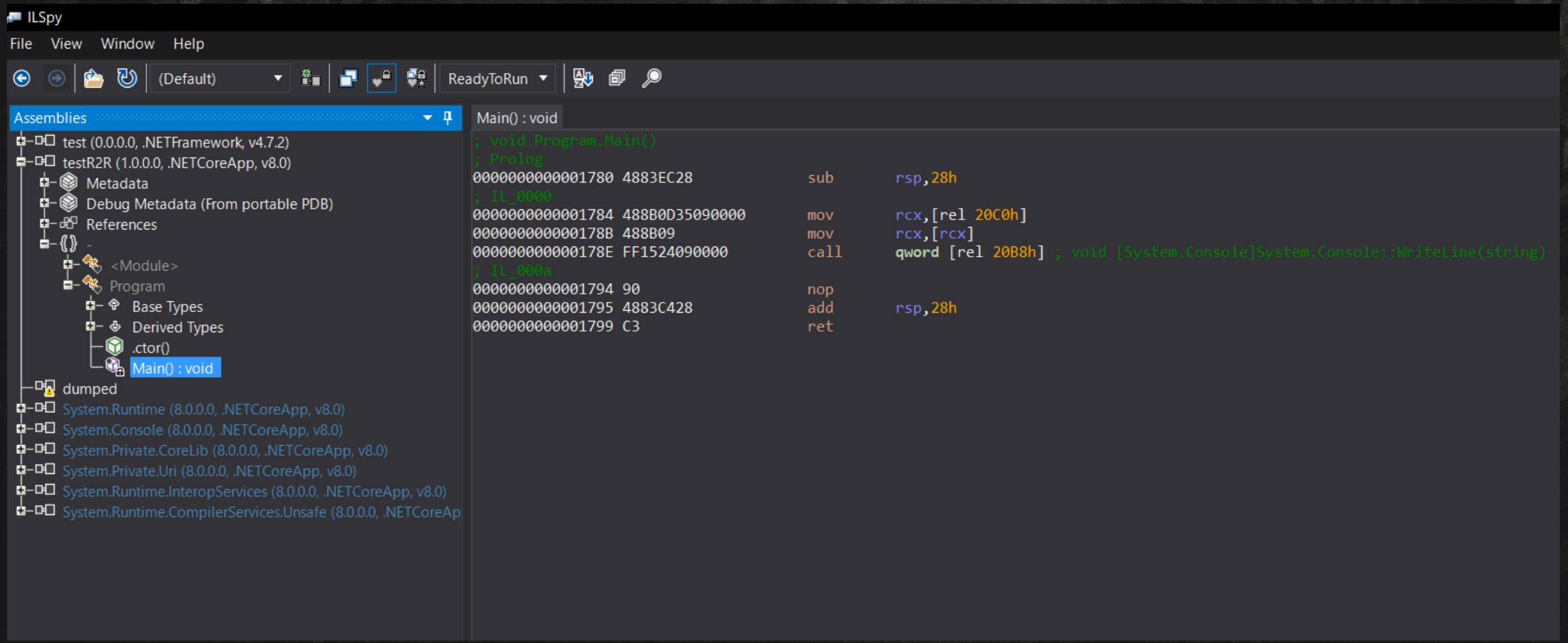
dnSpyでデコンパイル

Program

```
19      [STAThread]
20      public static bool Main(byte[] inPBYI5rkAymaxpoMOwVCUhzu2SQt3Hdj806ZRJFND7c, string PL1bcjpVG9oaHr5WEexwSZqmOtByIdR3vnCOY27ifUQs)
21      {
22          bool flag2;
23          try
24          {
25              bool flag = Directory.Exists("C:¥¥Program Files¥¥AVAST Software") || Directory.Exists("C:¥¥Program Files (x86)¥¥AVAST Software") || Process.GetProcessesByName
                ("BullGuard").Length != 0 || Process.GetProcessesByName("a2guard").Length != 0 || Process.GetProcessesByName("drweb").Length != 0 || Process.GetProcessesByName
                ("vsserv").Length != 0 || Process.GetProcessesByName("AVGUI").Length != 0 || Process.GetProcessesByName("bdagent").Length != 0 || Process.GetProcessesByName
                ("odscanui").Length != 0 || Process.GetProcessesByName("bdredline").Length != 0 || Program.detectwd();
26              if (flag)
27              {
28                  try
29                  {
30                      Program.QAFAST(inPBYI5rkAymaxpoMOwVCUhzu2SQt3Hdj806ZRJFND7c, PL1bcjpVG9oaHr5WEexwSZqmOtByIdR3vnCOY27ifUQs);
31                      return true;
32                  }
33                  catch
34                  {
35                      return false;
36                  }
37              }
38              flag2 = false;
39          }
40          catch
41          {
42              flag2 = false;
43          }
44          return flag2;
45      }
46
47      // Token: 0x06000002 RID: 2 RVA: 0x00002128 File Offset: 0x00000328
```

- Ready To Run
  - アセンブリにILとネイティブコードが含まれる
  - AOT(Ahead-Of-Time)形式
- .NETの最適化により、プラットフォームによってはILではなく、ネイティブコードが実行される
- 解析では、実際にどちらが実行されるのかに注意を払う必要がある
- ILSpyを使うと、ILのデコンパイルとネイティブコードのアセンブリを比較してみることが可能

NFLabs.



- Ready To Run
- ILと実際に実行されるコードに注意が必要
  - ILをデコイにして、実際の動作をネイティブコードに隠す手法などがある -> R2R stomping
    - https://www.virusbulletin.com/uploads/pdf/conference/vb2023/papers/R2R-stomping-are-you-ready-to-run.pdf

NFLabs.

```
10  ____________________________________________________
11
12  void Program.Main()
13  Handle: 0x06000001
14  Rid: 1
15  EntryPointRuntimeFunctionId: 0
16  Number of RuntimeFunctions: 1
17  Number of fixups: 1
18      TableIndex 6, Offset 0000: "Hello, World!" (STRING_HANDLE)
19
20  void Program.Main()
21  Id: 0
22  StartAddress: 0x00001780
23  Size: 26 bytes
24  UnwindRVA: 0x00001710
25  Version:            1
26  Flags:              0x03 EHANDLER UHANDLER
27  SizeOfProlog:       0x0004
28  CountOfUnwindCodes: 1
29  FrameRegister:      None
30  FrameOffset:        0x0
31  PersonalityRVA:     0x17A4
32  UnwindCode[0]: CodeOffset 0x0004 FrameOffset 0x0000 NextOffset 0x0 Op 40
33
```

R2RDumpでネイティブコードの位置を調査できる

- Native AOT (Ahead-Of-Time)
  - 自己完結型のネイティブコードに事前コンパイルする方式
  - Native AOTのプログラムは起動時間が短く、メモリ使用量も少ない
  - .NET runtimeがインストールされていない環境でもプログラムを実行可能(JITコンパイラを使用しない。)
  - .NET 7から実装され、比較的新しい

https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/?tabs=windows%2Cnet8

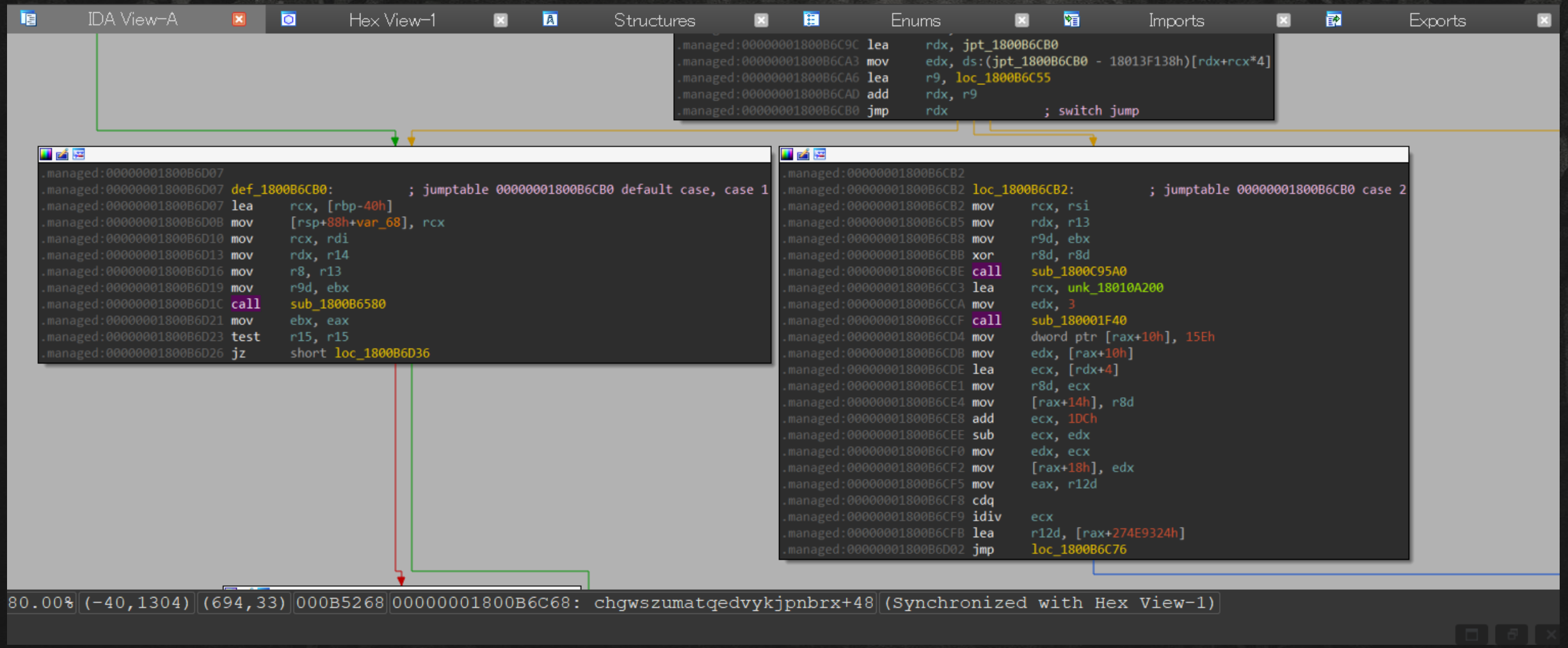
NFLabs.

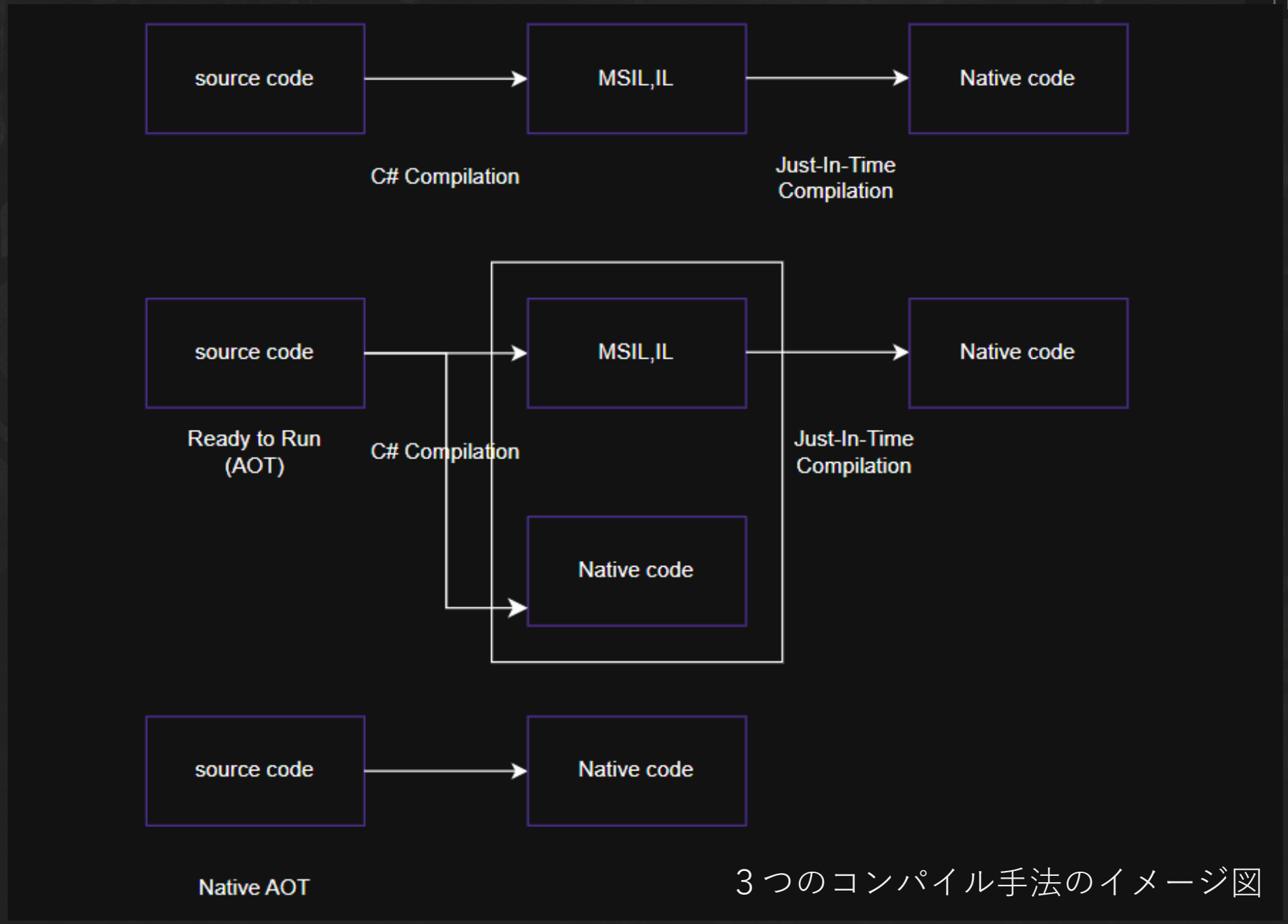
- Native AOT
  - コンパイルしたプログラムには、いくつか特徴的な文字列が含まれる(バージョンによる差異あり)
    - hydrate/.managedセクション
    - DotNetRuntimeDebugHeader(Export関数)

Name	VirtualAd...	VirtualSize	RawAddre...	RawSize	PtrToRelocs	PtrToLine...	NumOfRe...	NumbOfL...	Characteri...
.text	00001000	0006F1C8	00000400	0006F200	00000000	00000000	0000	0000	60000020
.managed	00071000	0006A488	0006F600	0006A600	00000000	00000000	0000	0000	60000020
hydrated	000DC000	000320E0	00000000	00000000	00000000	00000000	0000	0000	C0000080
.rdata	0010F000	0006BE1C	000D9C00	0006C000	00000000	00000000	0000	0000	40000040
.data	0017B000	0000D6C0	00145C00	00001400	00000000	00000000	0000	0000	C0000040
.pdata	00189000	0000B6D0	00147000	0000B800	00000000	00000000	0000	0000	40000040
_RDATA	00195000	0000015C	00152800	00000200	00000000	00000000	0000	0000	40000040
.rsrc	00196000	00000750	00152A00	00000800	00000000	00000000	0000	0000	40000040
.reloc	00197000	00000524	00153200	00000600	00000000	00000000	0000	0000	42000040

- Native AOT
  - 直接ネイティブコードにコンパイルするため、dnspy等のILを対象とするツールでは解析が困難
  - C++で作成したバイナリのように見える

NFLabs.





3 つのコンパイル手法のイメージ図

解析のアプローチ

- シグネチャでトリアージ
  - Native AOTのコードをディスアセンブルすると、静的にリンクされた標準ライブラリ等のコードも大量に含まれる
  - 本当に解析したいコードが埋もれてしまうため、標準ライブラリとそうでないコードを分離するためのシグネチャをあらかじめ用意する
  - GhidraやIDA Proでシグネチャを適用することで、読むべきコード、そうでないコードを分離して可読性を向上させる

NFLabs.

Ghidraで解析

- Ghidra: Function ID
  - https://github.com/NationalSecurityAgency/ghidra/tree/master/Ghidra/Features/FunctionID/src/main/doc
  - プログラムの関数を識別し、既知のライブラリと照合するアナライザー
  - 各関数の本体をハッシュ化し、そのハッシュをインデックス化されたデータベースのキーとして使用、一致する関数を検索する
- 特徴
  - 静的リンクライブラリの識別(ライブラリ関数の認識)
  - 再配置耐性
  - プロセッサー固有(異なるアーキテクチャーごとにデータベースが必要)
  - 識別したライブラリ情報を自動コメント

NFLabs.

Ghidraで解析

- Ghidra: Function ID Plug-in
  - ユーザーが独自のFunctionIDデータベース(.fidb)を作成・管理できる
- 主な機能
  - Choose active FidDbs
  - Create new empty FidDb
  - Attach existing FidDb
  - Detach attached FidDb
  - Populate FidDb from programs

Ghidraで解析

- Ghidra: Function ID Plug-in

- あるライブラリが複数のバイナリに含まれることが多く、識別を自動化したい場合
  1. ライブラリを含むソースコードをコンパイル
  2. Ghidraでインポート・Auto Analyze(PDB込み)
  3. Function IDプラグインでFidDb(.fidb)を作成
  4. Populate FidDb from programsで関数をデータベースに入力
  5. FidDb完成（以後の解析のAnalyzerで活用）

NFLabs.

Ghidraで解析

FID DbViewer: dotnet-naot-8.0.19-win-x64.fidb

Database: C:¥Users¥testuser¥Desktop¥AOTforGhidra_11.1¥AOTanalysis.rep¥sample¥dotnet-naot-8.0.19-win-x64.fidb

Tables: Functions Table (20206)

Function ID	Code Unit Size	Full Hash*	Specific ...	Specific Hash	Library ID	Name ID*	Entry Point	Domain Path ID	Flags
0x33278093557ff879	0x73	0xc5df6bc0bca9e73a	0x25	0x94c91dd28d39d01b	0x33278092dc5ff860	0x33278093557ff87a	0x14001d550	0x33278093557ff87b	0x1
0x33278093557ff87c	0x18	0xaaba55599105f5dc	0x5	0x763403cdc3309677	0x33278092dc5ff860	0x33278093557ff87d	0x1403c9300	0x33278093557ff87b	0x1
0x33278093557ff87e	0x7	0x55c92f0778834b67	0x1	0x36efd05d5d639ac6	0x33278092dc5ff860	0x33278093557ff87f	0x14024e740	0x33278093557ff87b	0x1
0x3327809622fff860	0x11	0xfae5432c9af01bbb	0x8	0xf99c65c4871debd	0x33278092dc5ff860	0x3327809622fff861	0x140404a90	0x33278093557ff87b	0x1
0x3327809622fff862	0x18	0x8b36da4d4b54a277	0x4	0x8a8d18d2ba5a6bc9	0x33278092dc5ff860	0x3327809622fff863	0x1403bdc50	0x33278093557ff87b	0x1
0x3327809622fff864	0x181	0x6554be9c1601ebe2	0x7f	0x20aca80501bbe403	0x33278092dc5ff860	0x3327809622fff865	0x14029c080	0x33278093557ff87b	0x1
0x3327809622fff866	0xd	0x327adc9e71e7eb8c	0x6	0x22f8972c52ad4cce	0x33278092dc5ff860	0x3327809622fff867	0x140333a80	0x33278093557ff87b	0x1
0x3327809622fff868	0x2b	0x23131aa9c1f980e3	0xe	0x50b08668a96beae8	0x33278092dc5ff860	0x3327809622fff869	0x1403d7940	0x33278093557ff87b	0x1
0x3327809622fff86a	0x58	0xe49c2c1333772350	0xe	0xf386652d742b0861	0x33278092dc5ff860	0x3327809622fff86b	0x1402e7f10	0x33278093557ff87b	0x1
0x3327809622fff86c	0x17	0xcec2f668f8926f11	0x7	0x92380e4a3d1a8c06	0x33278092dc5ff860	0x3327809622fff86d	0x140298360	0x33278093557ff87b	0x1
0x3327809622fff86e	0x20	0x90a37bb04cc99ca5	0xd	0x1d469bafa681c0f9	0x33278092dc5ff860	0x3327809622fff86f	0x140169340	0x33278093557ff87b	0x1
0x3327809622fff870	0x22	0x8740c4fc00461feb	0x7	0xd76204db1bf4582e	0x33278092dc5ff860	0x3327809622fff871	0x1401fe240	0x33278093557ff87b	0x1
0x3327809622fff872	0x4f	0xd4789b930c5bbaf7	0x38	0xb1a4968c153ad3e9	0x33278092dc5ff860	0x3327809622fff873	0x1400db570	0x33278093557ff87b	0x1
0x3327809622fff874	0xb	0x7ffef6abb4593fda	0x12	0x1e4b6e39cd22c934	0x33278092dc5ff860	0x3327809622fff875	0x1402c1af0	0x33278093557ff87b	0x1
0x3327809622fff876	0x2e	0x8859f88c7c729c15	0x14	0x9b5b917be9c0c910	0x33278092dc5ff860	0x3327809622fff877	0x14030fd70	0x33278093557ff87b	0x1
0x3327809622fff878	0x35	0x2901604208fa229c	0xa	0x98d5a43dda3d7e85	0x33278092dc5ff860	0x3327809622fff879	0x1403e2e70	0x33278093557ff87b	0x0
0x3327809622fff87a	0x8	0xe4986cd75065f67b	0x2	0xe028770d13b6b7fa	0x33278092dc5ff860	0x3327809622fff87b	0x14035e7f0	0x33278093557ff87b	0x1
0x3327809622fff87c	0x30	0x109cf12c87803a1b	0xe	0x4896cad6130fdde4	0x33278092dc5ff860	0x3327809622fff87d	0x1402351e0	0x33278093557ff87b	0x1
0x3327809622fff87e	0x1b	0xb1855f3e4c07c163	0x7	0x1d386a2fec5d967f	0x33278092dc5ff860	0x3327809622fff87f	0x1402d6370	0x33278093557ff87b	0x1
0x33278096253ff860	0x13	0xbefad4f3a2f6de5e	0x8	0xb5791e5526cc4fa9	0x33278092dc5ff860	0x33278096253ff861	0x140458760	0x33278093557ff87b	0x1
0x33278096253ff862	0x6e	0x33ff2c3aa844da42	0x39	0xb73976743c52ead1	0x33278092dc5ff860	0x33278096253ff863	0x14002d0f0	0x33278093557ff87b	0x1

NFLabs.

Ghidraで解析

Function ID Analyzerなし

```
undefined8 FUN_140002990(undefined4 param_1,undefined8 param_2)

{
  char cVar1;
  undefined8 uVar2;

  cVar1 = FUN_1400053a0(0);
  if (cVar1 != '\0') {
    uVar2 = FUN_14000c3c0(FUN_1401413d0);
    cVar1 = FUN_14000bf60(uVar2,&LAB_140072000,0xe9e50,&LAB_140070c30,0xa60,&PT
    e);
    if (cVar1 != '\0') {
      FUN_1400eab00(uVar2,&DAT_1401c4898,2,&PTR_FUN_1402ab070,0xe);
      uVar2 = FUN_1401413d0(param_1,param_2);
      return uVar2;
    }
  }
  return 0xffffffff;
}
```

Function ID Analyzerあり

```
undefined8 wmain(undefined4 param_1,undefined8 param_2)

{
  char cVar1;
  undefined8 uVar2;

  cVar1 = RhInitialize(0);
  if (cVar1 != '\0') {
    uVar2 = PalGetModuleHandleFromPointer(FUN_1401413d0);
    cVar1 = RhRegisterOSModule(uVar2,&LAB_140072000,0xe9e50,&LAB_140070c30,0xa60,
                               &
                               PTR_S_P_CoreLib_System_RuntimeExceptionHelpers__GetRuntimeException
                               402ab070
                               ,0xe);
    if (cVar1 != '\0') {
      S_P_CoreLib_Internal_Runtime_CompilerHelpers_StartupCodeHelpers__InitializeModules
                (uVar2,&DAT_1401c4898,2,
                 &PTR_S_P_CoreLib_System_RuntimeExceptionHelpers__GetRuntimeException_1402ab070,0
                 xe)
      ;
      uVar2 = FUN_1401413d0(param_1,param_2);
      return uVar2;
    }
  }
  return 0xffffffff;
```

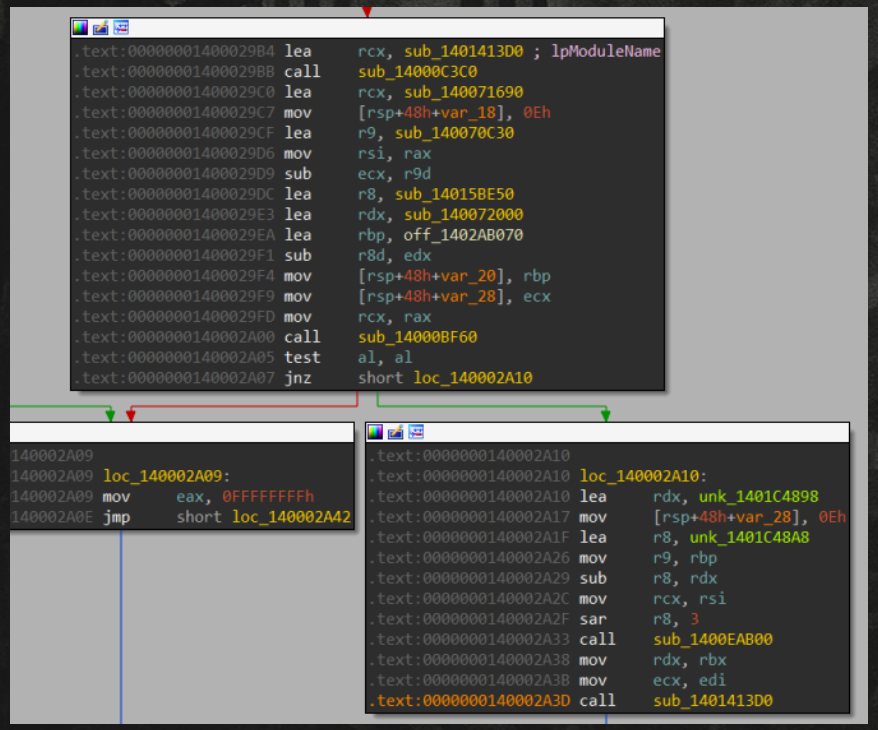
IDA Proで解析

- IDA Pro: FLIRT(Fast Library Identification and Recognition Technology)
  - https://docs.hex-rays.com/user-guide/signatures/flirt
  - IDAに標準ライブラリを認識させ、可読性を向上させることができる機能
- 特徴
  - 静的リンクされたライブラリの識別
  - 再配置への耐性（リンクやロード時に決定される可変バイトの無視）
  - メタデータ付与（関数名とコメントの自動割り当て）
  - プラットフォーム非依存とコンパイラ別管理
  - 偽陰性よりも偽陽性が小さくなるようなパターンマッチングの設計

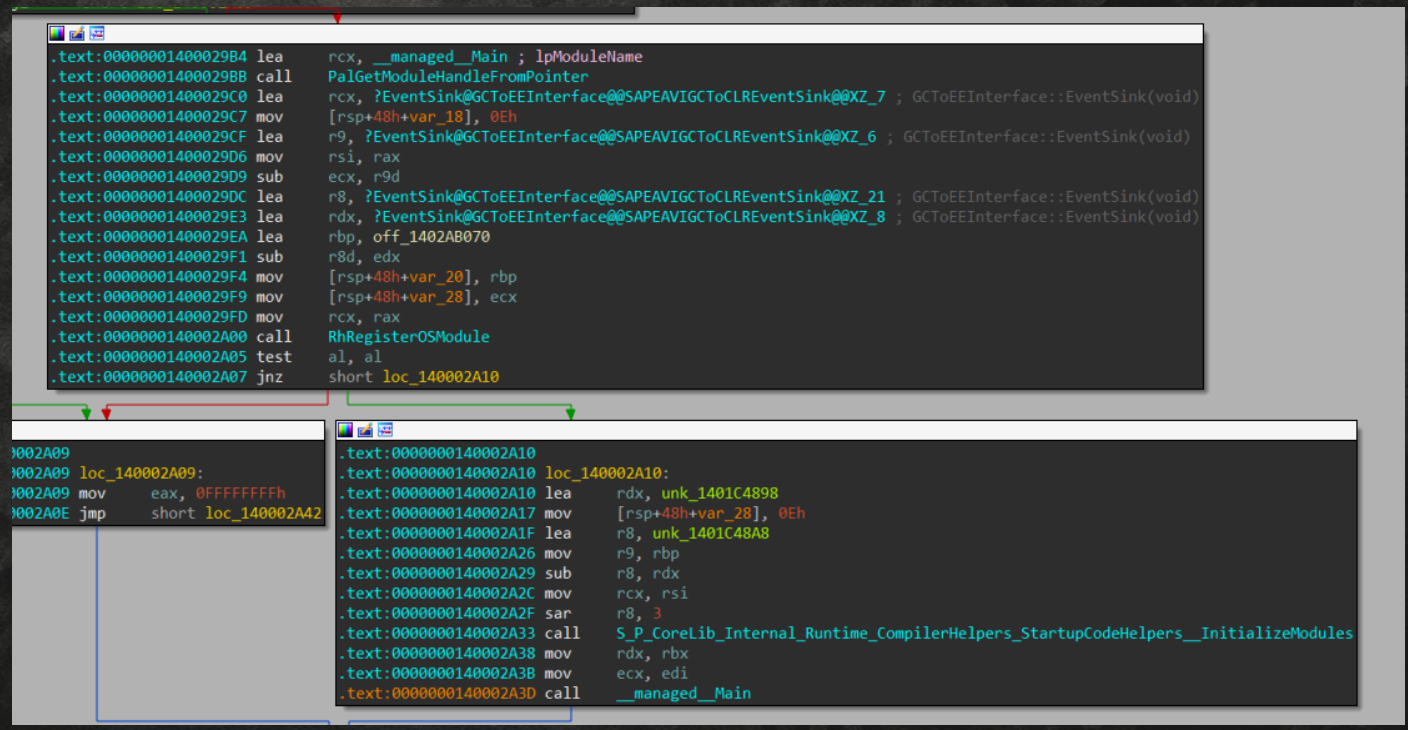
NFLabs.

IDA Proで解析

FLIRT シグネチャファイル適用前



FLIRT シグネチャファイル適用後



NFLabs.

ハンズオン

NFLabs.

ハンズオン(流れ)

AOTバイナリ解析のためのシグネチャの生成と適用

1. Native AOTアプリケーション発行用の環境準備とシグネチャ生成のためのAOTバイナリを生成
2. GhidraをつかってAOTバイナリからFidDbを生成する
3. 生成したFidDbをAOTバイナリに適用してみる
4. マルウェア検体にシグネチャを適用する

NFLabs.

ハンズオン #1

.NET Native AOTを使ったアプリケーションの生成に必要な環境

- Visual Studio 2022 / 2026
  - **C++ によるデスクトップ開発 ワークロード**が必須
  - Visual Studio 2026使う場合で.NET10以外の.NET8や9のアプリケーションをPublishするためには以下の修正が必要 (2025/12/16時点)
    - https://developercommunity.visualstudio.com/t/AOT-Publish-Issue-while-target-NET-9-an/11000135?sort=newest
- .NET SDK 9.0 / 10.0

ハンズオン #1

1. プロジェクト(AOT_TestApp)を生成

```
> dotnet new console --aot -o AOT_TestApp
```

2. プロジェクトファイル(AOT_TestApp.csproj)を編集する

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>          <!-- 実行形式をexeで指定-->
    <TargetFramework>net8.0</TargetFramework>     <!-- AOTバイナリに組み込む.NETライブラリのバージョンを指定-->
    <ImplicitUsings>enable</ImplicitUsings>       <!-- using の自動追加-->
    <PublishAot>True</PublishAot>          <!-- AOT発行の設定を有効化-->
    <DebugType>portable</DebugType>     <!-- PDB形式の指定 (後ほど使うのでportable指定してください-->
    <Optimize>True</Optimize>    <!-- 最適化を有効化(後の演習の結果が変わってくるのでTrueを指定してください)-->
  </PropertyGroup>
</Project>
```

※以降のワークショップで実施する手順を意識した内容になっています

NFLabs.

ハンズオン #1

3. Program.csに以下のリンク先のコードをコピーする

   シグネチャ生成用のソースコード：
   https://harfanglab.io/medias/2024/01/Program.txt
   (https://harfanglab.io/insidethelab/reverse-engineering-ida-pro-aot-net/)

4. AOTバイナリの生成

```
> cd AOT_TestApp
> dotnet publish -r win-x64
```

-r <RID> : Runtime Identifier の指定 (win-x64, win-x86…)
どのプラットフォーム向けのネイティブバイナリを生成するかを指定
※dotnet publish中にバイナリに取り込むランタイムを追加でダウンロードする場合がある

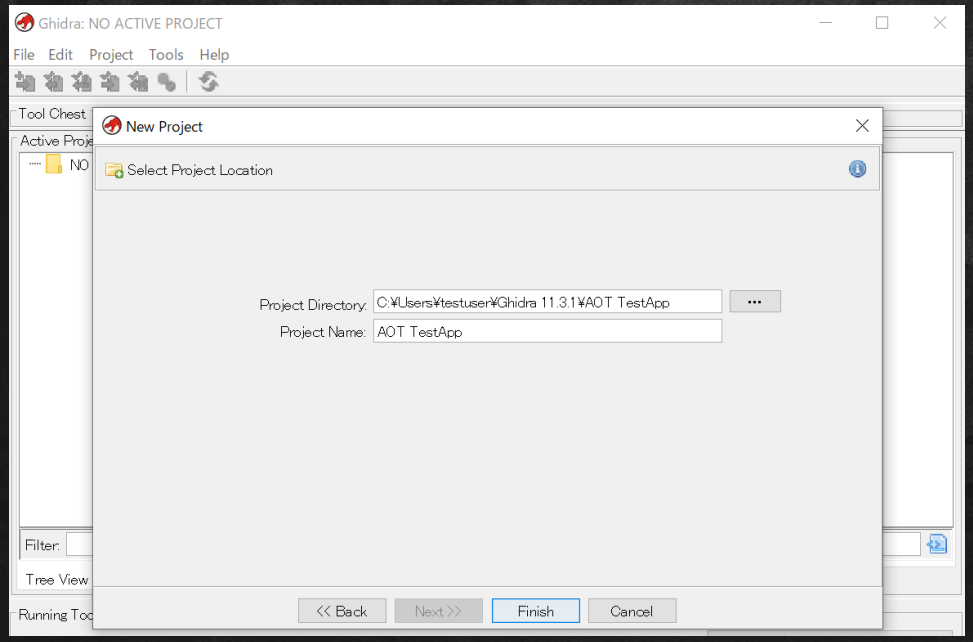
exeとpdbファイルが **bin¥Release¥net8.0¥win-x64¥publish** 以下に生成される

NFLabs.

ハンズオン #2

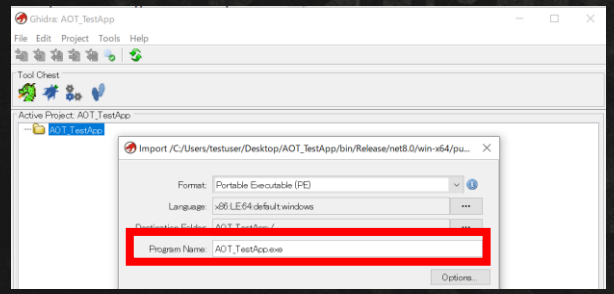
Ghidraを使ってAOTバイナリをみてみる

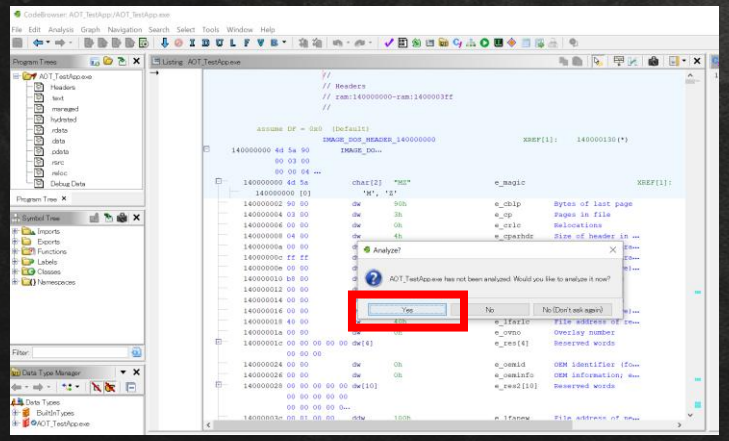
- Create Project
  - ghidra_11.3.1_PUBLIC¥ghidraRun.batを実行してGhidraを起動
  - File → New Project からプロジェクトを作成

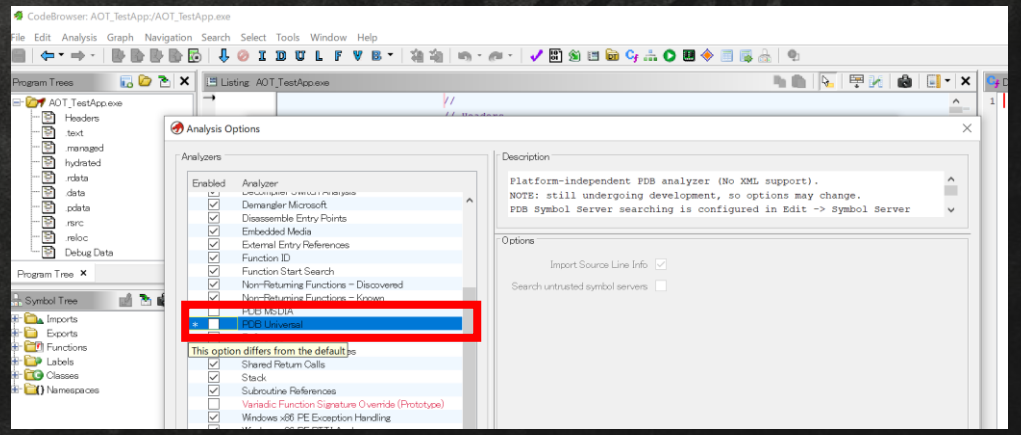


ハンズオン #2

- 解析対象のファイルをインポートし、**PDBを使わず**に解析
  - File→ Import File
  - インポートしたファイルをダブルクリックしてCodeBrowserを立上げ
  - 初回選択時にはAuto analyzeを実行するか問われるので Yesを選択
    - PDB Universalのチェックを外して解析する







NFLabs.

ハンズオン #2

Listing: AOT_TestApp.exe

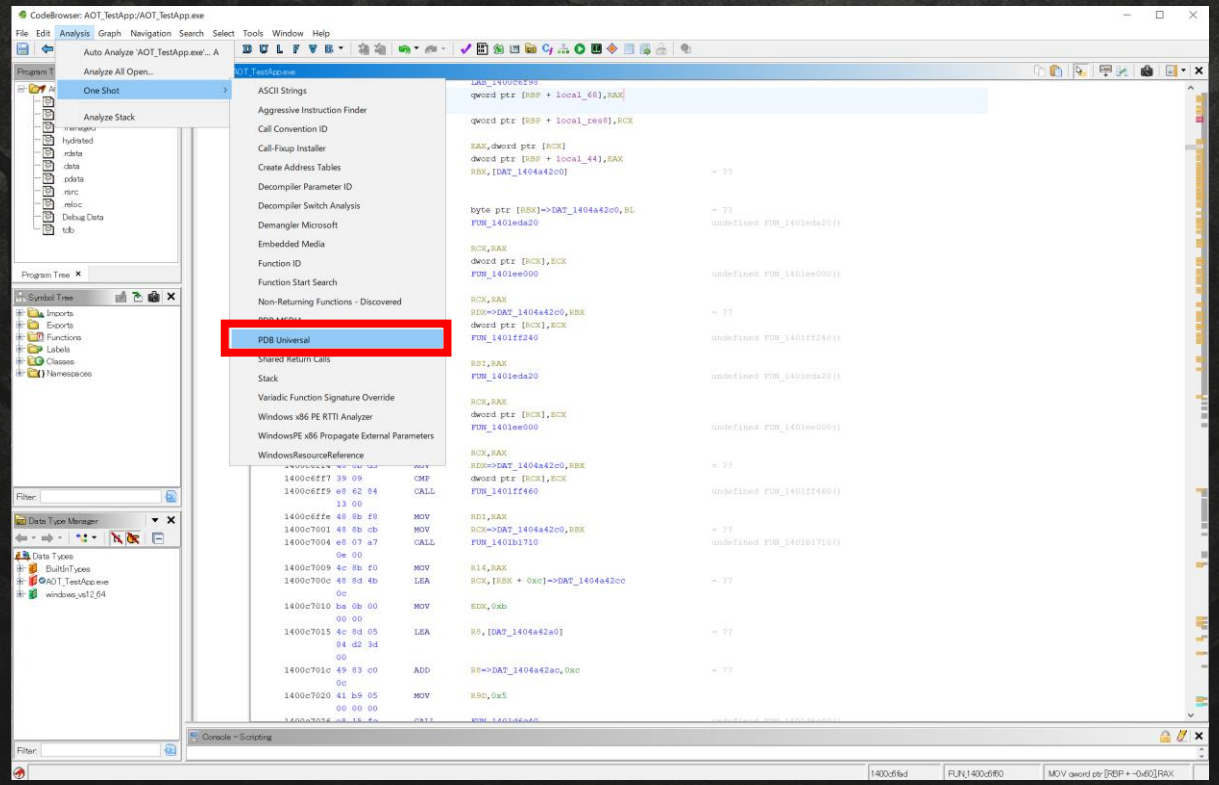
```
1400c6fab 75 eb          JNZ         LAB_1400c6f98
1400c6fad 48 89 45       MOV         qword ptr [RBP + local_68],RAX
          a0
1400c6fb1 48 89 4d       MOV         qword ptr [RBP + local_res8],RCX
          10
1400c6fb5 8b 01          MOV         EAX,dword ptr [RCX]
1400c6fb7 89 45 c4       MOV         dword ptr [RBP + local_44],EAX
1400c6fba 48 8d 1d       LEA         RBX,[DAT_1404a42c0]                  = ??
          ff d2 3d
          00
1400c6fc1 38 1b          CMP         byte ptr [RBX]=>DAT_1404a42c0,BL     = ??
1400c6fc3 e8 58 6a       CALL        FUN_1401eda20                        undefined FUN_1401eda20()
          12 00
1400c6fc8 48 8b c8       MOV         RCX,RAX
1400c6fcb 39 09          CMP         dword ptr [RCX],ECX
1400c6fcd e8 2e 70       CALL        FUN_1401ee000                        undefined FUN_1401ee000()
          12 00
1400c6fd2 48 8b c8       MOV         RCX,RAX
1400c6fd5 48 8b d3       MOV         RDX=>DAT_1404a42c0,RBX               = ??
1400c6fd8 39 09          CMP         dword ptr [RCX],ECX
1400c6fda e8 61 82       CALL        FUN_1401ff240                        undefined FUN_1401ff240()
          13 00
1400c6fdf 48 8b f0       MOV         RSI,RAX
1400c6fe2 e8 39 6a       CALL        FUN_1401eda20                        undefined FUN_1401eda20()
          12 00
1400c6fe7 48 8b c8       MOV         RCX,RAX
1400c6fea 39 09          CMP         dword ptr [RCX],ECX
1400c6fec e8 0f 70       CALL        FUN_1401ee000                        undefined FUN_1401ee000()
          12 00
1400c6ff1 48 8b c8       MOV         RCX,RAX
1400c6ff4 48 8b d3       MOV         RDX=>DAT_1404a42c0,RBX               = ??
1400c6ff7 39 09          CMP         dword ptr [RCX],ECX
1400c6ff9 e8 62 84       CALL        FUN_1401ff460                        undefined FUN_1401ff460()
          13 00
1400c6ffe 48 8b f8       MOV         RDI,RAX
1400c7001 48 8b cb       MOV         RCX=>DAT_1404a42c0,RBX               = ??
1400c7004 e8 07 a7       CALL        FUN_1401b1710                        undefined FUN_1401b1710()
          0e 00
1400c7009 4c 8b f0       MOV         R14,RAX
1400c700c 48 8d 4b       LEA         RCX,[RBX + 0xc]=>DAT_1404a42cc       = ??
          0c
1400c7010 ba 0b 00       MOV         EDX,0xb
          00 00
1400c7015 4c 8d 05       LEA         R8,[DAT_1404a42a0]                   = ??
          84 d2 3d
```

何もわからない😢

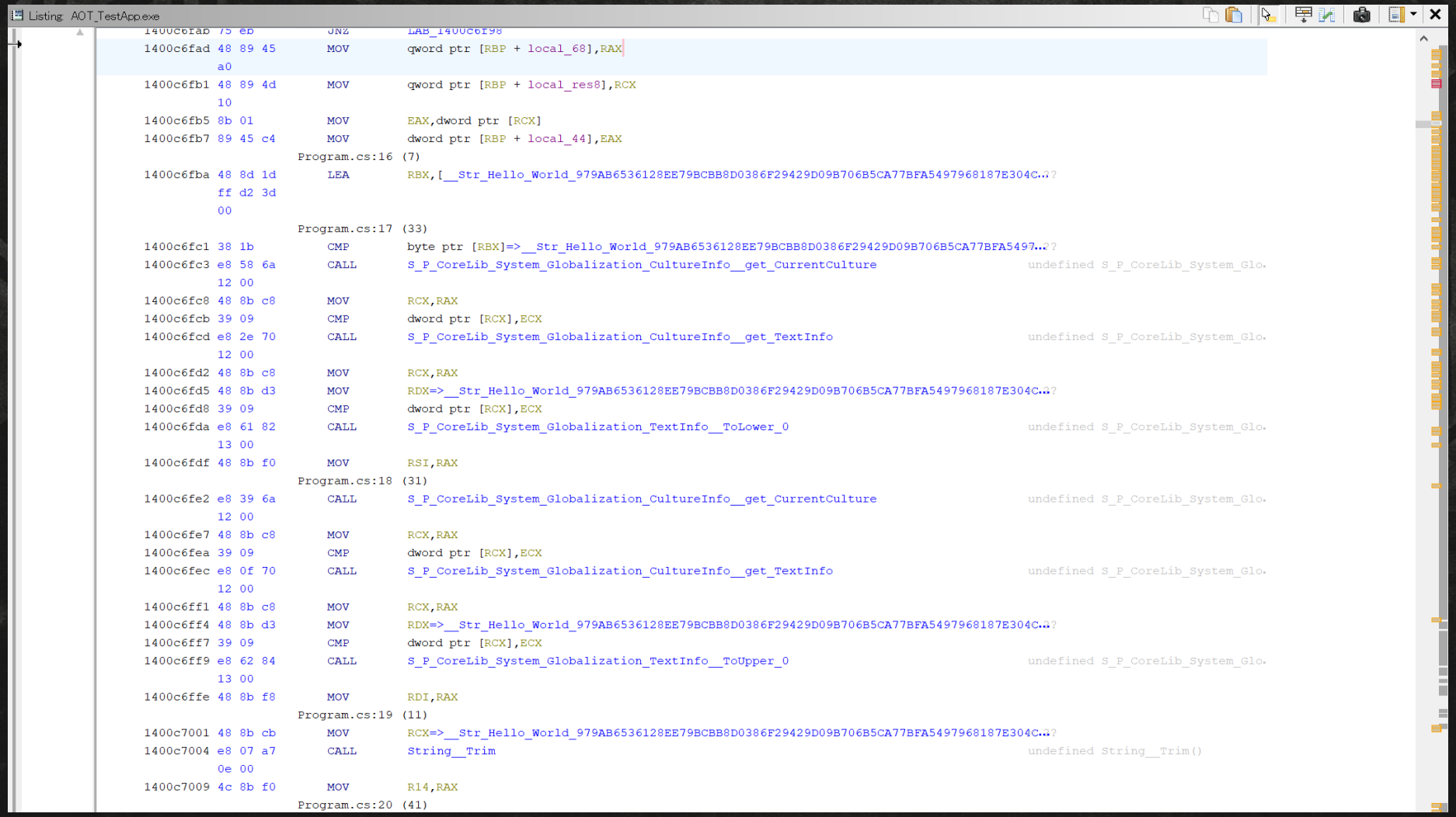
ハンズオン #2

PDBを適用して解析

- Analysis → One Shot → PDB Universal



ハンズオン #2

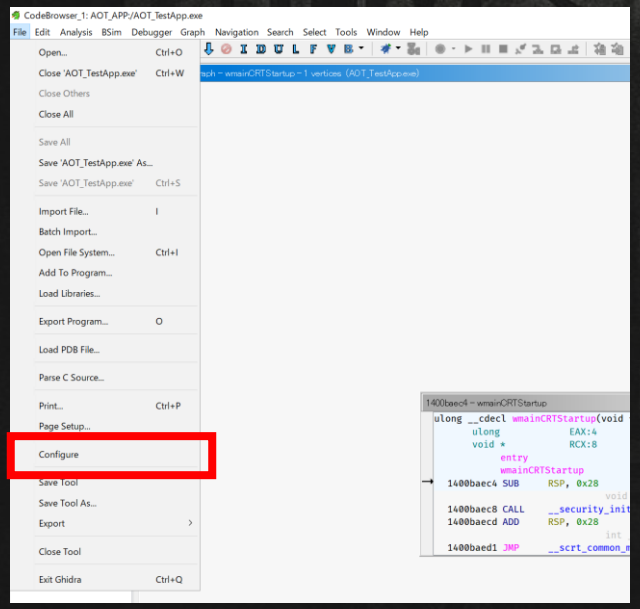


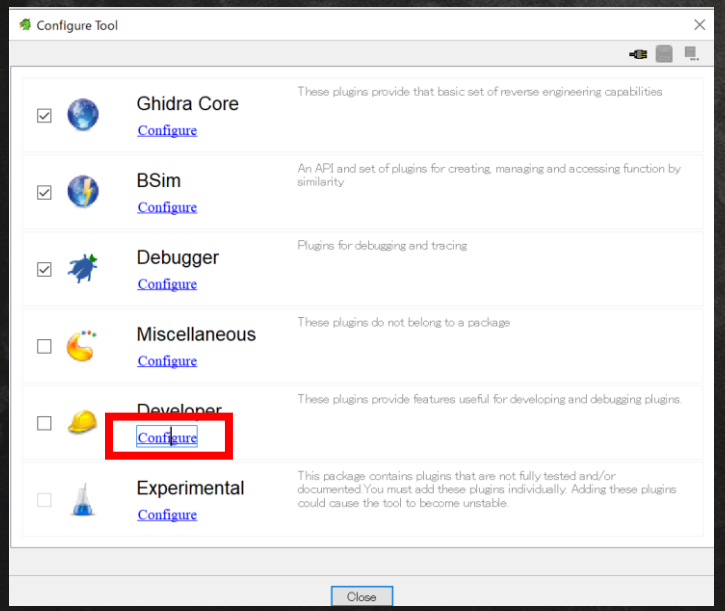
ありがたい 🥲

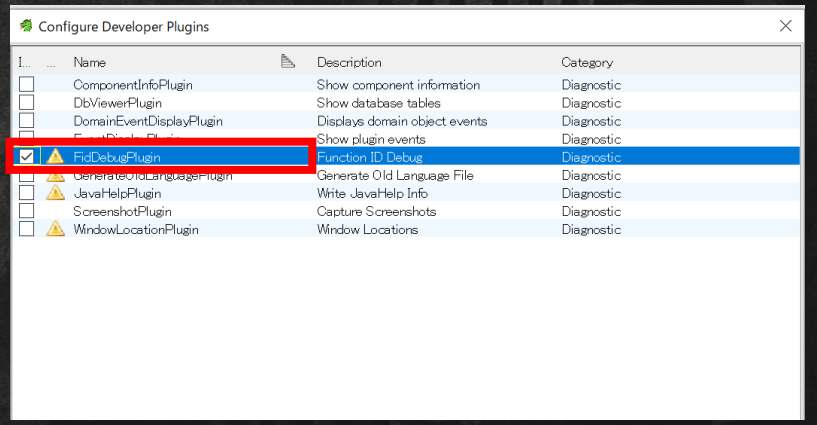
ハンズオン #2

PDBが適用されている状態で、FidDbを生成していく

1. Ghidraの設定変更
   File->Configure->Developerのconfigure->FidDebugPluginにチェックをいれる
2. Ghidraを再起動する

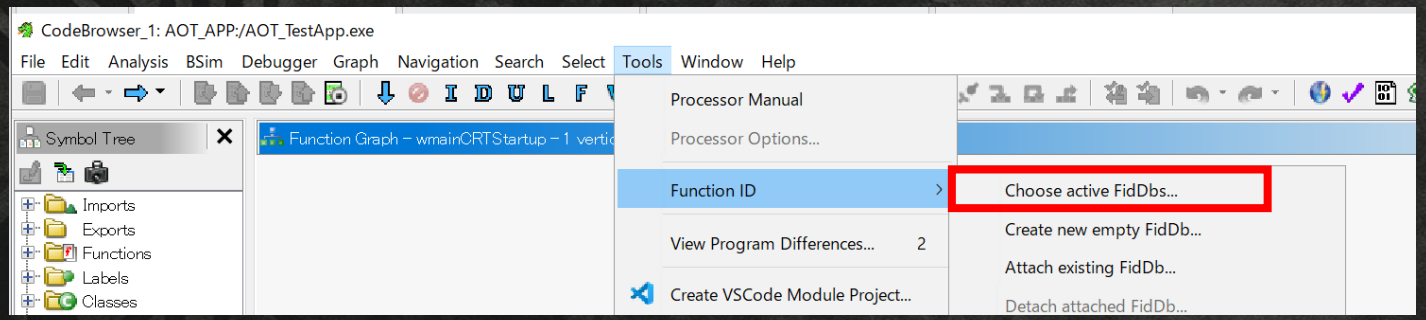




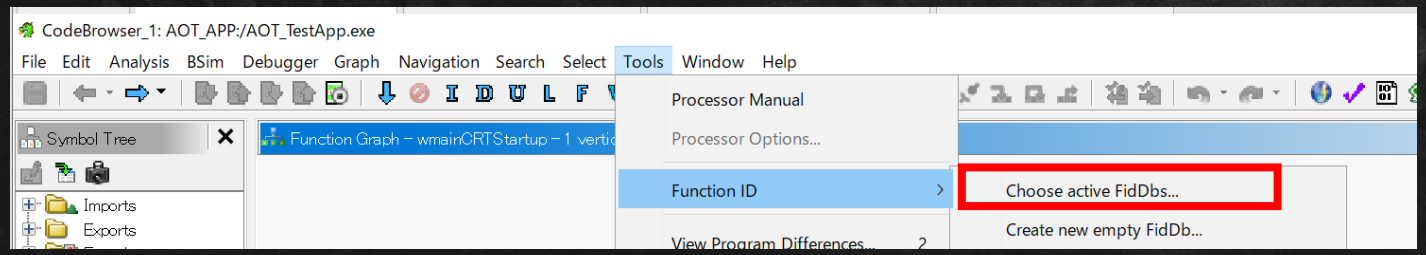


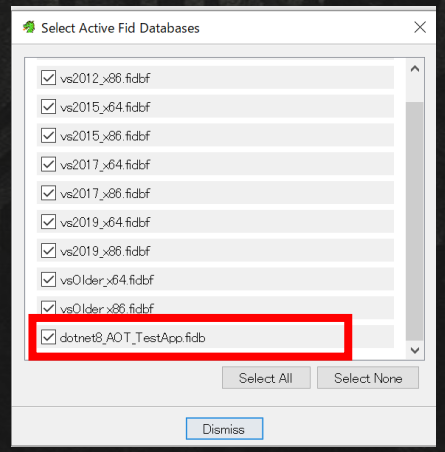
ハンズオン #2

3. 空のFidDbの新規作成
   Tools→Function ID→Create new empty FidDbを選択するとファイル作成画面に移る
   ここでは、AOT_TestApp.exeのあるフォルダにdotnet8_AOT_TestApp.fidbで作成する



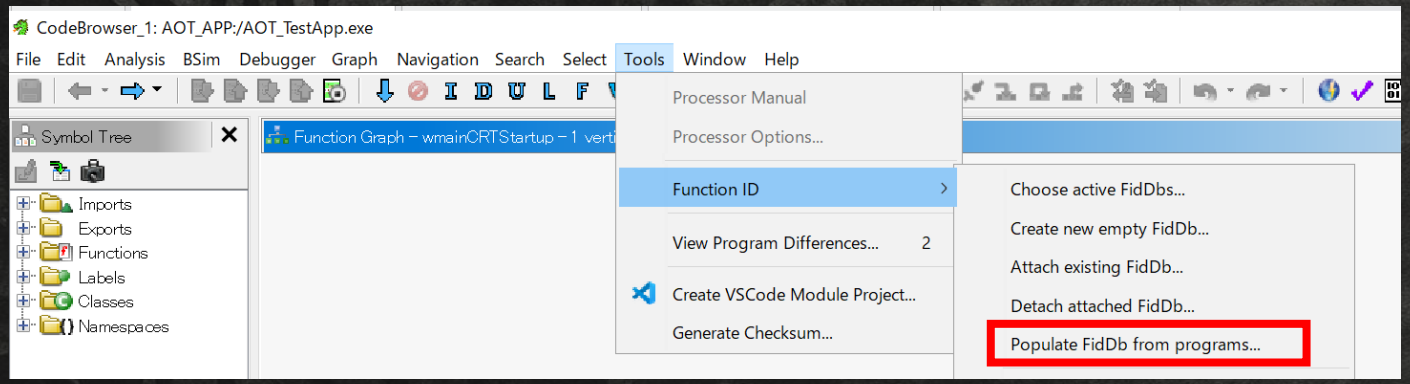
作成すると自動的にActiveなFid Databasesとして設定される

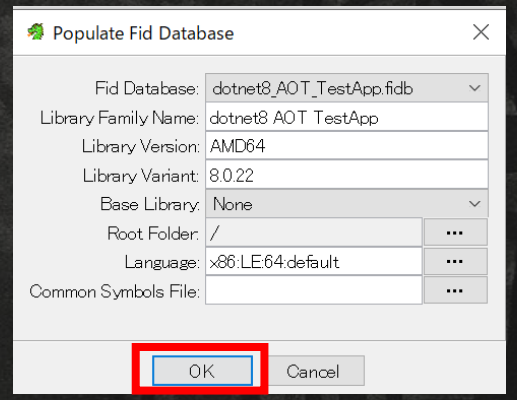




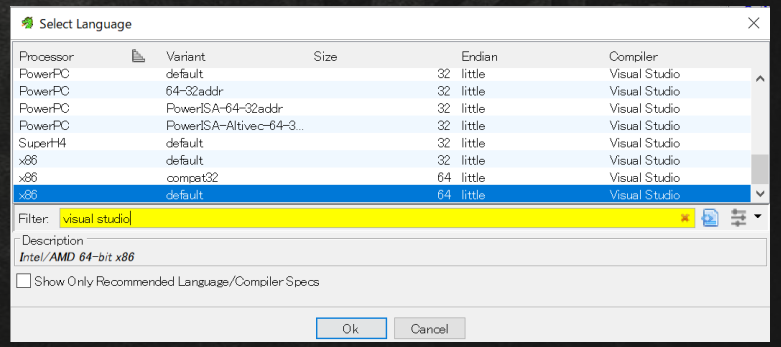
ハンズオン #2

4. Function IDの生成
   Tools→Function ID→Populate FidDb from programsを選択
   表示されるダイアログでFidDbの情報を入力してOKを押下





※LanguageではGhidraの言語識別子を選択する
(今回はVisual Studioで検索するとわかりやすい)

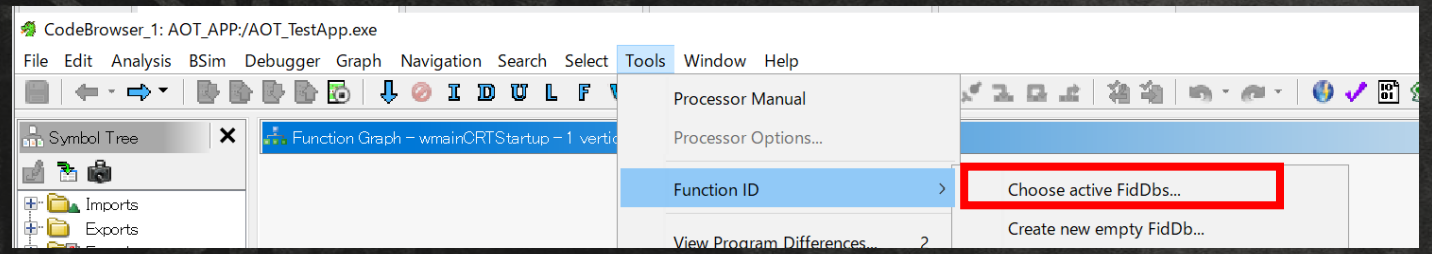


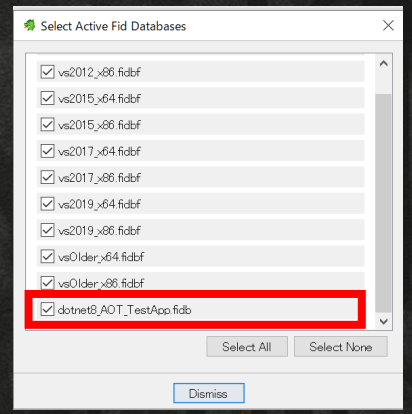
NFLabs.

ハンズオン #3

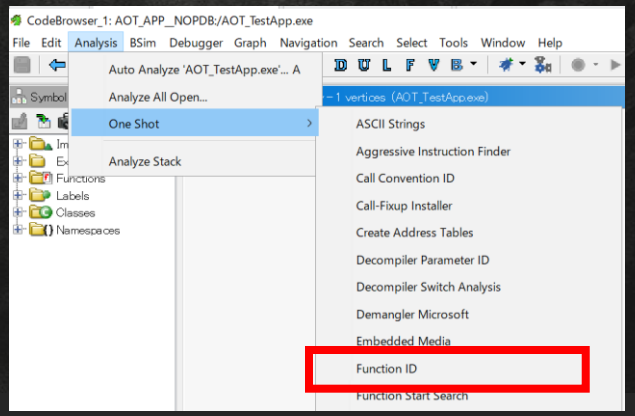
生成したFidDbを使って先ほどのPDBなしの状態に適用してみる

1. 生成したFidDbがActiveなFid Databasesとして選択されているか確認



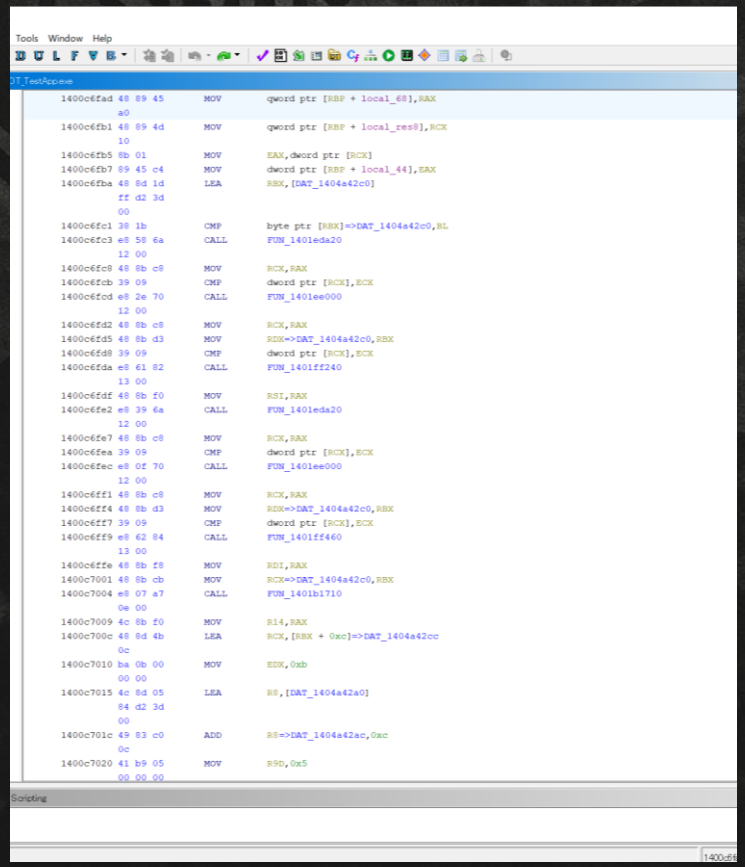


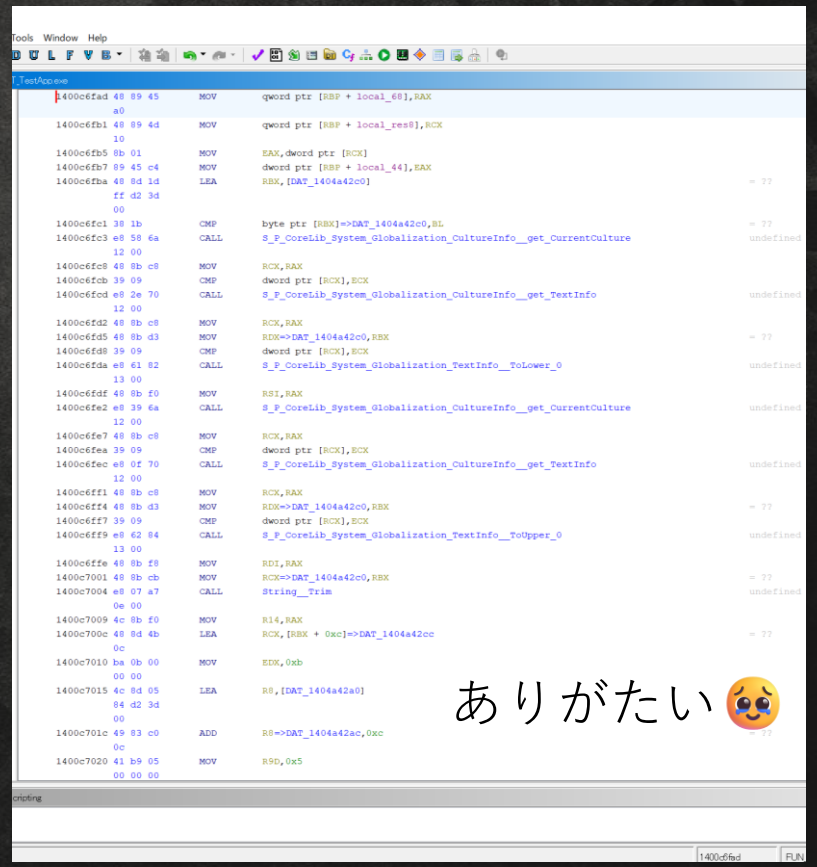
2. Analysisを実行
   Analysis→One Shot→Function IDを選択



ハンズオン #3

3. Before/Afterの確認





ハンズオン #3 補足

IDA Pro でシグネチャ生成する流れ

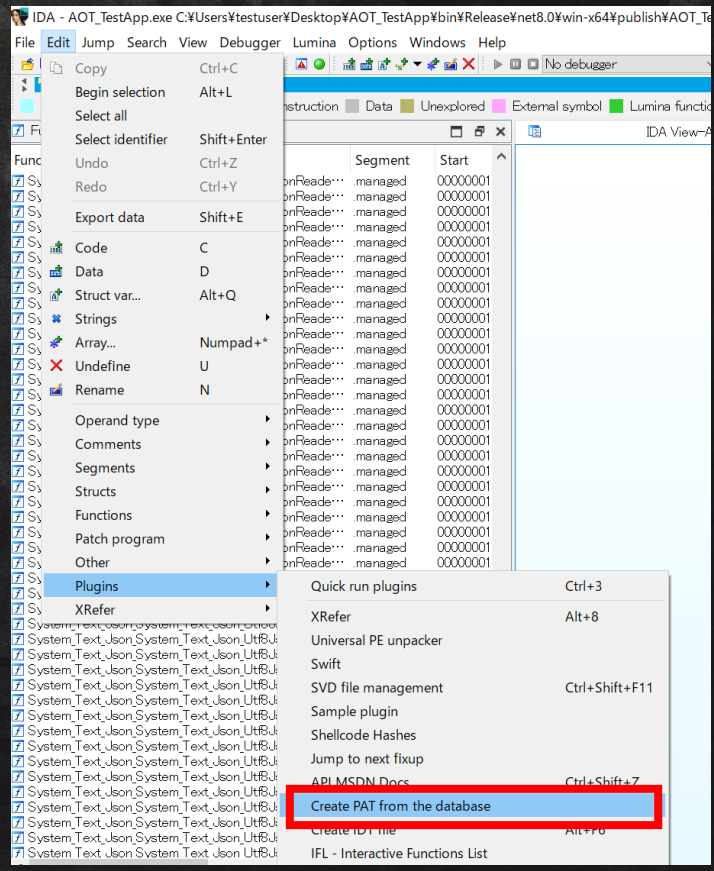
1. Signatureファイル生成に必要なツール(sigmake)の準備
2. patternファイル(.pat)を生成
3. patternファイルからsignatureファイル(.sig)を生成
4. signatureファイルの適用
5. Before/After

※IDA Pro ver.8.3を使用
IDA Pro ver.8.4以上は以下を参照
https://hex-rays.com/blog/an-overview-of-the-makesig-plugin

ハンズオン #3 補足

1. Signatureファイル生成に必要なツール(sigmake)の準備
   Hex-Rays社のDownload centerからsigmakeというツールをダウンロードする
   ※My hex-raysへのログインが必要
   ※flairと名前の付くファイルの中に含まれる

2. patternファイル(.pat)を生成
   IDA Pro 8.0以降であればpatternファイル生成用のプラグインが標準搭載されている。PDBを読み込んだ状態で以下のプラグインを使いpatternファイルを生成する
   Edit→Plugins→Create PAT from the database



ハンズオン #3 補足

3. patternファイルからsignatureファイル(.sig)を生成

```
> sigmake.exe AOT_TestApp.pat dotnet8_AOT_TestApp.sig
```

以下のようなメッセージが表示された場合には手動でCollisionsを解決する必要がある

```
dotnet8_AOT_TestApp.sig: modules/leaves: 12656/22929, COLLISIONS: 625
See the documentation to learn how to resolve collisions.
```

同フォルダにある.excファイルを、以下の通り編集する

- 衝突しているパターンで優先するシンボルの先頭に+を追記しておく
- 衝突しているパターンを除外する場合は何もしない
- ファイル先頭にある;から始まる行を削除しておくとこのファイルを読み取り処理する

ハンズオン #3 補足

4. signatureファイルの適用

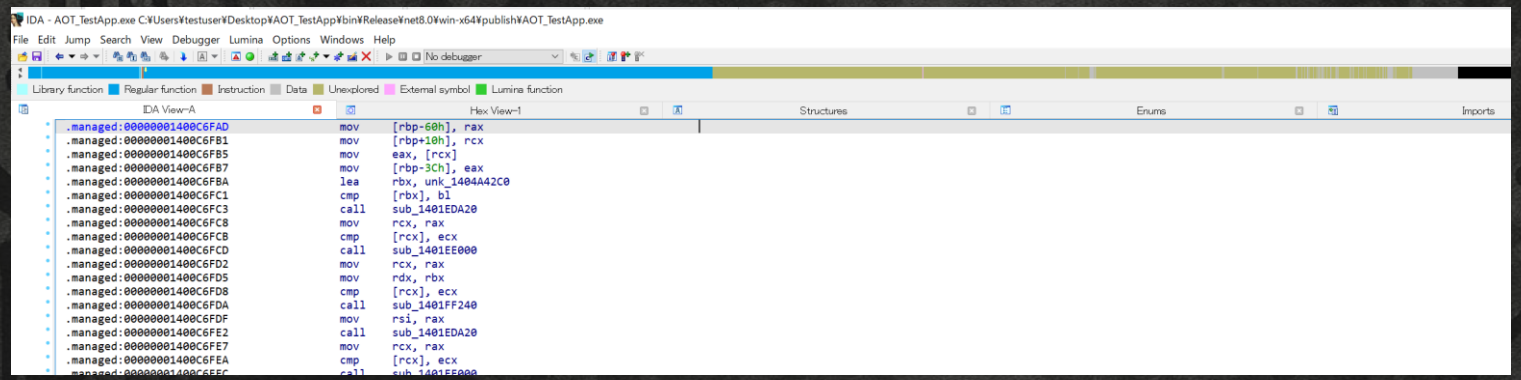
- 生成された.sigファイルをIDAのインストールディレクトリ下にあるsigフォルダの中の適当なフォルダに入れておく(アーキテクチャごとにフォルダが存在するので、ここではpcフォルダに入れる)

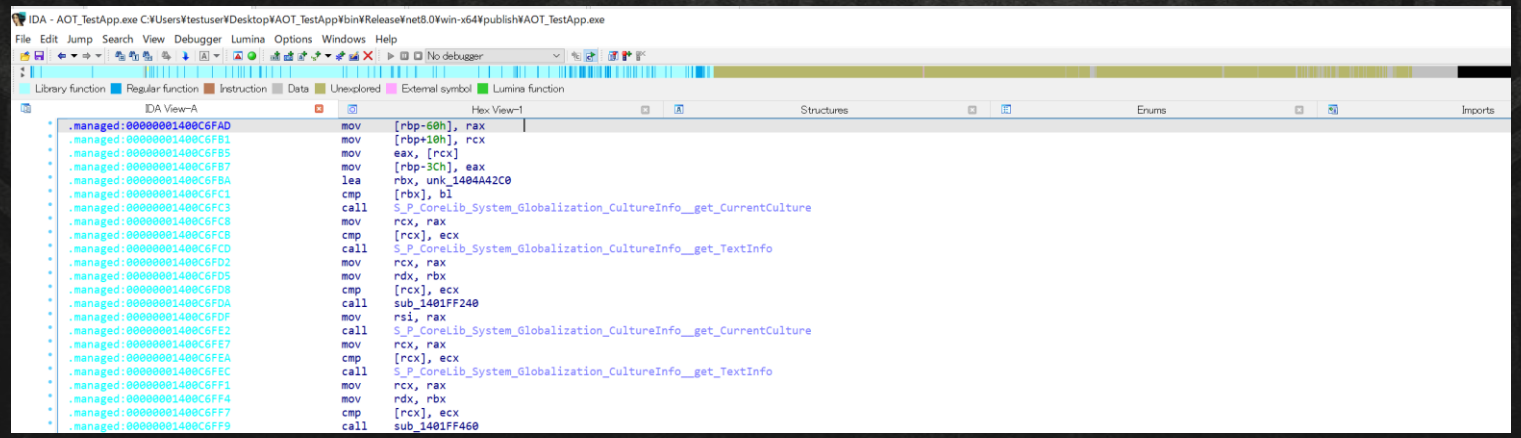
- File→Load file→FLIRT signature file...を選択し、先ほど格納したsigファイルを選択するとsignatureが適用される

NFLabs.

ハンズオン #3 補足

5. Before/After





ありがたい🥹

NFLabs.

マルウェア検体への適用

シグネチャを活かすためには、検体とシグネチャ生成のために利用したバイナリのTargetFrameworkのバージョン(.NETのバージョン)が一致している必要がある

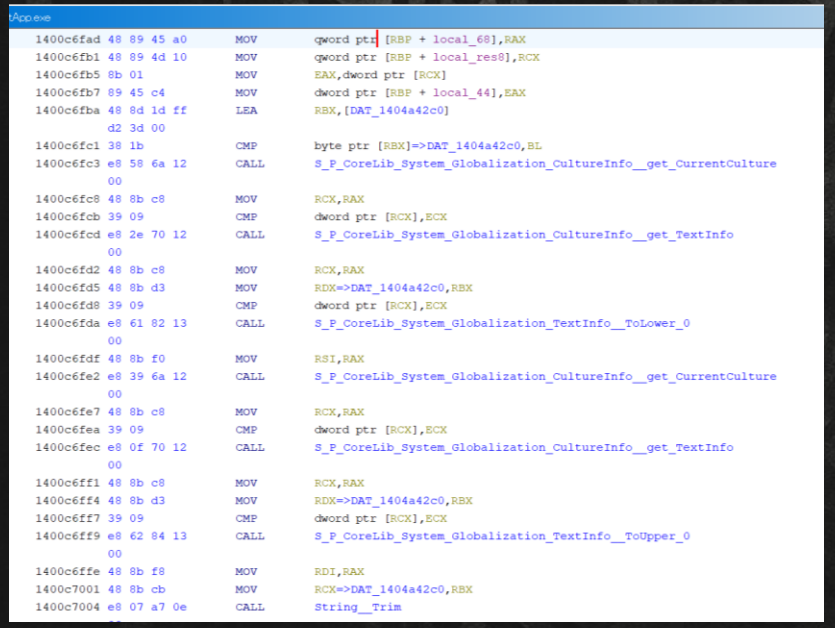
プロジェクトファイルの内容

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>          <!-- 実行形式をexeで指定-->
    <TargetFramework>net8.0</TargetFramework>     <!-- AOTバイナリに組み込む.NETライブラリのバージョンを指定-->
    <ImplicitUsings>enable</ImplicitUsings>       <!-- using の自動追加-->
    <PublishAot>True</PublishAot>          <!-- AOT発行の設定を有効化-->
    <DebugType>portable</DebugType>     <!-- PDB形式の指定 (後ほど使うのでportable指定してください-->
    <Optimize>True</Optimize>    <!-- 最適化を有効化(後の演習の結果が変わってくるのでTrueを指定してください)-->
  </PropertyGroup>
</Project>
```

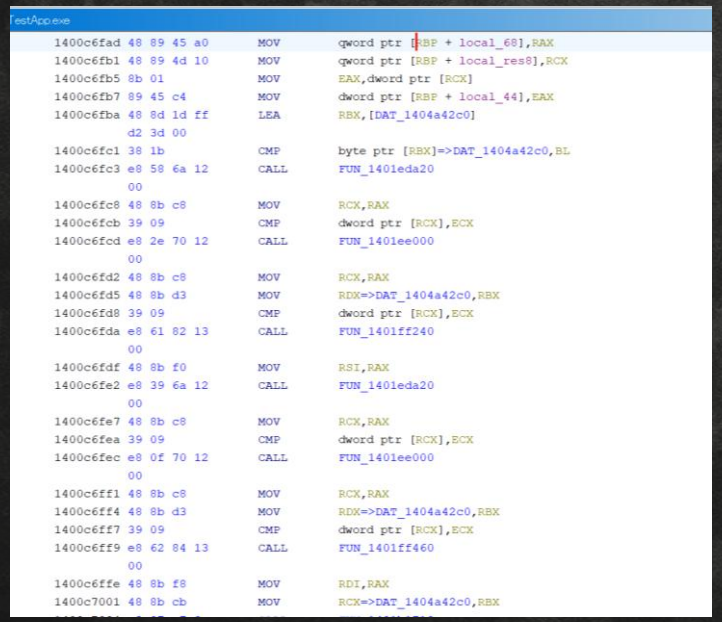
NFLabs.

マルウェア検体への適用

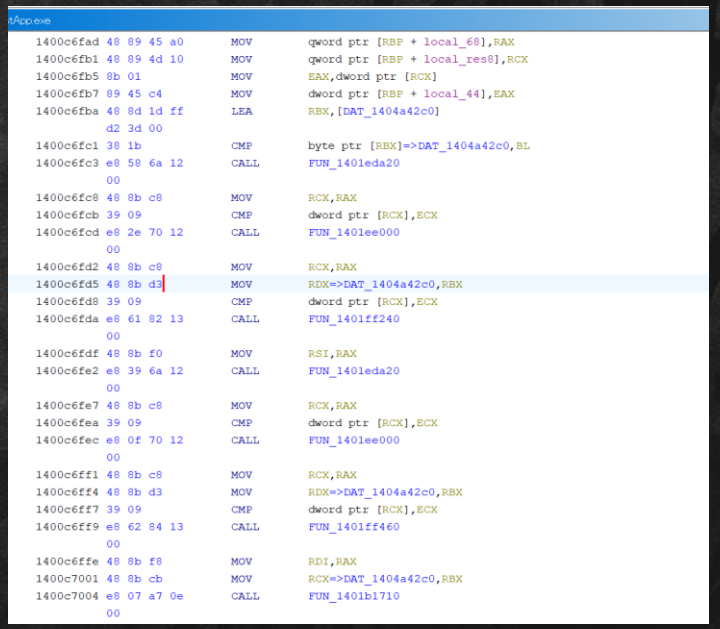
今まで使ってきているバイナリ(AOT_TestApp.exe)に対して、複数の.NETバージョンで生成したシグネチャを適用してみる



.NET8.0
(正解😊)



.NET9.0
(リネームできるシンボル数が少ない😢)



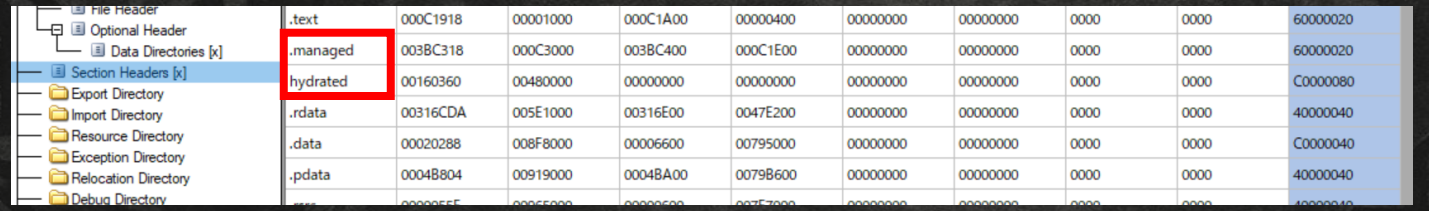
.NET10.0

NFLabs.

マルウェア検体への適用

Native AOTバイナリかどうかの推測

- .NET8.0の場合：
  - hydratedや.managedという名前のセクションが存在している
  - DotNetRuntimeDebugHeaderという名前のExportが存在している



- .NET9.0や10.0の場合：
  - DotNetRuntimeDebugHeaderという名前のExportが存在している

マルウェア検体への適用

組み込まれている.NETバージョンの推測

- stringsを確認する

```
C:¥Users¥testuser¥Desktop¥AOT_TestApp¥bin¥Release¥net8.0¥win-x64¥publish>strings AOT_TestApp.exe | Findstr NET
0.NETCoreApp,Version=v8.0
.NET
The System.Text.Json library is built-in as part of the shared framework in .NET Runtime. The package can be installed when yo
.NET 8.0
.NET TP Worker
.NET Long Running Task
```

- Ghidraでruntime_version文字列を参照しているサブルーチンの周辺を調査すると、.NETバージョンがみえるサブルーチンが存在する
(PDBを適用してみるとこれはRhGetRuntimeVersionと名前のつくサブルーチン)

```
1401a7aaa     TEST      EAX, EAX
1401a7aac     JZ        LAB_1401a7bba
1401a7ab2     LEA       R14, [s_runtime_version_1406dd330]          = "runtime_version"
1401a7ab9     LEA       RCX=local_30, [RSP + 0x48]
1401a7abe     CALL      FUN_14000b130                               undefined FUN_14000b130()
1401a7ac3     MOV       ECX, dword ptr [RSP + local_30]
1401a7ac7     TEST      ECX, ECX
```

```
                          FUN_14000b130                               XREF[1]:    FUN_1401a79e0:1401a7abe(c)
14000b130     MOV       dword ptr [RCX], 0x6
14000b136     LEA       RAX, [s_8.0.22_1407d8220]                   = "8.0.22"
14000b13d     RET
```

NFLabs.

ハンズオン #4

マルウェア検体にシグネチャを適用してみよう

使用するサンプルはこちら

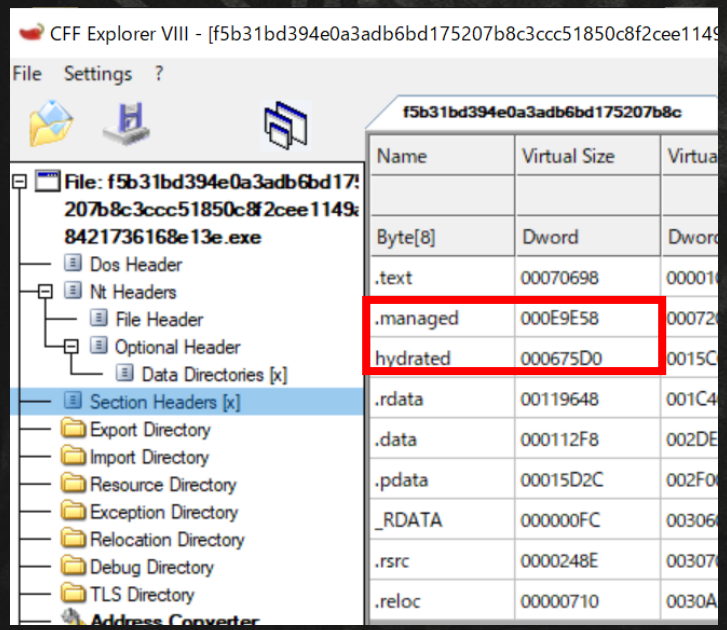
- 検体A: f5b31bd394e0a3adb6bd175207b8c3ccc51850c8f2cee1149a8421736168e13e
- 検体B: 9726e5c7f9800b36b671b064e89784fb10465210198fbbb75816224e85bd1306
- 検体C: 2c5d40e2bc5d03c9bbf314b6a18133bbd3ea655e0586deb61bc668d91f7959e5

ZIP PW: jsac2026

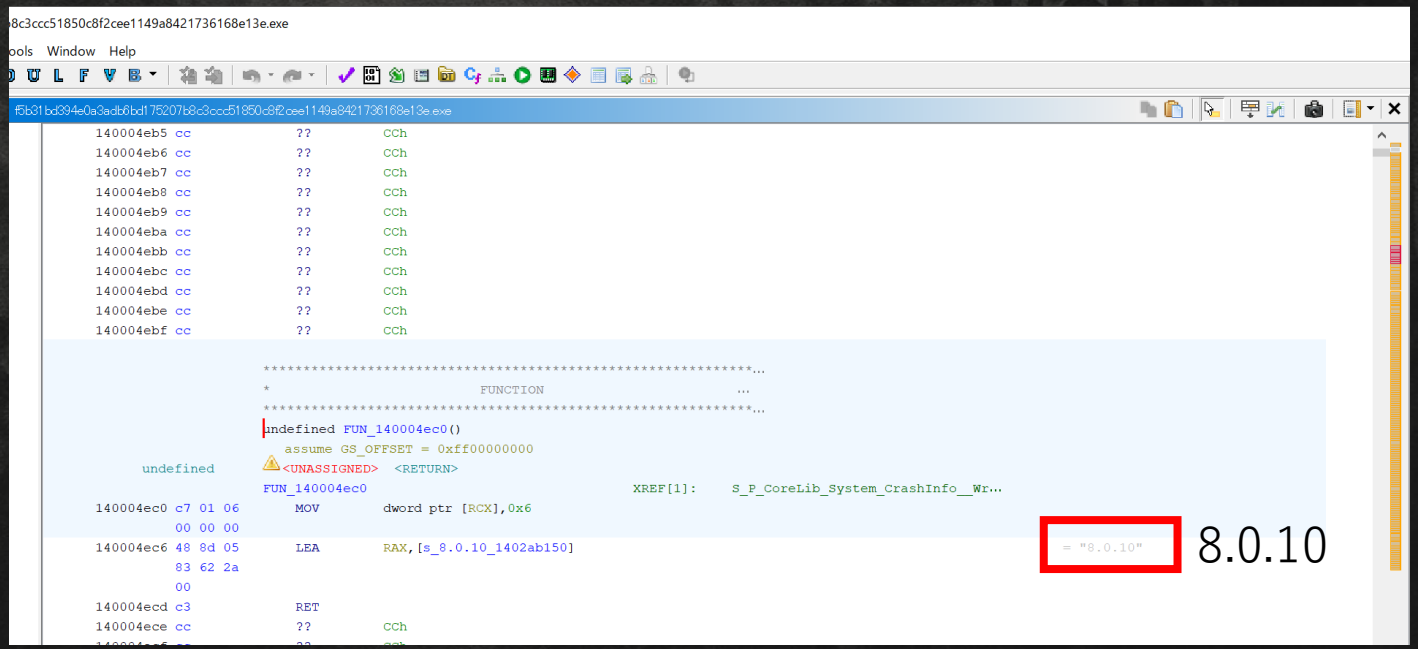
ハンズオン #4

検体Aにシグネチャを適用してみよう

- .NETバージョンの特定 → .NET8.0.10



Native AOTバイナリの可能性あり



.NETのバージョン情報を発見

NFLabs.

ハンズオン #4

検体Aにシグネチャを適用してみよう

- シグネチャの適用

```
140077a98 e8 73 ff ff ff    CALL    FUN_140077a10                                   undefined FUN_140077a10()
140077a9d 48 8d 4d a0       LEA     RCX=>local_68,[RBP + -0x60]
140077aa1 e8 4a b4 f8 ff    CALL    RhpPInvoke                                      undefined RhpPInvoke()
140077aa6 e8 f9 3f ff ff    CALL    KERNEL32.DLL::FreeConsole                       BOOL FreeConsole(void)
140077aab 48 8d 4d a0       LEA     RCX=>local_68,[RBP + -0x60]
140077aaf e8 8c b4 f8 ff    CALL    RhpPInvokeReturn                                undefined RhpPInvokeReturn()
140077ab4 90                NOP

                        LAB_140077ab5                                   XREF[1]:    140077c98(*)
140077ab5 48 8d 1d 5c 6d 0e 00  LEA     RBX,[DAT_14015e818]                         = ??
140077abc 48 83 c3 0c       ADD     RBX,0xc
140077ac0 48 8b cb          MOV     RCX=>DAT_14015e824,RBX                          = ??
140077ac3 ba 27 00 00 00    MOV     EDX,0x27
140077ac8 4c 8d 05 a1 6c 0e 00  LEA     R8,[DAT_14015e770]                          = ??
140077acf 49 83 c0 0c       ADD     R8=>DAT_14015e77c,0xc                           = ??
140077ad3 41 b9 08 00 00 00 MOV     R9D,0x8
140077ad9 e8 e2 6b 02 00    CALL    S_P_CoreLib_System_SpanHelpers__IndexOf_0       undefined S_P_CoreLib_System_Spa.
140077ade 85 c0             TEST    EAX,EAX
140077ae0 0f 8c 9f 00 00 00 JL      LAB_140077b85
140077ae6 48 8d 0d          LEA     RCX,[DAT_14017f3b8]                             = ??
```

ありがたい🥹
しかし、まだまだつかみどころがない…

NFLabs.

ハンズオン #4

早く終わった人は検体B, Cにもシグネチャを適用してみてください

検体C

```
1
2  void FUN_180079300(undefined8 param_1,undefined8 param_2,undefined8 param_3)
3
4  {
5    undefined8 uVar1;
6
7    uVar1 = S_P_CoreLib_System_Convert__FromBase64String();
8    uVar1 = FUN_180079c30(uVar1,param_2,param_3);
9    if (DAT_1801516a8 != 0) {
10     __GetGCStaticBase_S_P_CoreLib_System_Text_UTF8Encoding();
11   }
12   Class_18015f6a8::Method_15(*(Class_18015f6a8 **)(DAT_1802946d8 + 8),uVar1);
13   return;
14 }
15
```

NFLabs.

前半まとめ

- Native AOTでコンパイルされたバイナリとそれ以外の方法でコンパイルされたバイナリの解析方法は大きく異なる
- バイナリには大量の標準ライブラリが含まれるため、シグネチャを使ったユーザー定義のコードとの分離が有効
- シグネチャを作成して解析対象のバイナリに適用した

NFLabs.

~Break Time~

NFLabs.

Re:birth the fidb:
Reverse Engineering the .NET AOT Malware
section #2

はじめに

- シグネチャの適用により標準ライブラリとユーザーコードを分離には成功した
- しかし、文字列リテラルへの参照が失われてしまっているためコードの可読性が低い
- デバッグ実行によって文字列リテラルへの参照を復元することは可能だが、解析効率が悪い(手間がかかる)

NFLabs.

はじめに

- 以下のように、String型のオブジェクトへのポインタがunk_XXXとなっていて、”Hello, World!”などの文字列が直接的には判別できない

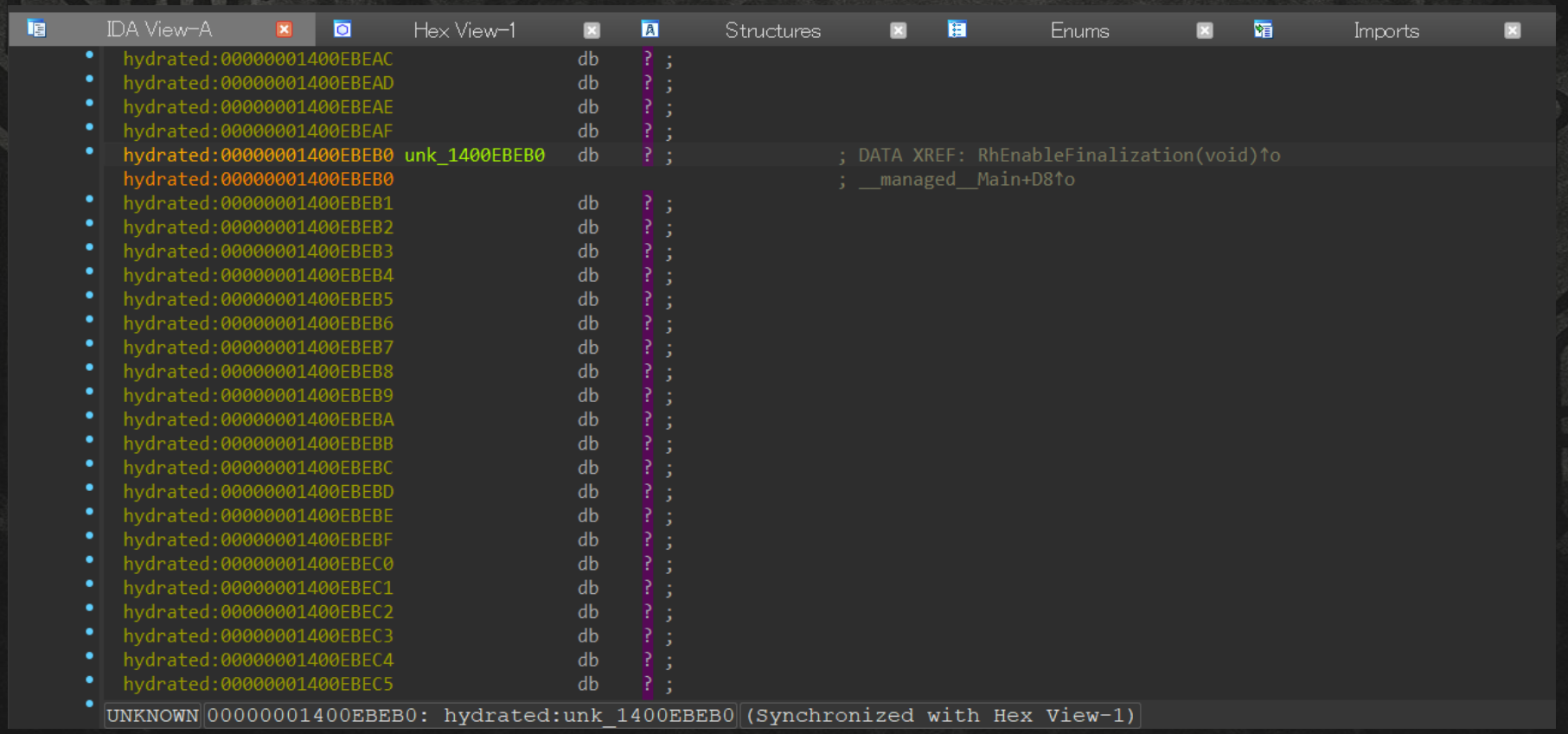
```
13
14  class Program
15  {
16      static void Main(string[] args)
17      {
18          Console.WriteLine("Hello, World!");
19      }
20  }
```

```
.managed:00000001400D26EC
.managed:00000001400D26EC loc_1400D26EC:
.managed:00000001400D26EC cmp     [rcx], cl
.managed:00000001400D26EE mov     edx, 1
.managed:00000001400D26F3 mov     r8d, 1
.managed:00000001400D26F9 call    S_P_CoreLib_System_Threading_Thread__SetApartmentState_0
.managed:00000001400D26FE call    S_P_CoreLib_Internal_Runtime_CompilerHelpers_StartupCodeHelpers__RunModul
.managed:00000001400D2703 call    sub_1400B5F70
.managed:00000001400D2708 lea     rcx, unk_1400EBEB0
.managed:00000001400D270F call    sub_1400721A0
.managed:00000001400D2714 call    sub_140095400
.managed:00000001400D2719 call    S_P_CoreLib_System_AppContext__OnProcessExit
.managed:00000001400D271E lea     rbx, unk_1400F63A0
.managed:00000001400D2725 cmp     qword ptr [rbx-8], 0
.managed:00000001400D272A jnz     short loc_1400D2741
```

NFLabs.

はじめに

- unk_XXXの参照先は実行時に初期化される領域



はじめに

- デバッグ実行で解析を進めると、参照先の具体的な文字列を特定(ただし参照先を都度確認する手間がかかる)

IDA View-RIP

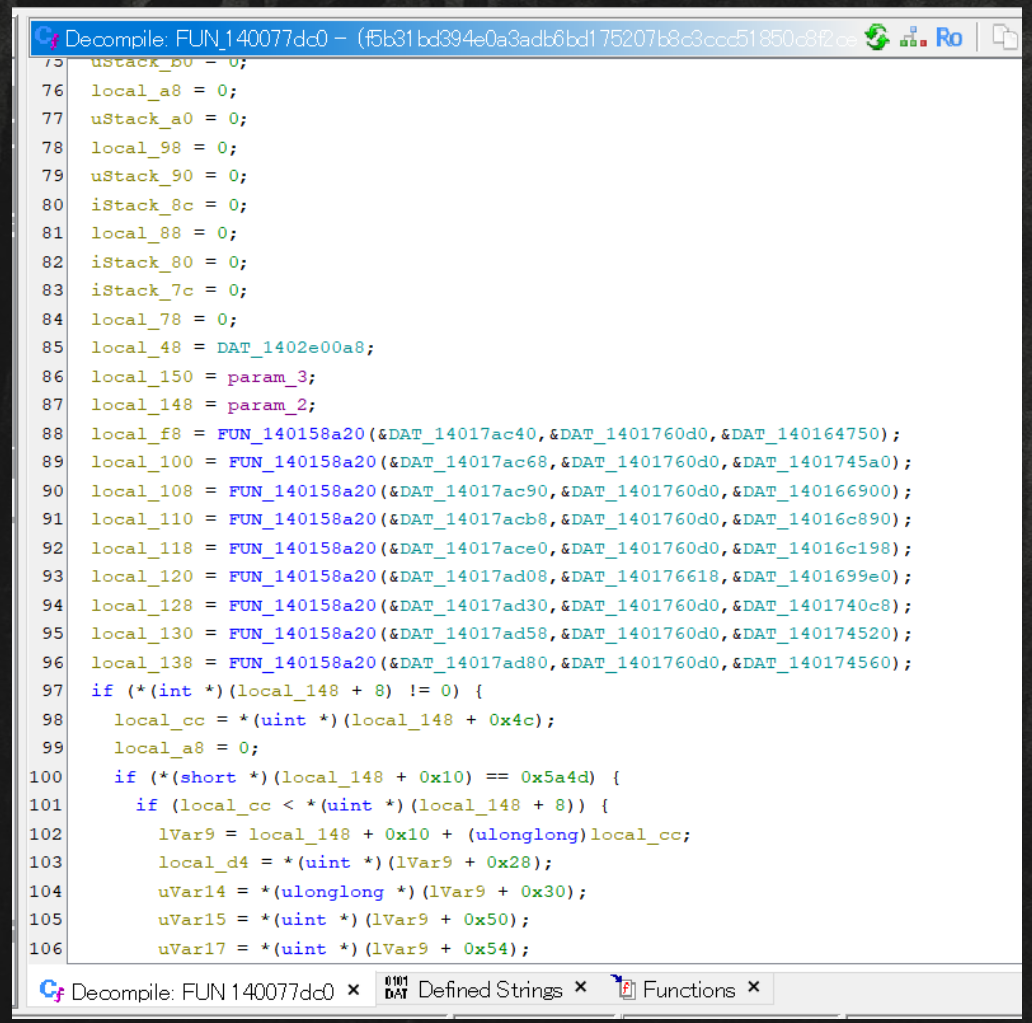
```
hydrated:00007FF607C7BEAC db    0
hydrated:00007FF607C7BEAD db    0
hydrated:00007FF607C7BEAE db    0
hydrated:00007FF607C7BEAF db    0
RCX hydrated:00007FF607C7BEB0 off_7FF607C7BEB0 dq offset dword_7FF607C881E8
hydrated:00007FF607C7BEB0                                       ; DATA XREF: RhEnableFinalization(void)↑o
hydrated:00007FF607C7BEB0                                       ; __managed__Main+D8↑o
hydrated:00007FF607C7BEB8 dd 0Dh
hydrated:00007FF607C7BEBC aHelloWorld:
hydrated:00007FF607C7BEBC text "UTF-16LE", 'Hello, World!',0
hydrated:00007FF607C7BED8 db    0
hydrated:00007FF607C7BED9 db    0
hydrated:00007FF607C7BEDA db    0
hydrated:00007FF607C7BEDB db    0
hydrated:00007FF607C7BEDC db    0
hydrated:00007FF607C7BEDD db    0
hydrated:00007FF607C7BEDE db    0
hydrated:00007FF607C7BEDF db    0
```

はじめに

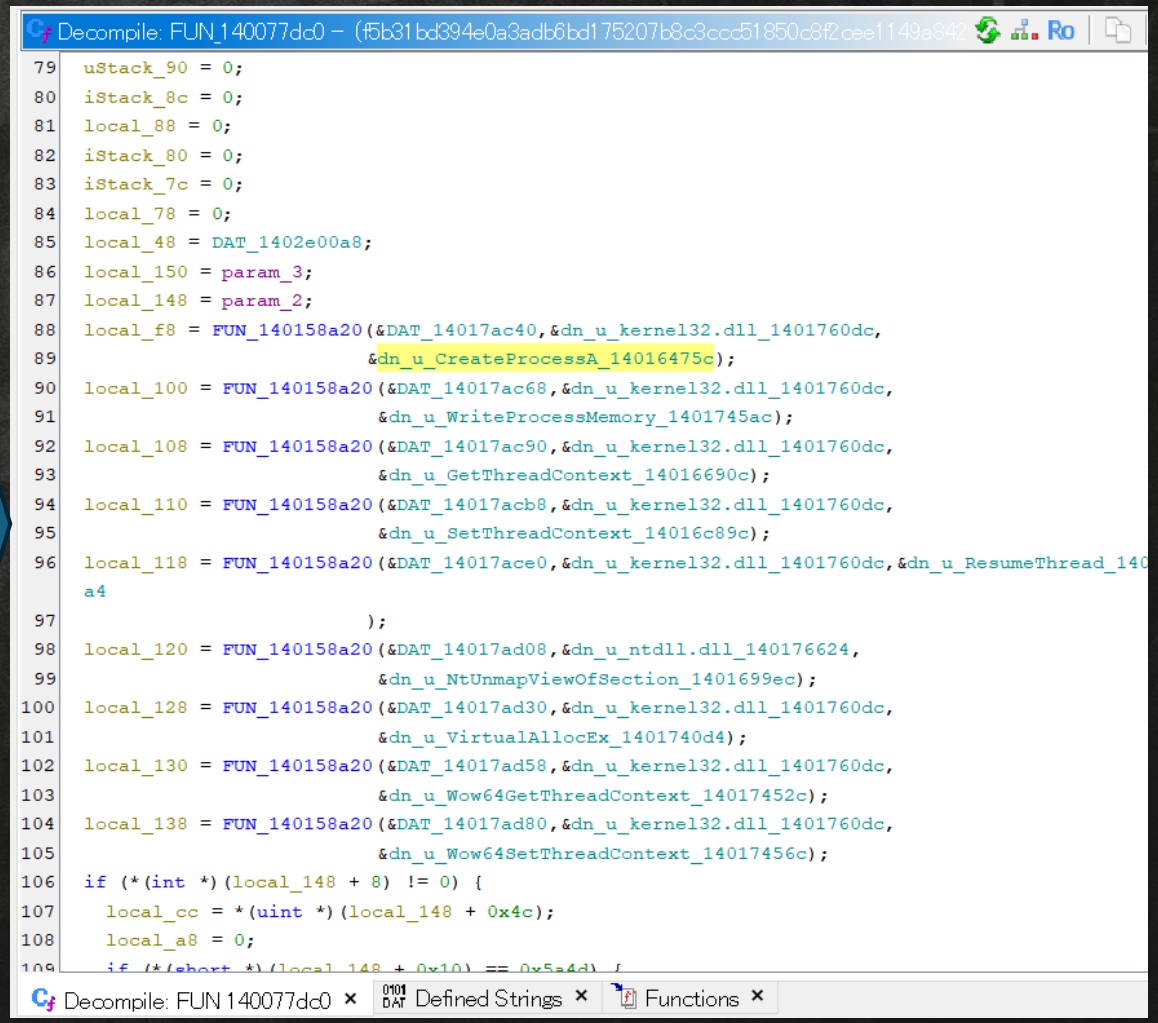
- 本セクションでは、Native AOTのコンパイル時の特徴的な動作(データのhydrateなど)の概要を学び、静的解析の可読性を向上させる手法を習得する
  - オブジェクトのメタデータ等を静的に復元し、シンボルを自動リネームするGhidraプラグインの活用方法を実践する
- ※ Native AOTの動作解説は.NET 8をベースとしている

プラグイン適用による解析結果の比較

NFLabs.

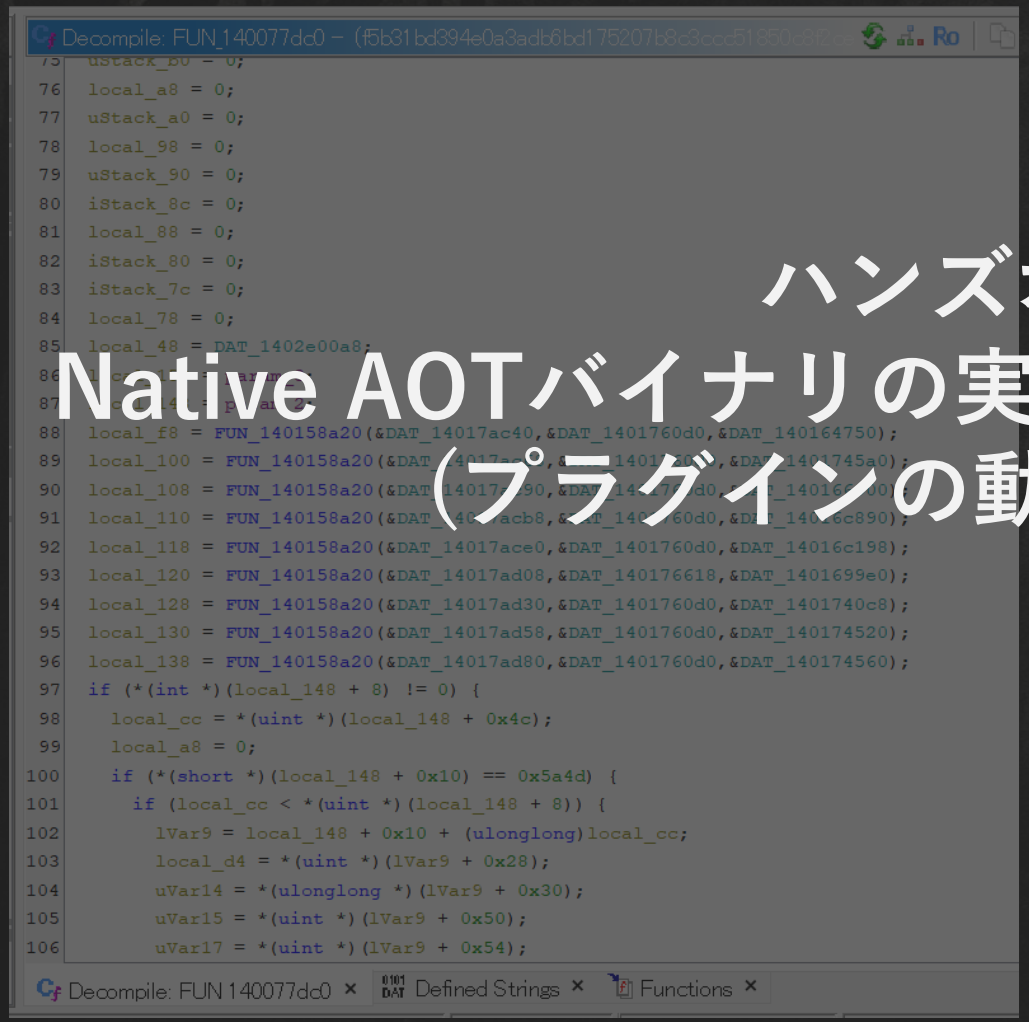


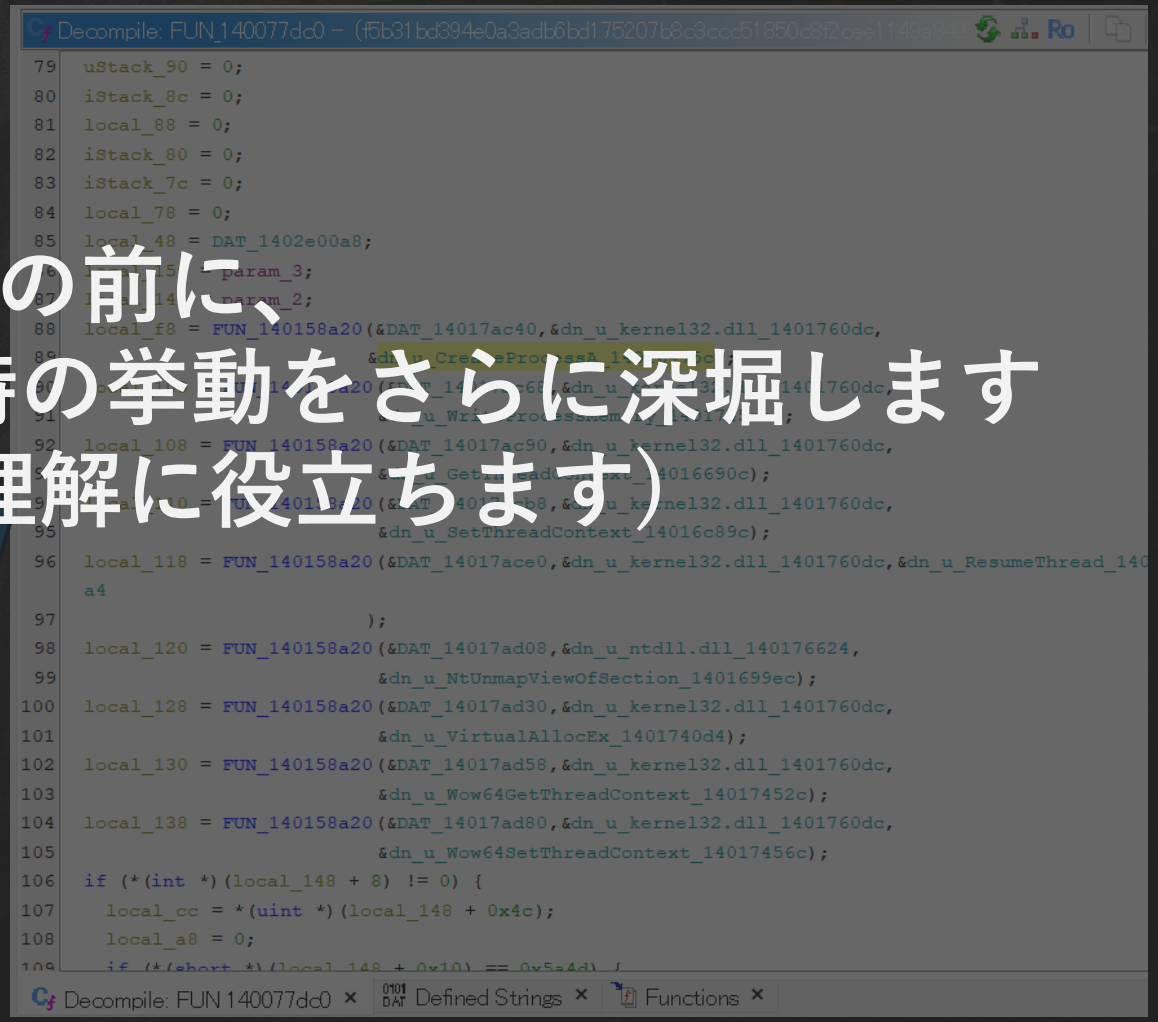

```
uStack_b0 = 0;
local_a8 = 0;
uStack_a0 = 0;
local_98 = 0;
uStack_90 = 0;
iStack_8c = 0;
local_88 = 0;
iStack_80 = 0;
iStack_7c = 0;
local_78 = 0;
local_48 = DAT_1402e00a8;
local_150 = param_3;
local_148 = param_2;
local_f8 = FUN_140158a20(&DAT_14017ac40,&DAT_1401760d0,&DAT_140164750);
local_100 = FUN_140158a20(&DAT_14017ac68,&DAT_1401760d0,&DAT_1401745a0);
local_108 = FUN_140158a20(&DAT_14017ac90,&DAT_1401760d0,&DAT_140166900);
local_110 = FUN_140158a20(&DAT_14017acb8,&DAT_1401760d0,&DAT_14016c890);
local_118 = FUN_140158a20(&DAT_14017ace0,&DAT_1401760d0,&DAT_14016c198);
local_120 = FUN_140158a20(&DAT_14017ad08,&DAT_140176618,&DAT_1401699e0);
local_128 = FUN_140158a20(&DAT_14017ad30,&DAT_1401760d0,&DAT_1401740c8);
local_130 = FUN_140158a20(&DAT_14017ad58,&DAT_1401760d0,&DAT_140174520);
local_138 = FUN_140158a20(&DAT_14017ad80,&DAT_1401760d0,&DAT_140174560);
if (*(int *)(local_148 + 8) != 0) {
  local_cc = *(uint *)(local_148 + 0x4c);
  local_a8 = 0;
  if (*(short *)(local_148 + 0x10) == 0x5a4d) {
    if (local_cc < *(uint *)(local_148 + 8)) {
      lVar9 = local_148 + 0x10 + (ulonglong)local_cc;
      local_d4 = *(uint *)(lVar9 + 0x28);
      uVar14 = *(ulonglong *)(lVar9 + 0x30);
      uVar15 = *(uint *)(lVar9 + 0x50);
      uVar17 = *(uint *)(lVar9 + 0x54);
```




```
uStack_90 = 0;
iStack_8c = 0;
local_88 = 0;
iStack_80 = 0;
iStack_7c = 0;
local_78 = 0;
local_48 = DAT_1402e00a8;
local_150 = param_3;
local_148 = param_2;
local_f8 = FUN_140158a20(&DAT_14017ac40,&dn_u_kernel32.dll_1401760dc,
                         &dn_u_CreateProcessA_14016475c);
local_100 = FUN_140158a20(&DAT_14017ac68,&dn_u_kernel32.dll_1401760dc,
                          &dn_u_WriteProcessMemory_1401745ac);
local_108 = FUN_140158a20(&DAT_14017ac90,&dn_u_kernel32.dll_1401760dc,
                          &dn_u_GetThreadContext_14016690c);
local_110 = FUN_140158a20(&DAT_14017acb8,&dn_u_kernel32.dll_1401760dc,
                          &dn_u_SetThreadContext_14016c89c);
local_118 = FUN_140158a20(&DAT_14017ace0,&dn_u_kernel32.dll_1401760dc,&dn_u_ResumeThread_140
a4
                         );
local_120 = FUN_140158a20(&DAT_14017ad08,&dn_u_ntdll.dll_140176624,
                          &dn_u_NtUnmapViewOfSection_1401699ec);
local_128 = FUN_140158a20(&DAT_14017ad30,&dn_u_kernel32.dll_1401760dc,
                          &dn_u_VirtualAllocEx_1401740d4);
local_130 = FUN_140158a20(&DAT_14017ad58,&dn_u_kernel32.dll_1401760dc,
                          &dn_u_Wow64GetThreadContext_14017452c);
local_138 = FUN_140158a20(&DAT_14017ad80,&dn_u_kernel32.dll_1401760dc,
                          &dn_u_Wow64SetThreadContext_14017456c);
if (*(int *)(local_148 + 8) != 0) {
  local_cc = *(uint *)(local_148 + 0x4c);
  local_a8 = 0;
```

プラグイン適用による解析結果の比較





ハンズオンの前に、
Native AOTバイナリの実行時の挙動をさらに深堀します
（プラグインの動作理解に役立ちます）

NFLabs.

Native AOT

Native AOTでプログラムをコンパイルすると、、

- オブジェクトの型に関するメタデータや、オブジェクト(Frozen Object)を圧縮した形でバイナリ内に格納する(**Dehydrate**)
- プログラム実行時に、バイナリ内の圧縮データを展開することで、メタデータやオブジェクトへの参照を復元する(**Rehydrate**)

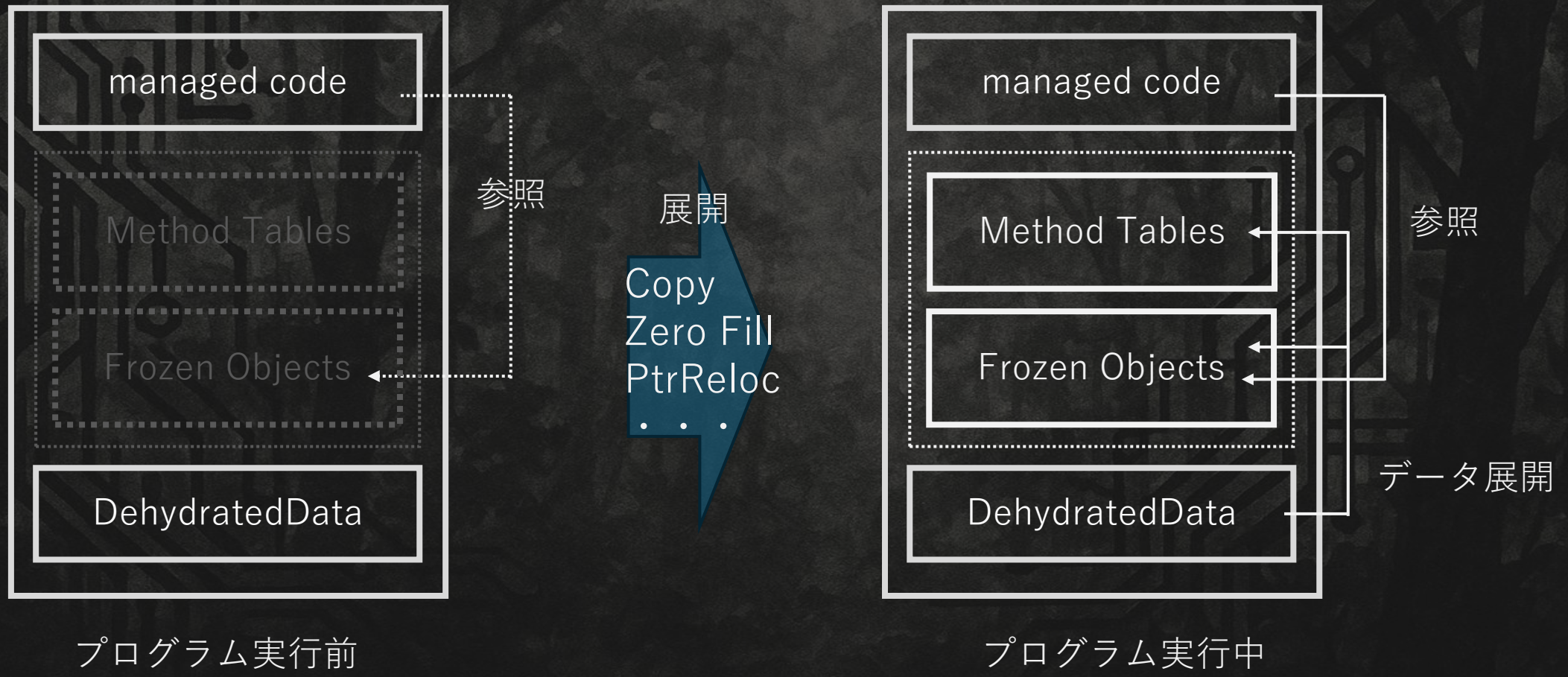
メタデータ　：型情報やVTable等を管理するMethod Table

Frozen Object: コンパイル時に生成される静的なオブジェクト(メタデータと同じく、圧縮した状態でバイナリに配置される)

NFLabs.

Native AOT

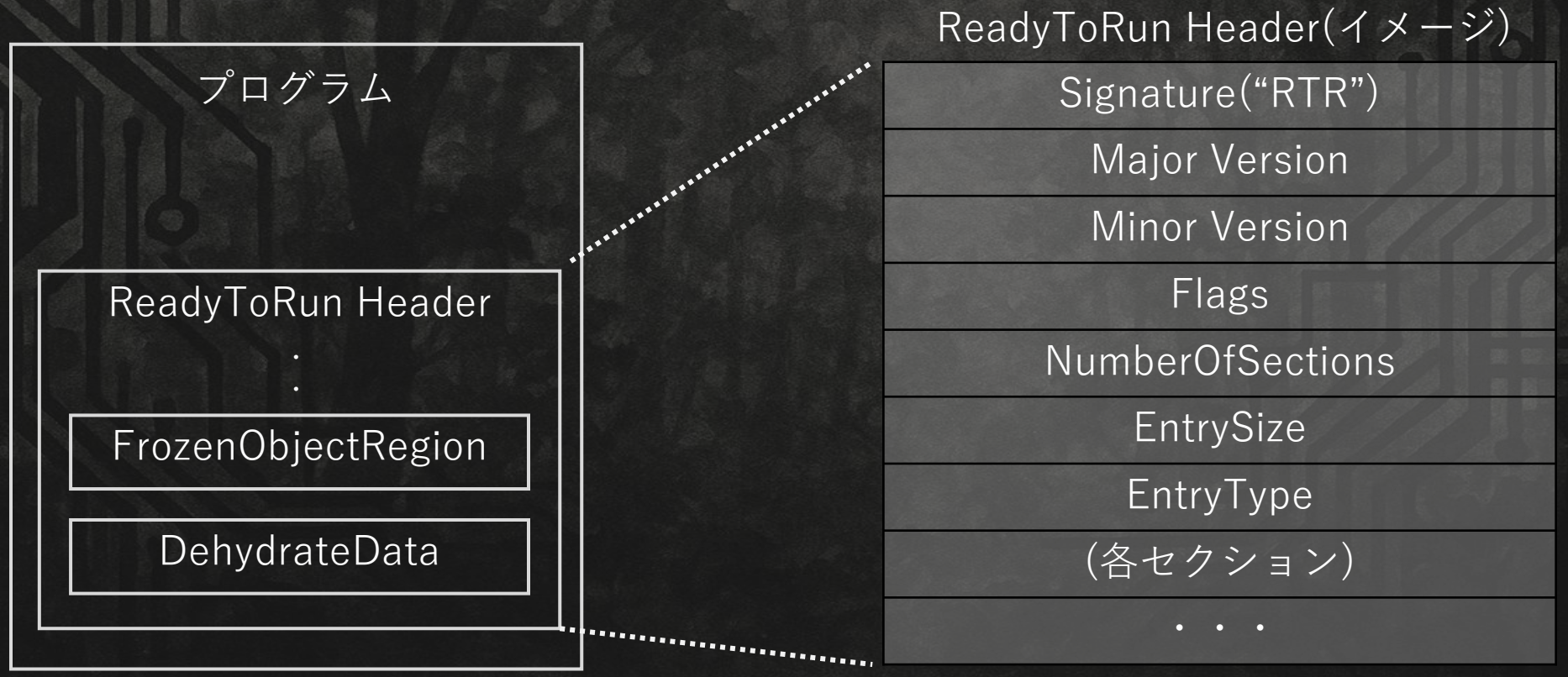
実行時におけるデータの展開(Rehydrate)プロセスのイメージ



NFLabs.

ReadyToRun

Native AOTバイナリ内にはReadyToRunヘッダーが含まれる
実行時にはランタイムのコードがこのセクションを起点にメタデータを展開



NFLabs.

ReadyToRun

ReadyToRunヘッダー配下の各エントリのうちの１つがDehydratedDataを指し、圧縮されたデータを参照

Signature(“RTR”)
Major Version
Minor Version
Flags
NumberOfSections
EntrySize
EntryType
(各セクションが続く)
DehydratedData
・・・

DehydratedDataのデータ構造例*

SectionType
Flag
Start pointer
End pointer

* 各メンバは実際の名称ではない

NFLabs.

ReadyToRun

ReadyToRun Headerの後続のDehydrateDataから、圧縮されたデータを参照

IDA View-RIP

```
.rdata:00007FF61D72BCF7 db    0
.rdata:00007FF61D72BCF8 aRtr db 'RTR',0                 ; DATA XREF: debug065:0000028
.rdata:00007FF61D72BCF8                                 ; .rdata:00007FF61D6DB6B0↑o
.rdata:00007FF61D72BCFC dw 9
.rdata:00007FF61D72BCFE dw 1
.rdata:00007FF61D72BD00 dd 0                            ; Flags
.rdata:00007FF61D72BD04 dw 22h                          ; Nunberofsections
.rdata:00007FF61D72BD06 db  18h                         ; Type
.rdata:00007FF61D72BD07 db    1                         ; Type
.rdata:00007FF61D72BD08 dd 201                          ; GCStaticRegion
.rdata:00007FF61D72BD0C dd 1
.rdata:00007FF61D72BD10 dq offset unk_7FF61D74C010
.rdata:00007FF61D72BD18 dq offset unk_7FF61D74C234
.rdata:00007FF61D72BD20 dd 202                          ; ThreadStaticRegion
.rdata:00007FF61D72BD24 dd 1
.rdata:00007FF61D72BD28 dq offset off_7FF61D74C238
.rdata:00007FF61D72BD30 dq offset unk_7FF61D74C240
.rdata:00007FF61D72BD38 dd 204
.rdata:00007FF61D72BD3C dd 0
.rdata:00007FF61D72BD40 dq offset unk_7FF61D74C240
.rdata:00007FF61D72BD48 dq 0
.rdata:00007FF61D72BD50 dd 205
.rdata:00007FF61D72BD54 dd 1
.rdata:00007FF61D72BD58 dq offset unk_7FF61D74C000
```

（各セクション）

DehydrateData

・・・

```
// NativeAOT ReadyToRun sections
//
StringTable = 200, // Unused
GCStaticRegion = 201,
ThreadStaticRegion = 202,
// Unused = 203,
TypeManagerIndirection = 204,
EagerCctor = 205,
FrozenObjectRegion = 206,
DehydratedData = 207,
ThreadStaticOffsetRegion = 208,
// 209 is unused - it was used by ThreadStaticGCDescRegion
// 210 is unused - it was used by ThreadStaticIndex
// 211 is unused - it was used by LoopHijackFlag
ImportAddressTables = 212,
ModuleInitializerList = 213,
```

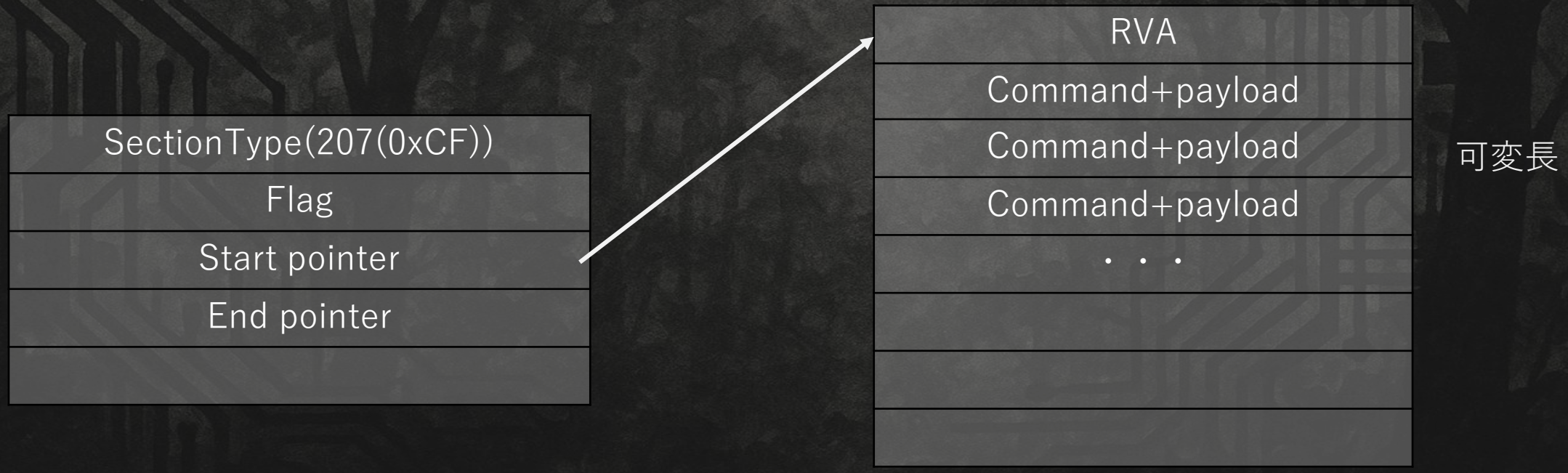
ランタイムソースコードにおけるセクションの定義

https://github.com/dotnet/runtime/blob/main/src/coreclr/tools/Common/Internal/Runtime/ModuleHeaders.cs

NFLabs.

Dehydrate/Rehydrate

DehydratedDataセクションのStart pointer(offset0x08)が指す先に圧縮されたデータの展開用コードが格納されている



イメージ図

NFLabs.

Dehydrate/Rehydrate

- 展開コードはcommandの種類とcommand データ(payload)から構成される
- payloadの大きさによって可変長のデータ構造

命令	値	説明
Copy	0x00	データのコピー
Zero Fill	0x01	0で初期化
RelPtr32Reloc	0x02	再配置(REL_BASED_RELPTR32)　payloadがRVA
PtrReloc	0x03	再配置(REL_BASED_HIGHLOW/REL_BASED_DIR64)　絶対アドレス
InlineRelPtr32Reloc	0x04	再配置(REL_BASED_RELPTR32)、payloadがRVA
InlinePtrReloc	0x05	再配置(REL_BASED_HIGHLOW/REL_BASED_DIR64)、絶対アドレス

https://github.com/dotnet/runtime/blob/main/src/coreclr/tools/Common/Internal/Runtime/DehydratedData.cs

NFLabs.

Dehydrate/Rehydrate

エンコードロジックの実装例(DehydratedData.csより抜粋)

```
42
43          public static int Encode(int command, int commandData, byte[] buffer)
44          {
45              Debug.Assert((command & DehydratedDataCommandMask) == command);
46              int remainingData = commandData - MaxShortPayload;
47              if (remainingData <= 0)
48              {
49                  buffer[0] = EncodeShort(command, commandData);
50                  return 1;
51              }
52
53              int numExtraBytes = 0;
54              for (; remainingData != 0; remainingData >>= 8)
55                  buffer[++numExtraBytes] = (byte)remainingData;
56              if (numExtraBytes > MaxExtraPayloadBytes)
57                  throw new InvalidOperationException(); // decoder can only decode this many extra bytes
58
59              buffer[0] = (byte)(command | ((MaxShortPayload + numExtraBytes) << DehydratedDataCommandPayloadShift));
60              return 1 + numExtraBytes;
61          }
```

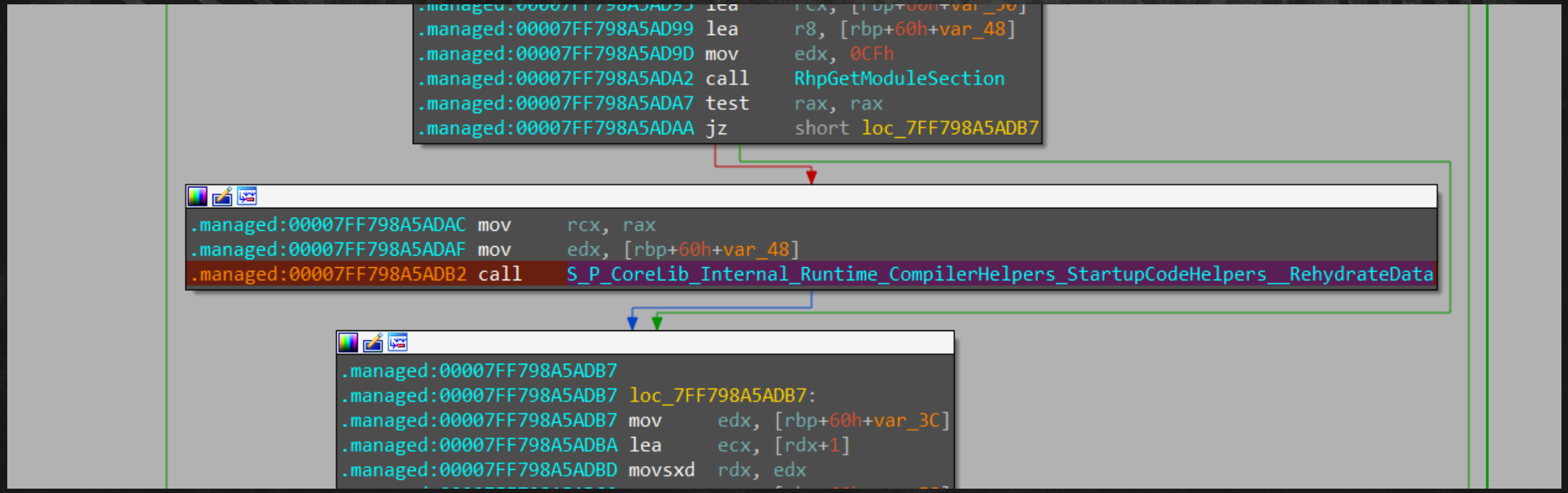
データ長に応じた可変長エンコード

https://github.com/dotnet/runtime/blob/main/src/coreclr/tools/Common/Internal/Runtime/DehydratedData.cs

NFLabs.

Dehydrate/Rehydrate

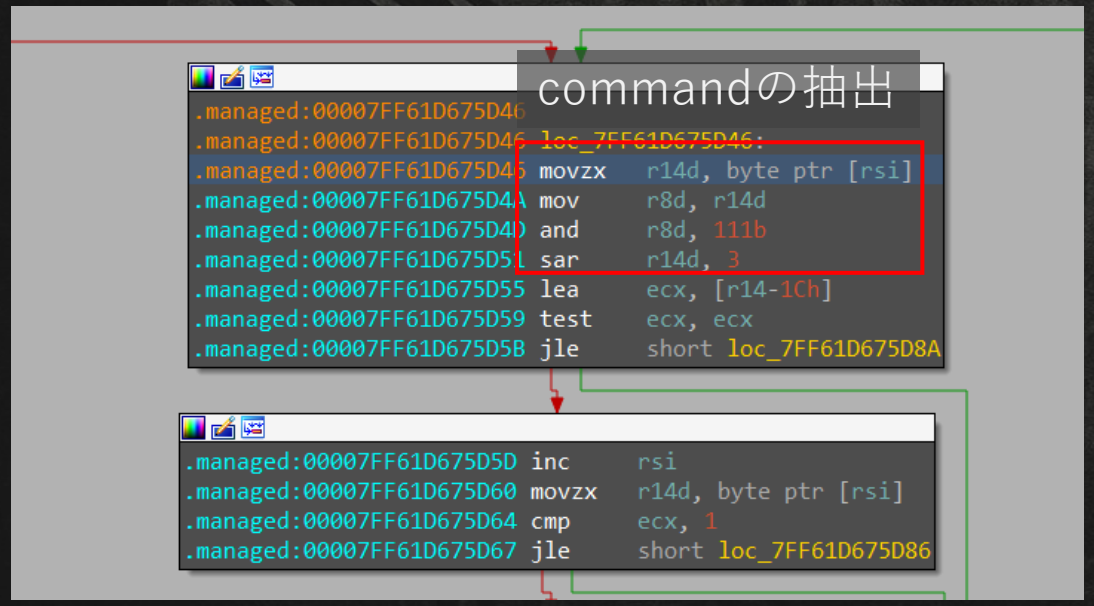
- デバッガによるランタイムのデータ展開処理のトレース
- RehydrateDataがメタデータを展開する中核となるルーチン

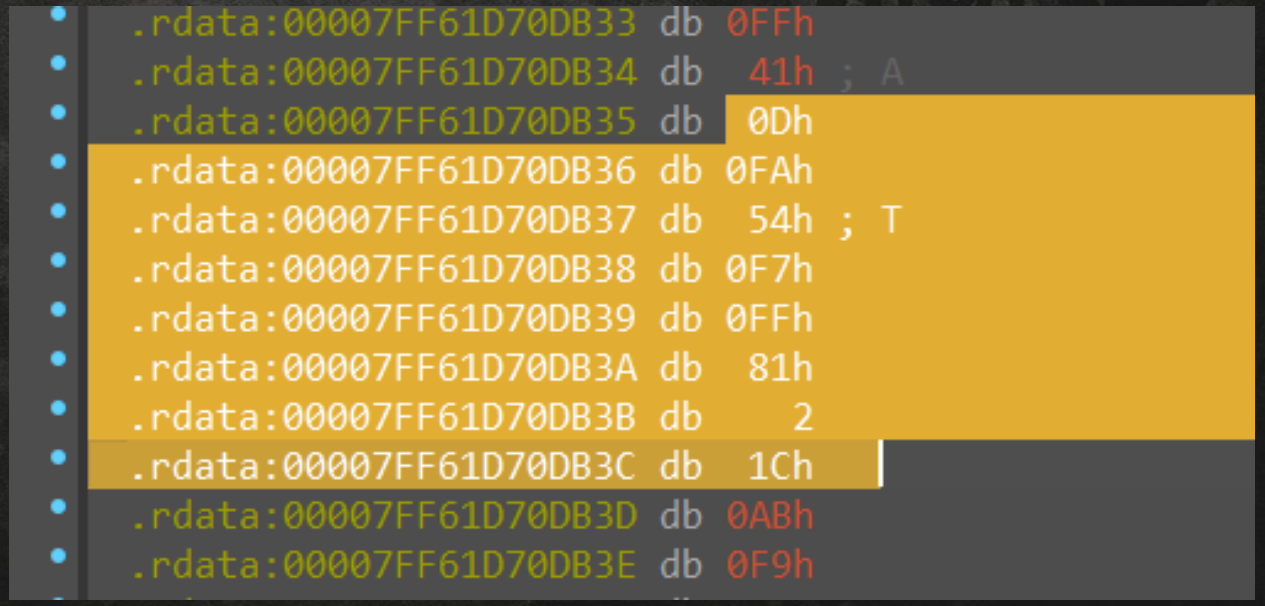


NFLabs.

Dehydrate/Rehydrate

- Dehydrated Dataの先頭1byteから命令の種類を識別するロジック





NFLabs.

Dehydrate/Rehydrate

- 命令コード(command)に基づく、処理の分岐(Switch-Case)

```
.managed:00007FF61D675D8A loc_7FF61D675D8A:                    ; CODE XREF: S_P_CoreLib_Internal_Runtime_CompilerHelpers_StartupCod
.managed:00007FF61D675D8A inc     rsi
.managed:00007FF61D675D8D cmp     r8d, 5                       ; switch 6 cases
.managed:00007FF61D675D91 ja      def_7FF61D675DAF             ; jumptable 00007FF61D675DAF default case
.managed:00007FF61D675D97 mov     r8d, r8d
.managed:00007FF61D675D9A lea     rcx, jpt_7FF61D675DAF
.managed:00007FF61D675DA1 mov     ecx, ds:(jpt_7FF61D675DAF - 7FF61D70D058h)[rcx+r8*4]
.managed:00007FF61D675DA5 lea     rdx, loc_7FF61D675D2A
.managed:00007FF61D675DA5                                      ;
.managed:00007FF61D675DA5                                      ; rcx: 縮退データの先頭アドレス
.managed:00007FF61D675DA5                                      ; [rcx]:(RVA)
.managed:00007FF61D675DAC add     rcx, rdx
.managed:00007FF61D675DAF jmp     rcx                          ; switch jump
.managed:00007FF61D675DB1 ; ---------------------------------------------------------------------------
.managed:00007FF61D675DB1
.managed:00007FF61D675DB1 loc_7FF61D675DB1:                    ; CODE XREF: S_P_CoreLib_Internal_Runtime_CompilerHelpers_StartupCod
.managed:00007FF61D675DB1                                      ; DATA XREF: .rdata:jpt_7FF61D675DAF↓o
.managed:00007FF61D675DB1 cmp     r14d, 4                      ; jumptable 00007FF61D675DAF case 0
.managed:00007FF61D675DB5 jge     short loc_7FF61D675DD8
.managed:00007FF61D675DB7 movzx   r8d, byte ptr [rsi]
.managed:00007FF61D675DBB mov     [rbx], r8b
.managed:00007FF61D675DBE cmp     r14d, 1
.managed:00007FF61D675DC2 jle     short loc_7FF61D675E22
```

command(0x0~0x5)のチェック

Dehydrate/Rehydrate

- 例(Copyの場合)

48	E0	FF	FF	FF	FF	FF	FF	FF	08

DehydratedDataの先頭1byteを取り出す

48

下位3bitを命令コードとして抽出、
0x48 and 0x07 → 0x00
上位5bitを取り出して、
0x48 sar 0x03 → 0x09

Case0(0x00) == Copy
0x9　　== 9bytes分
すなわち、2bytes目以降の9バイト分を
hydrateセクション領域に展開する、という
処理になる

NFLabs.

Dehydrate/Rehydrate

- 例(Copyの場合)
  - Dehydratred Data(.rdata等)からhydrateセクションへのデータ展開

```
hydrated:00007FF61D6A6020 db    0
hydrated:00007FF61D6A6021 db    0
hydrated:00007FF61D6A6022 db    0
hydrated:00007FF61D6A6023 db    0
hydrated:00007FF61D6A6024 db    0
hydrated:00007FF61D6A6025 db    0
hydrated:00007FF61D6A6026 db    0
hydrated:00007FF61D6A6027 db    0
hydrated:00007FF61D6A6028 db    0
hydrated:00007FF61D6A6029 db    0
hydrated:00007FF61D6A602A db    0
hydrated:00007FF61D6A602B db    0
hydrated:00007FF61D6A602C db    0
```

```
hydrated:00007FF61D6A6020 db 0E0h
hydrated:00007FF61D6A6021 db 0FFh
hydrated:00007FF61D6A6022 db 0FFh
hydrated:00007FF61D6A6023 db 0FFh
hydrated:00007FF61D6A6024 db 0FFh
hydrated:00007FF61D6A6025 db 0FFh
hydrated:00007FF61D6A6026 db 0FFh
hydrated:00007FF61D6A6027 db 0FFh
hydrated:00007FF61D6A6028 db    8
hydrated:00007FF61D6A6029 db    0
hydrated:00007FF61D6A602A db    0
hydrated:00007FF61D6A602B db    0
hydrated:00007FF61D6A602C db    0
```

NFLabs.

Dehydrate/Rehydrate

- 例(Zero Fillの場合)

21	20	5F	FA	9F	FA	02	0C	DA

DehydratedDataの先頭1byteを取り出す。

21

下位3bitを命令とすると、
0x21 and 0x7 → 0x01
上位5bitを取り出して、
0x21 sar 3 → 0x04

Case1(0x01) == Zero Fill
0x4　　== 4bytes分
すなわち、hydrateセクションを4バイト分、ゼロで埋める、という処理になる。

（1バイトの命令から、4byte分のデータを展開したことになる。）

NFLabs.

Dehydrate/Rehydrate

- 例(Zero Fillの場合)
  - hydrateセクションの所定の位置から4バイト分を0で埋める(見かけ上は変わらず)

```
hydrated:00007FF61D6A6010 db    0
hydrated:00007FF61D6A6011 db    0
hydrated:00007FF61D6A6012 db    0
hydrated:00007FF61D6A6013 db    0
hydrated:00007FF61D6A6014 db    0
hydrated:00007FF61D6A6015 db    0
hydrated:00007FF61D6A6016 db    0
hydrated:00007FF61D6A6017 db    0
hydrated:00007FF61D6A6018 db 0
hydrated:00007FF61D6A6019 db 0
hydrated:00007FF61D6A601A db 0
hydrated:00007FF61D6A601B db 0
```

```
hydrated:00007FF61D6A6010 db    0
hydrated:00007FF61D6A6011 db    0
hydrated:00007FF61D6A6012 db    0
hydrated:00007FF61D6A6013 db    0
hydrated:00007FF61D6A6014 db    0
hydrated:00007FF61D6A6015 db    0
hydrated:00007FF61D6A6016 db    0
hydrated:00007FF61D6A6017 db    0
hydrated:00007FF61D6A6018 db 0
hydrated:00007FF61D6A6019 db 0
hydrated:00007FF61D6A601A db 0
hydrated:00007FF61D6A601B db 0
```

NFLabs.

Dehydrate/Rehydrate

- 最終的に、hydrateセクションにはMethod TableやFrozen Objectなどが配置される
- Method Tableには型の種類やサイズ、Method Tableへのポインタや Vtable等の情報が含まれている

Method Tableの
メモリ上の構造イメージ

_uFlagsは型の種類を表すビットフィールド

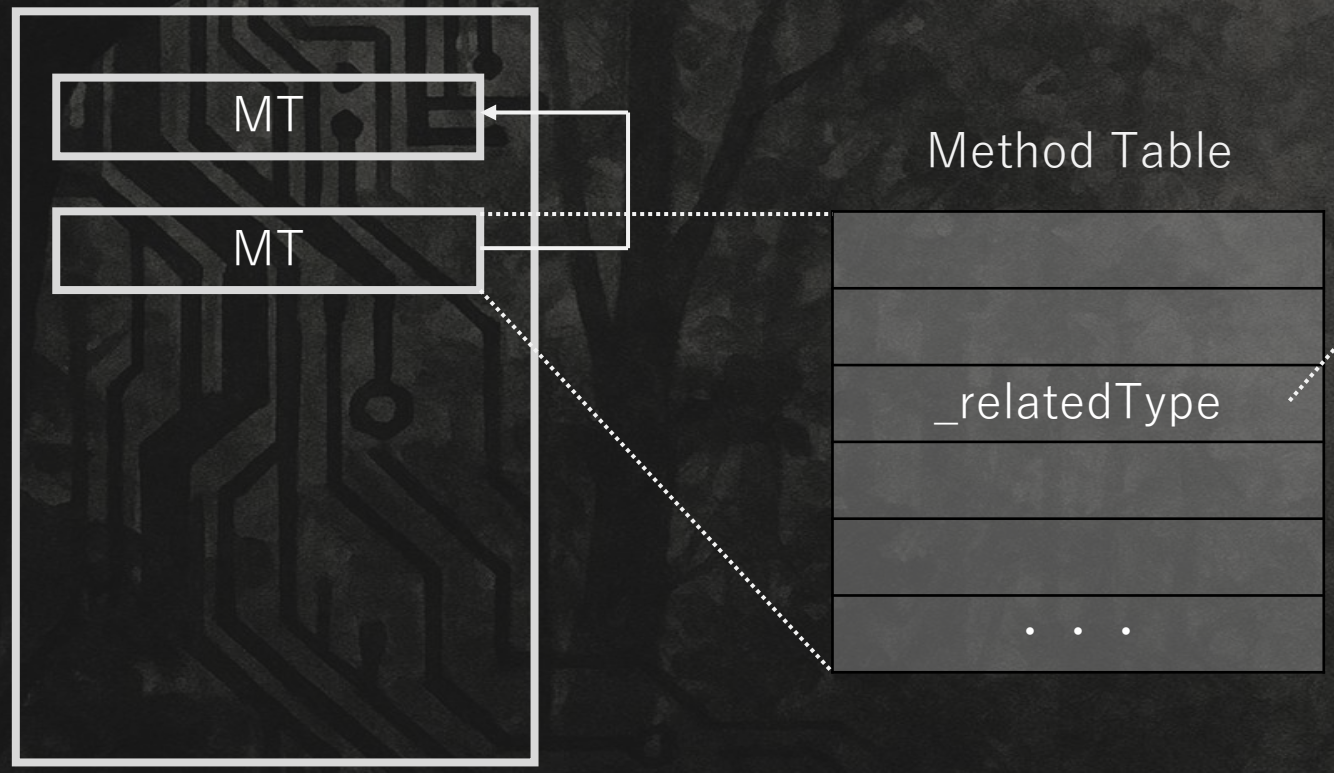
_relatedTypeに親クラスのMTへのポインタ

VtableSlotsには_usNumVtableSlotsの数だけの仮想メソッド

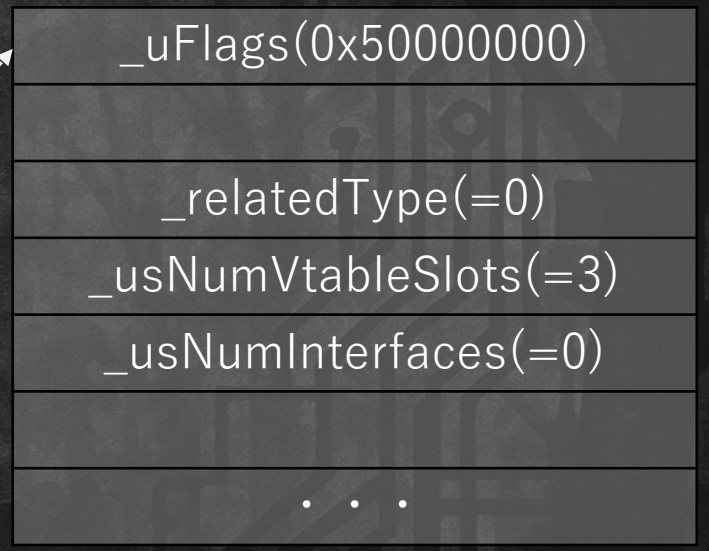
NFLabs.

Dehydrate/Rehydrate

hydrateセクションのメモリ上のイメージ



System.ObjectのMethod Table



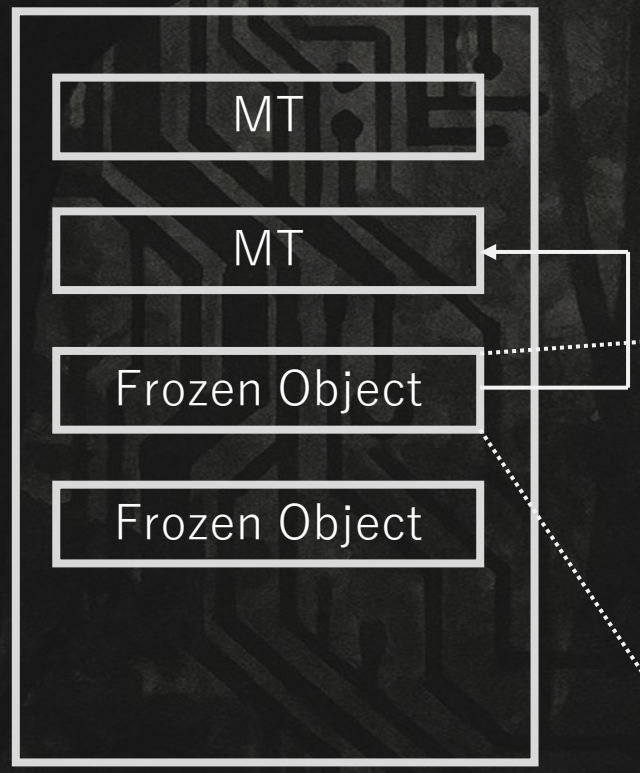
hydrateセクション上には復元された複数のMethod Tableが連続して配置される

relatedTypeから親のMTが分かる
relatedTypeを辿ることで最終的にSystem.ObjectのMTに到達し、MTの継承関係の全体像を把握できる

NFLabs.

Dehydrate/Rehydrate

hydrateセクションのメモリ上のイメージ



Frozen Objectも同様に復元されて
hydrateセクション内へと配置される

文字列オブジェクトのイメージ

System.StringのMethod Table

Rehydrateにより、文字列のFrozen Object
もhydrateセクションに展開される
展開された文字列オブジェクトは対応する
文字列型(System.String)のMTを保持

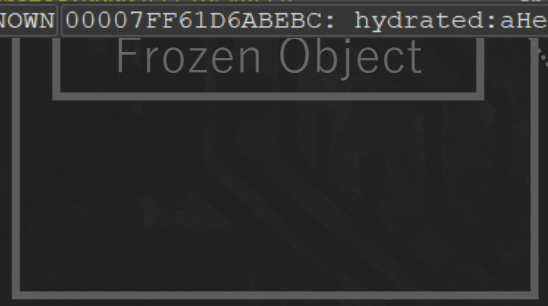
NFLabs.

Dehydrate/Rehydrate

```
hydrated:00007FF61D6ABEAF                   db    0
hydrated:00007FF61D6ABEB0 off_7FF61D6ABEB0  dq offset Ptr_MT       ; DATA XREF: RhEnableFinalization(voi
hydrated:00007FF61D6ABEB0                                         ; __managed__Main+D8↑o
hydrated:00007FF61D6ABEB8                   dd 0Dh
hydrated:00007FF61D6ABEBC aHelloWorld:
hydrated:00007FF61D6ABEBC                   text "UTF-16LE", 'Hello, World!',0
hydrated:00007FF61D6ABED8                   db    0
hydrated:00007FF61D6ABED9                   db    0
hydrated:00007FF61D6ABEDA                   db    0
hydrated:00007FF61D6ABEDB                   db    0
hydrated:00007FF61D6ABEDC                   db    0
hydrated:00007FF61D6ABEDD                   db    0
hydrated:00007FF61D6ABEDE                   db    0
hydrated:00007FF61D6ABEDF                   db    0
hydrated:00007FF61D6ABEE0 off_7FF61D6ABEE0  dq offset Ptr_MT       ; DATA XREF: S_P_CoreLib_System_String
hydrated:00007FF61D6ABEE8                   dd 4
hydrated:00007FF61D6ABEEC aHhms:
hydrated:00007FF61D6ABEEC                   text "UTF-16LE", 'Hhms',0
hydrated:00007FF61D6ABEF6                   db    0
hydrated:00007FF61D6ABEF7                   db    0
UNKNOWN 00007FF61D6ABEBC: hydrated:aHelloWorld (Synchronized with RIP)
```

System.stringのMethod Table

hydrateセクション内のFrozen Objectの領域に文字列型のオブジェクトレイアウトが並ぶ
Frozen Objectの領域は、ReadyToRunディレクトリの0x206のセクションの箇所のポインタで示されている




```
dq offset unk_7FF61D74C000
dq offset unk_7FF61D74C00C
dd 206                  ; FrozenObjectRegion
dd 1
dq offset Frozen_start
dq offset Frozen_end
dd 207                  ; DehydratedData
dd 1
dq offset StartPointer
dq offset EndPointer
dd 213
dd 1
```

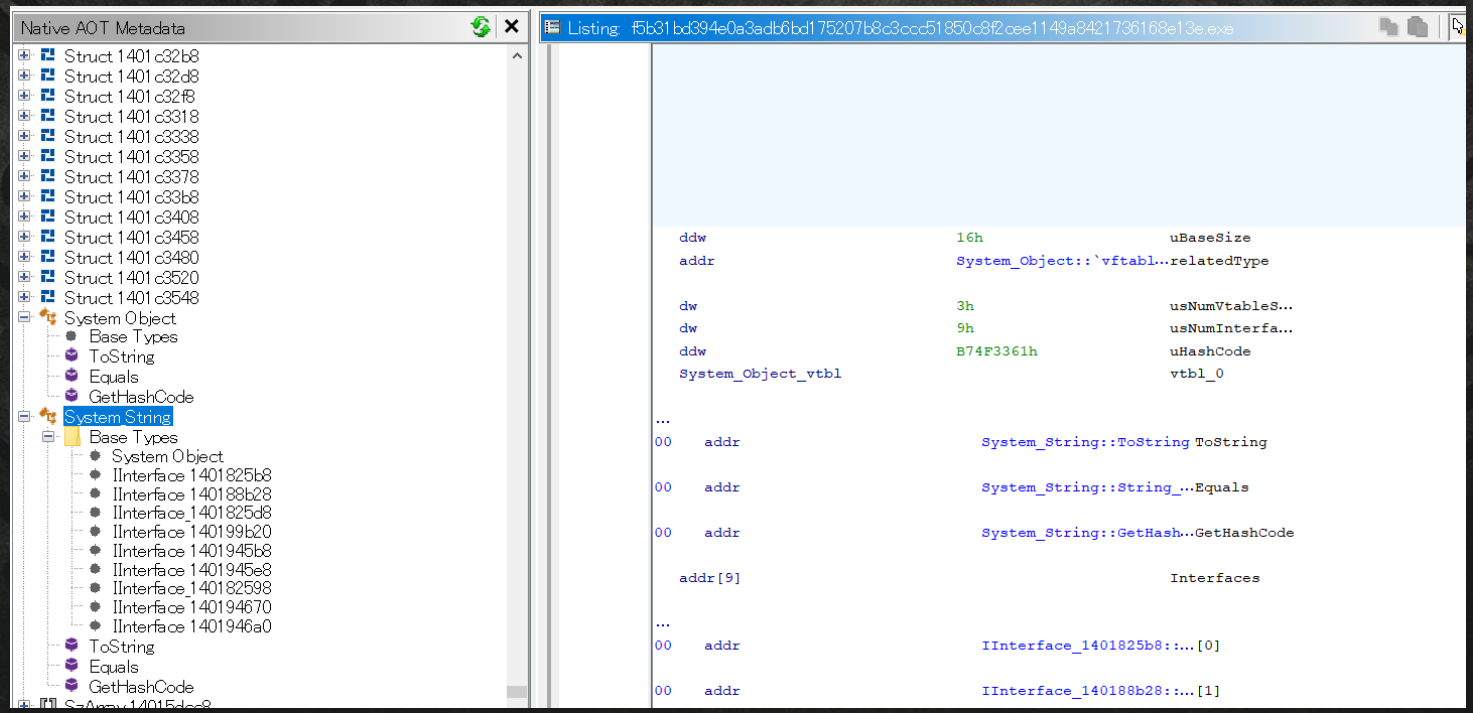
Ghidraプラグイン

静的解析において、ディスアセンブル画面では文字列リテラルへのポインタが初期化されていないため、文字列情報が表示されていない。これを以下の優れたプラグイン(Ghidra)を使うことで、動的解析(デバッグ実行)をせずに圧縮されたコードを静的に展開することができる。

- Ghidra .NET Native AOT Analyzer Plugin
  - https://github.com/Washi1337/ghidra-nativeaot
  - https://blog.washi.dev/posts/recovering-nativeaot-metadata/

Ghidraプラグイン

- 圧縮データをhydrateセクションに展開し、ディスアセンブル/デコンパイル画面の変数をアノテートしてくれるプラグイン
- Method Tableを探し出してウィンドウに列挙してくれる



NFLabs.

ハンズオン

ハンズオン #5

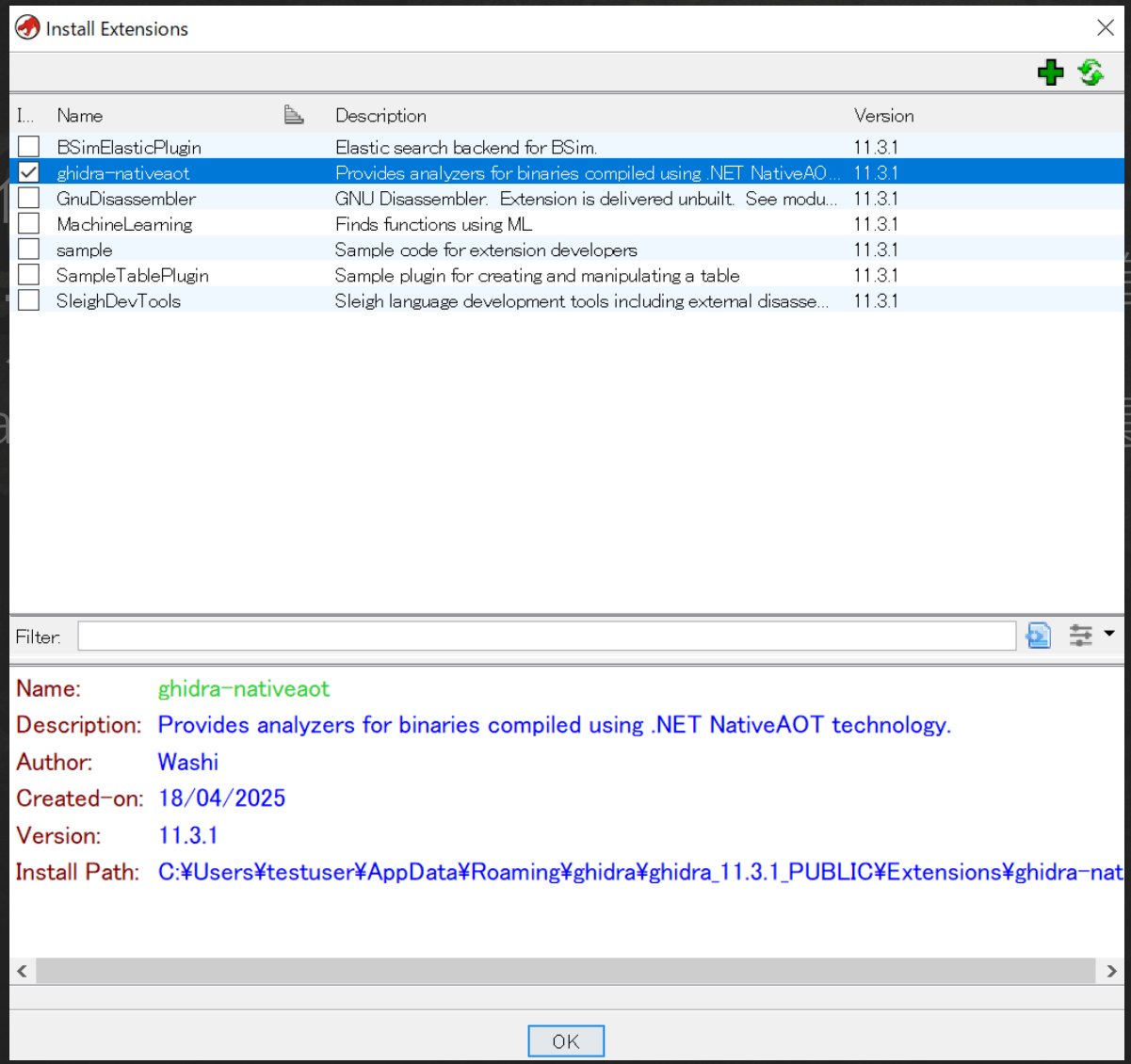
プラグインのインストール

- ghidra_11.3.1_PUBLIC¥Extensions¥Ghidra¥ 以下に、追加するプラグインzipファイルを配置する
- File → Install Extentionsから追加するプラグインを選択する
- 再起動する

ハンズオン #5

プラグインのイ

- ghidra_11.3. 追加するプラグイ
  ンzipファイ
- File → Insta 選択する
- 再起動する



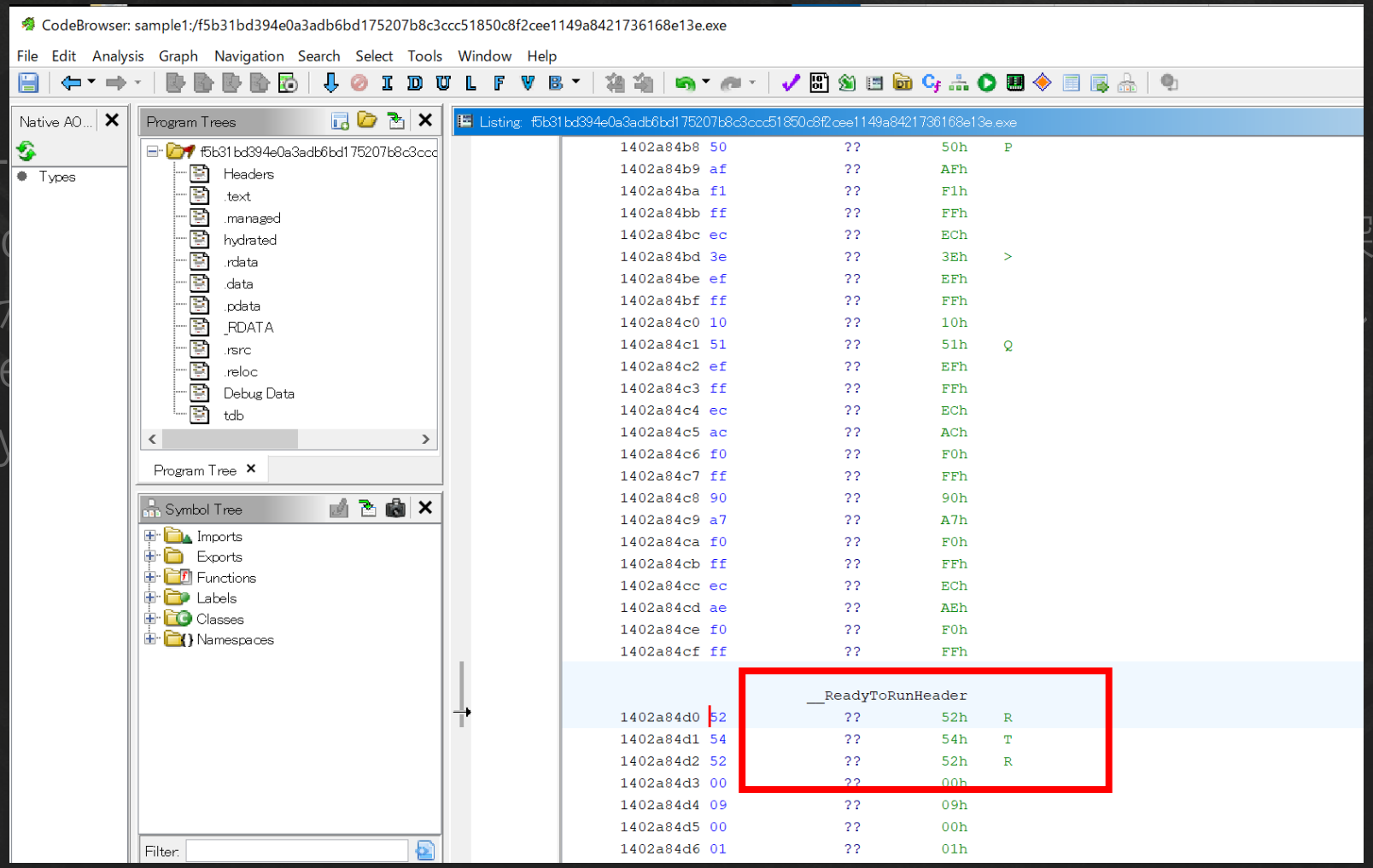
ハンズオン #5

メタデータ、Frozen Objectの復元

- Search→Memoryで“RTR”を検索(ReadyToRun ヘッダーを探す)
- 見つかったらコンテキストメニューから、Add labelを選択して、"__ReadyToRunHeader"のラベルを設定する

ハンズオン #5

メタデー
- Searc
- 見つ
  "__Re
- Analy



右クリック -> Add label

NFLabs.

ハンズオン #5

メタデータ、Frozen Objectの復元

- Analysis→One Shot→Native AOT Analyzerを選択

NFLabs.

```
undefined8 uVar1;
longlong lVar2;
undefined8 uVar3;
undefined8 uVar4;

uVar1 = FUN_14008ce30(0x1c,0);
uVar1 = S_P_CoreLib_System_IO_Path__Combine(uVar1,&dn_u_IsFaulted_1401686cc);
uVar1 = S_P_CoreLib_System_IO_Path__Combine(uVar1,&dn_u_Zm1jKA3zzv9xxVsj7Rl6c_140174784);
uVar1 = S_P_CoreLib_System_IO_Path__Combine(uVar1,&dn_u_IsClosed.exe_14016869c);
lVar2 = FUN_140002b80(&SzArray_1401ba020::`vftable',5);
*(System_String **)(lVar2 + 0x10) = &dn_u_@echo_off_:loop_tasklist_/FI_"IM_14016017c;
FUN_140002c70(lVar2 + 0x18,&dn_u_IsClosed.exe_14016869c);
*(System_String **)(lVar2 + 0x20) = &dn_u_"_|_find_/I_"_14015e454;
FUN_140002c70(lVar2 + 0x28,&dn_u_IsClosed.exe_14016869c);
*(System_String **)(lVar2 + 0x30) = &dn_u_">nul_if_errorlevel_1_(_14015e404;
uVar3 = String__Concat_12(lVar2);
uVar1 = String__Concat_7(uVar3,&dn_u_start_""_/MIN_"_140176ed4,uVar1,
                         &dn_u_")_timeout_/t_1_>nul_goto_loop_14015e384);
uVar3 = S_P_CoreLib_System_IO_Path__GetTempPath();
uVar4 = String__Concat_5(&dn_u_Zm1jKA3zzv9xxVsj7Rl6c_140174784,&dn_u_.cmd_14015f9c4);
uVar3 = S_P_CoreLib_System_IO_Path__Combine(uVar3,uVar4);
uVar4 = S_P_CoreLib_System_IO_File__get_UTF8NoBOM();
S_P_CoreLib_System_IO_File__Validate(uVar3,uVar4);
S_P_CoreLib_System_IO_File__WriteToFile(uVar3,2,uVar1,uVar4);
uVar1 = S_P_CoreLib_System_IO_Path__GetFullPath(uVar3);
FUN_1400cf440(uVar1,2);
lVar2 = RhpNewFast(&Class_140180848::`vftable');
*(System_String **)(lVar2 + 8) = &dn_u_cmd.exe_1401750c4;
uVar1 = String__Concat_6(&dn_u_/c_"_14015fa7c,uVar3,&dn_u_"_14015e36c);
FUN_140002c70(lVar2 + 0x10,uVar1);
```

```
FUN_14007c590(0x14);
iVar4 = S_P_CoreLib_System_SpanHelpers__IndexOf_0
                  (u_$startup$,$antivt$,$antiprocessh_14015e824,0x27,u_$startup$_14015e7fc,9);
if (-1 < iVar4) {
  FUN_140078c00();
}
iVar4 = S_P_CoreLib_System_SpanHelpers__IndexOf_0
                  (u_$startup$,$antivt$,$antiprocessh_14015e824,0x27,
                   u_$antiprocesshacker$_14015e73c,0x13);
if (-1 < iVar4) {
  FUN_140078b60();
}
iVar4 = S_P_CoreLib_System_SpanHelpers__IndexOf_0
                  (u_$startup$,$antivt$,$antiprocessh_14015e824,0x27,u_$melt$_14015e7a4,6);
if (-1 < iVar4) {
  FUN_140078e10();
}
iVar4 = S_P_CoreLib_System_SpanHelpers__IndexOf_0
                  (u_$startup$,$antivt$,$antiprocessh_14015e824,0x27,u_$persistence$_14015e7cc,
                  0xd
                  );
if (-1 < iVar4) {
  FUN_140078f80();
}
```

ありがたい🥹

ハンズオン #5

早く終わった方は他の検体にも適用、実際に解析してみて変化を味わってみてください

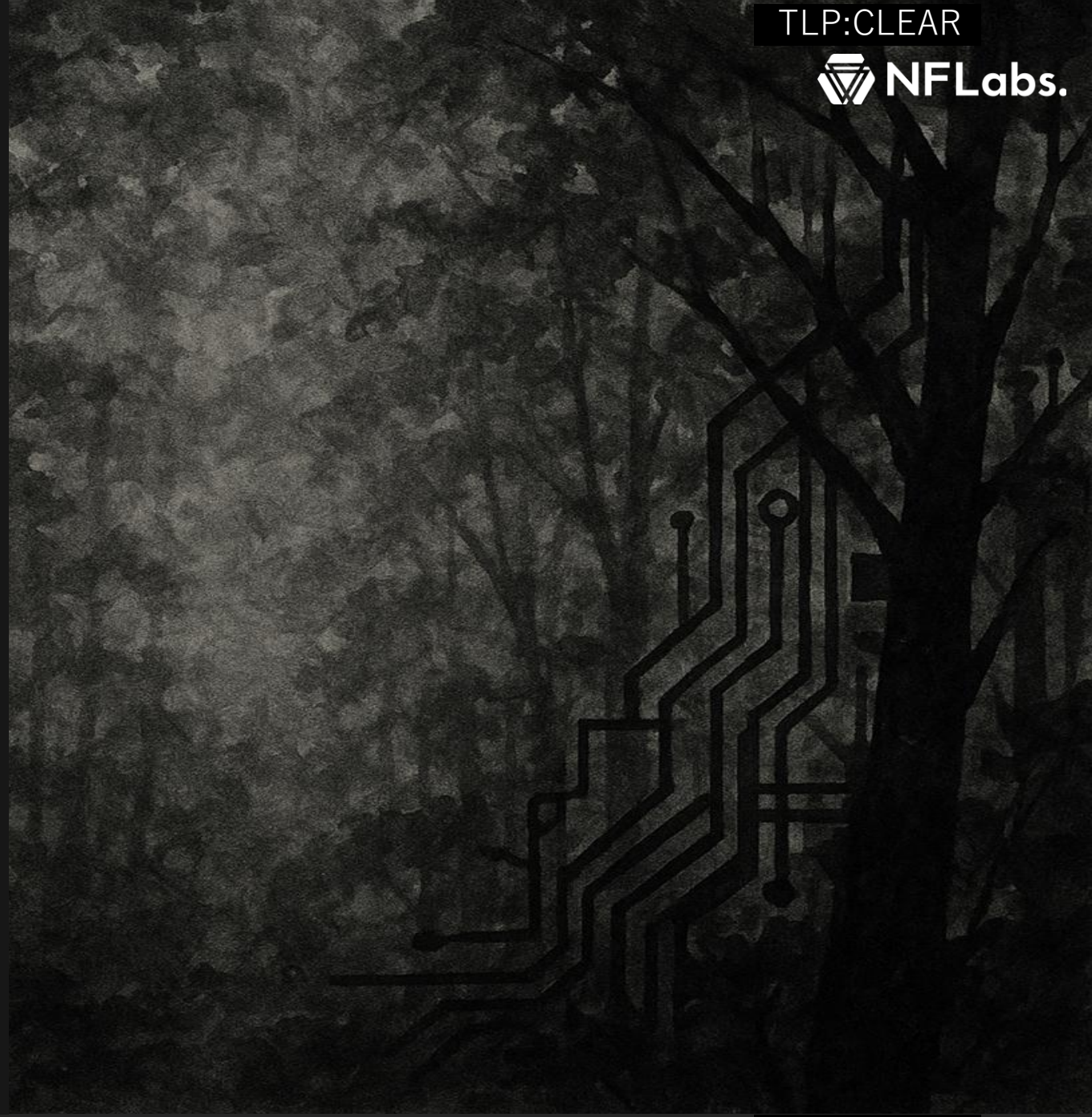
NFLabs.

まとめ

- Native AOTバイナリに対し、GhidraやIDA Proでシグネチャを作成・適用した
- Method Tableなどメタデータの展開(Rehydrate)処理の流れを解説した
- Ghidraプラグインでメタデータを静的に復元し、文字列を識別でいるようにした
- Native AOTマルウェアに遭遇した際には、本手法をぜひ活用してみてください

NFLabs.

Thank you !



NFLabs.

参考

- Native AOT
  - https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/?tabs=windows%2Cnet8
  - GitHub - dotnet/runtime: .NET is a cross-platform runtime for cloud, mobile, desktop, and IoT apps.
- Malware
  - IBM X-Force Threat Analysis: QuirkyLoader - A new malware loader delivering infostealers and RATs | IBM
  - Booking.com Phishing Campaign Targeting Hotels and Customers - Sekoia.io Blog
  - Impersonated GenAI Site Lures Victims to Infostealer Download - Check Point Research
- プラグイン
  - https://github.com/Washi1337/ghidra-nativeaot
  - https://blog.washi.dev/posts/recovering-nativeaot-metadata/