Appearances are deceiving: Novel offensive techniques in Windows 10/11 on ARM

株式会社ＦＦＲＩセキュリティ
https://www.ffri.jp

自己紹介

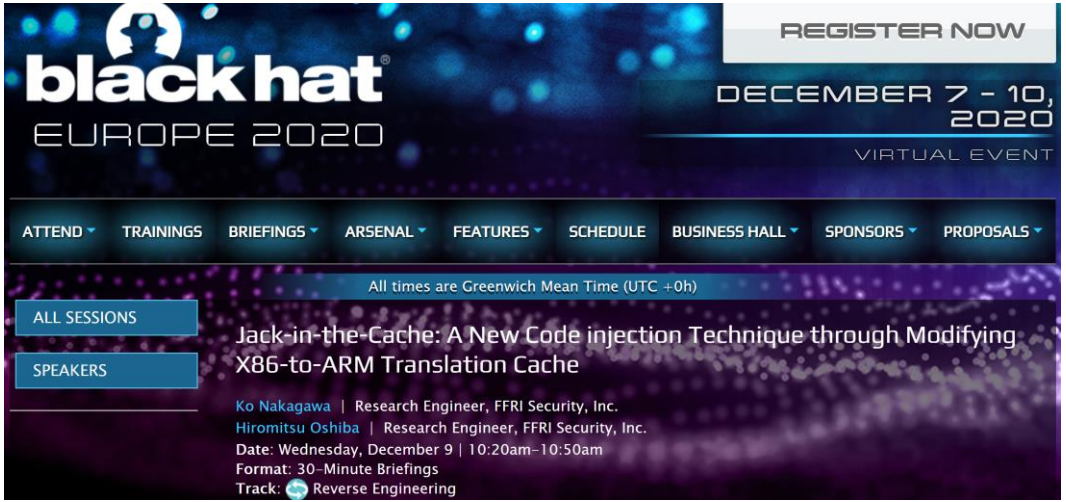
FFRI Security, Inc.に新卒入社

基礎技術研究室でリサーチエンジニアとして勤務

近年ではARM版WindowsやM1 Macの互換性テクノロジーの解析を実施

Black Hat EU 2020 Briefings Speaker

GitHub: https://github.com/kohnakagawa



https://www.blackhat.com/eu-20/briefings/schedule/index.html#jack-in-the-cache-a-new-code-injection-technique-through-modifying-x-to-arm-translation-cache-21324

ARMの台頭

ARM版Windows搭載のSurface Pro X

Surface Pro X – Ultra-thin & Always Connected 2-in-1 Laptop – Microsoft Surface

M1 Mac

MacBook Pro 13-inch - Apple

ARMベースのノートPCが次々と登場

※Microsoftのドキュメントに従い、ArmではなくARMと表記

なぜARMベースのノートPCか

ARMは電力辺りの性能に優れるため

バッテリー駆動時間が従来端末に比べ長いケースが多い※

- スマートフォンのようにノートPCを使うことができるAlways Connected PCも登場

ニューノーマルの時代では時間や場所にとらわれずに働くことが当たり前に

- バッテリー駆動時間が長いことは重要な要素に

今後、ARMベースのノートPCの需要が高まることが予想

※Windows on ARM Benchmarked > Impressions, Superb Battery Life, What Needs to Improve | TechSpot

アプリケーションの互換性の問題

```
int mean(int a, int b) {
    return (a + b) / 2;
}
```

x86/x64

```
lea eax, [ecx + edx]
cdq
sub eax, edx
sar eax, 1
ret
```

≠

ARM64

```
add w8, w0, w1
add w9, w8, w8, lsr 31
asr w0, w9, 1
ret
```

既存のx86/x64のソフトウェアが使えなくなる問題に直面

互換性の問題の解決

バイナリ変換とキャッシュ機構

x86/x64からARM64向けに実行可能なように命令列を変換

変換処理は重いため、高速化のためにキャッシュ機構が付随

- 変換結果をファイルとしてキャッシュし、次の実行のタイミングで使いまわす

x86/x64

```
lea eax, [ecx + edx]
cdq
sub eax, edx
sar eax, 1
ret
```

実行中 or 実行前に変換

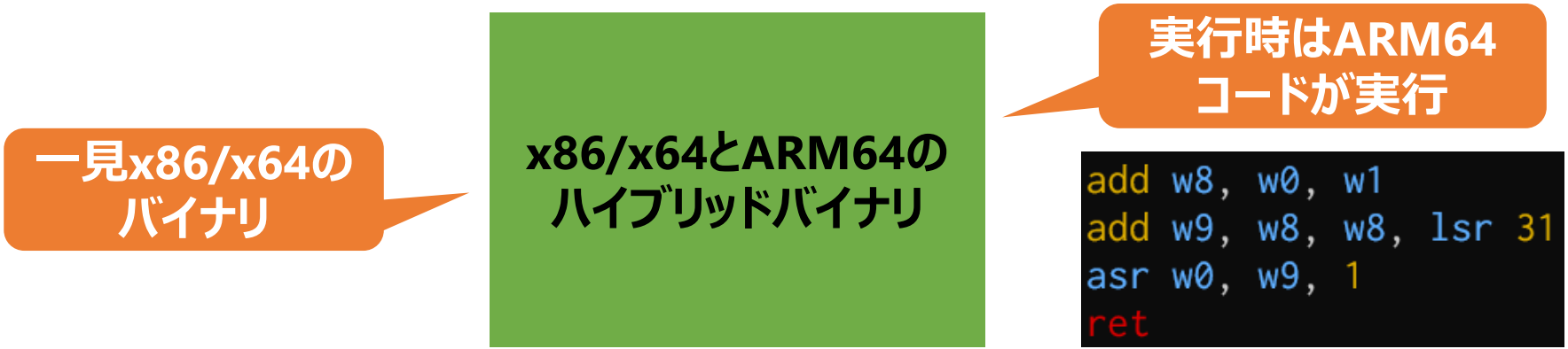
ARM64

```
add w8, w0, w1
add w9, w8, w8, lsr 31
asr w0, w9, 1
ret
```

互換性の問題の解決

ハイブリッドバイナリ

既存のバイナリとの互換性を保ちつつ、ネイティブ同等の速度でコード実行を可能に



互換性の問題の解決

ファットバイナリ

複数のアーキテクチャ向けのバイナリを組み合わせ、1つにしたバイナリ

- 1つのバイナリを複数の用途、あるいは複数のプラットフォームで利用可能な形に

WindowsとmacOSでの実装

互換性テクノロジー	Windowsでの実装	macOSでの実装
バイナリ変換とキャッシュ機構	XTAJITとXtaCache	Rosetta 2
ハイブリッドバイナリ	CHPE・ARM64EC	N/A
ファットバイナリ	ARM64X	Universal 2

ARM版WindowsとM1 Macそれぞれで互換性テクノロジーが用意

問題: 互換性テクノロジーの悪用可能性について

過去に互換性テクノロジーが悪用された事例は存在

例: Application Shimming

ARM版WindowsやM1 Macに導入された互換性テクノロジーの悪用可能性は?

-> **我々の知る限り、議論すらされていない**

そもそも、これらの互換性テクノロジーの詳細が明らかになっていない

MicrosoftやAppleが公式で発表している情報はわずか

リバースエンジニアリング結果もあまり多くない

本研究の目的

新しく導入された互換性テクノロジーの悪用による攻撃手法を明らかにすること

具体的には...

- 互換性テクノロジーの詳細 (※互換性テクノロジーに付随する高速化技術も含む)
- それらを悪用した攻撃手法

の2つを明らかにすること

本研究を契機に互換性テクノロジーに関するセキュリティの研究が活発化されることを期待

本研究で対象とする互換性テクノロジー

互換性テクノロジー	Windowsでの実装	macOSでの実装
バイナリ変換とキャッシュ機構	XTAJITとXtaCache	Rosetta 2
ハイブリッドバイナリ	CHPE・ARM64EC	N/A
ファットバイナリ	ARM64X	Universal 2

ARM版Windowsの3つの互換性テクノロジーを今回は対象

本研究で対象とする互換性テクノロジー

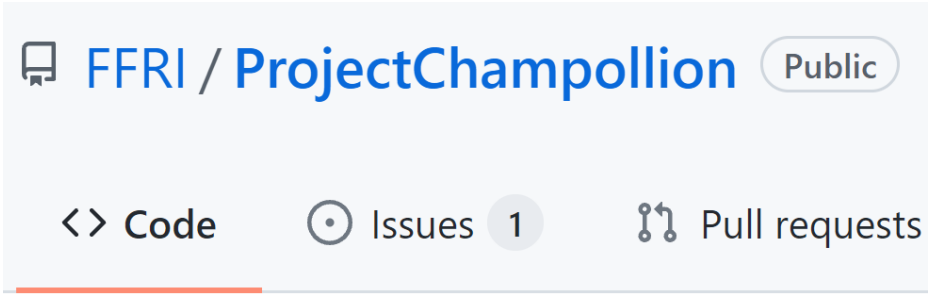
macOSについては…

互換性テクノロジー	Windowsでの実装	macOSでの実装
バイナリ変換とキャッシュ機構	XTAJITとXtaCache	Rosetta 2
ハイブリッドバイナリ	CHPE・ARM64EC	N/A
ファットバイナリ	ARM64X	Universal 2

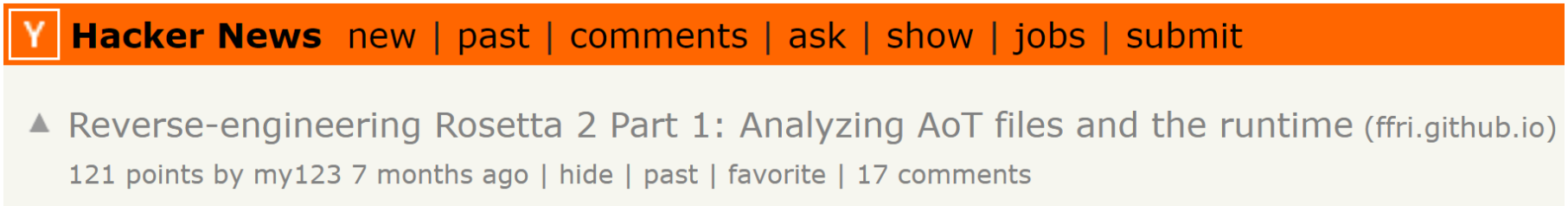
ARM版Windowsの3つの互換性テクノロジーを今回は対象

Project Champollion

macOSのRosetta 2のリバースエンジニアリング結果をまとめたリポジトリ



https://github.com/FFRI/ProjectChampollion



https://news.ycombinator.com/item?id=26346980

本研究で対象とする互換性テクノロジー

- バイナリ変換とキャッシュ機構 (XTAJITとXtaCache)
- ハイブリッドバイナリ (CHPE・ARM64EC)
- ファットバイナリ (ARM64X)

本研究で対象とする互換性テクノロジー

- バイナリ変換とキャッシュ機構 (XTAJITとXtaCache)
- ハイブリッドバイナリ (CHPE・ARM64EC)
- ファットバイナリ (ARM64X)

ARM版Windowsにおけるバイナリ変換とキャッシュ機構

バイナリ変換: XTAJIT (※XTAはX86-To-ARMの略と思われる)

実行時にx86/x64コードをARM64コードにJITバイナリ変換

- Windows 10 Insider PreviewとWindows 11ではx64もサポート

キャッシュ機構: XtaCache

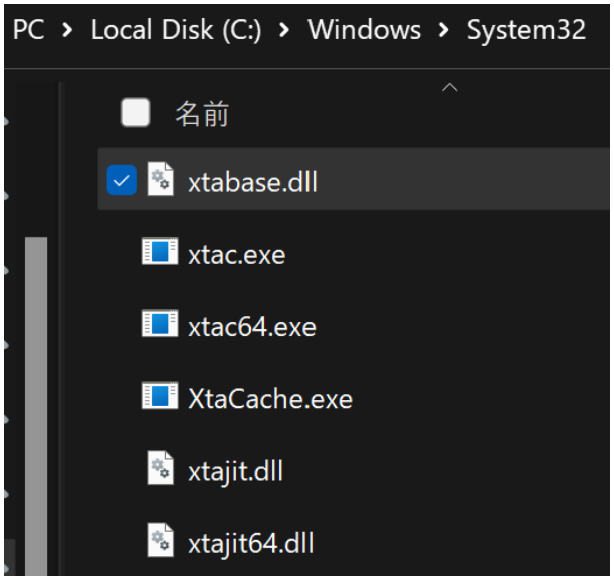
変換結果はXTA Cache Fileとして保存され、次の実行のタイミングで使い回される

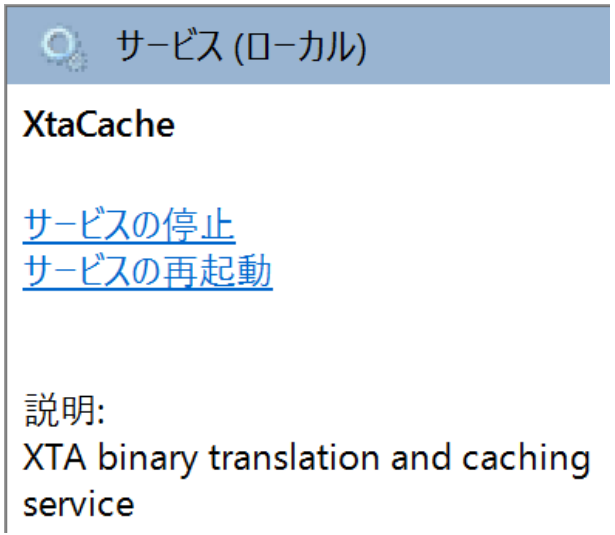
- JITバイナリ変換のオーバーヘッドを削減し、2回目のアプリケーションの実行を高速化

x86/x64エミュレーション

x86/x64エミュレーションに関連するEXEとDLL

- xtajit.dll/xtajit64.dll: x86/x64エミュレーターDLL
- xtac.exe/xtac64.exe: XTA Cache Fileを作成するためのコンパイラ
- XtaCache.exe: XTA Cache Fileのマネジメントサービス





x86/x64エミュレーションの流れ

※XTA Cache Fileを使った場合の実行の流れ

XTA Cache Fileのディレクトリー

ACCESSCHK.EXE.95...mp.1.jc

⋮

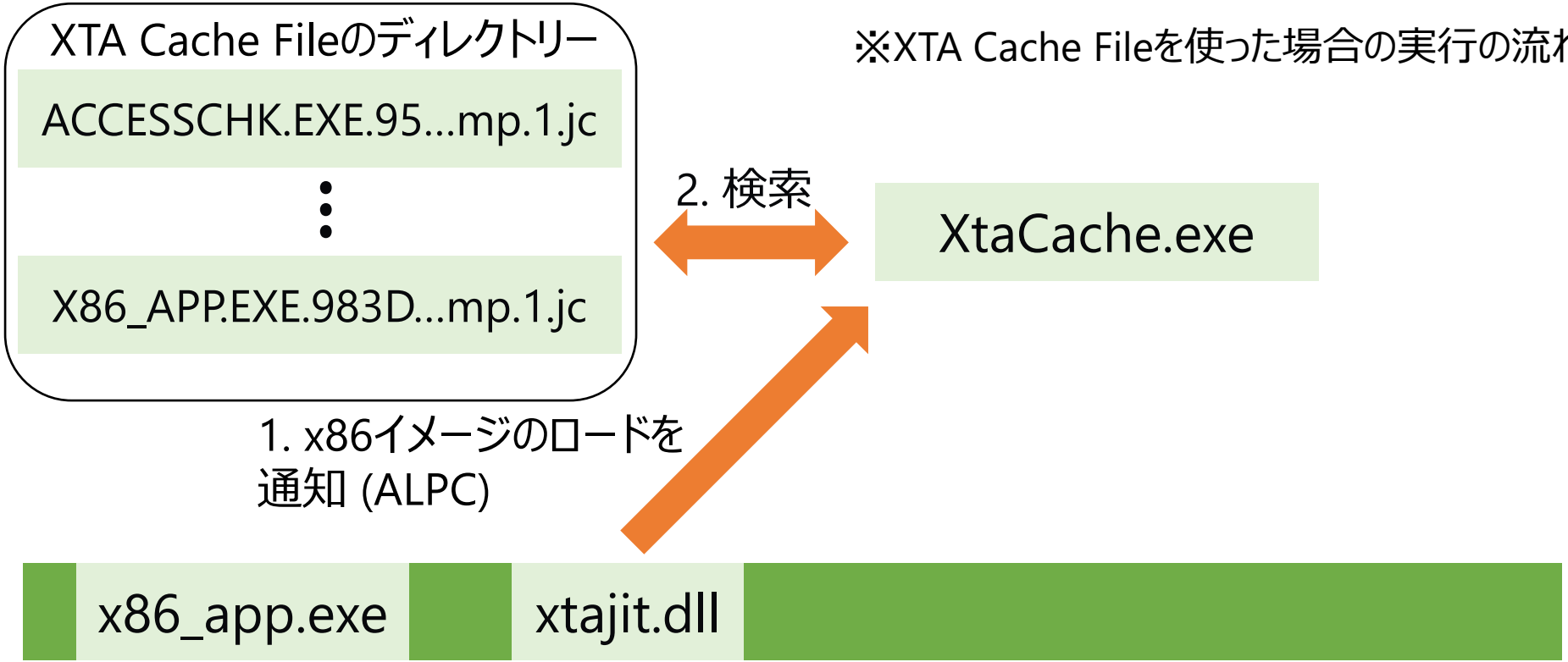
X86_APP.EXE.983D...mp.1.jc

XtaCache.exe

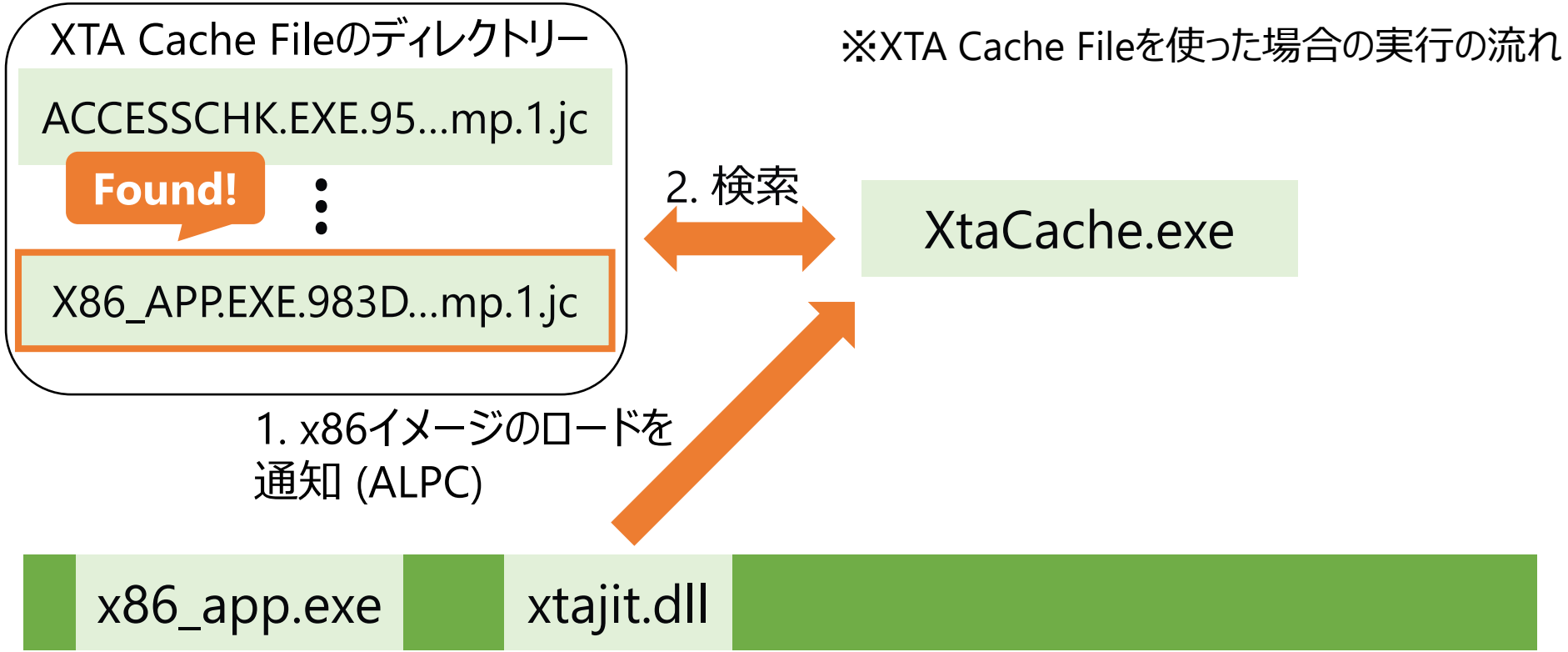
1. x86イメージのロードを
通知 (ALPC)

x86/x64エミュレーションの流れ

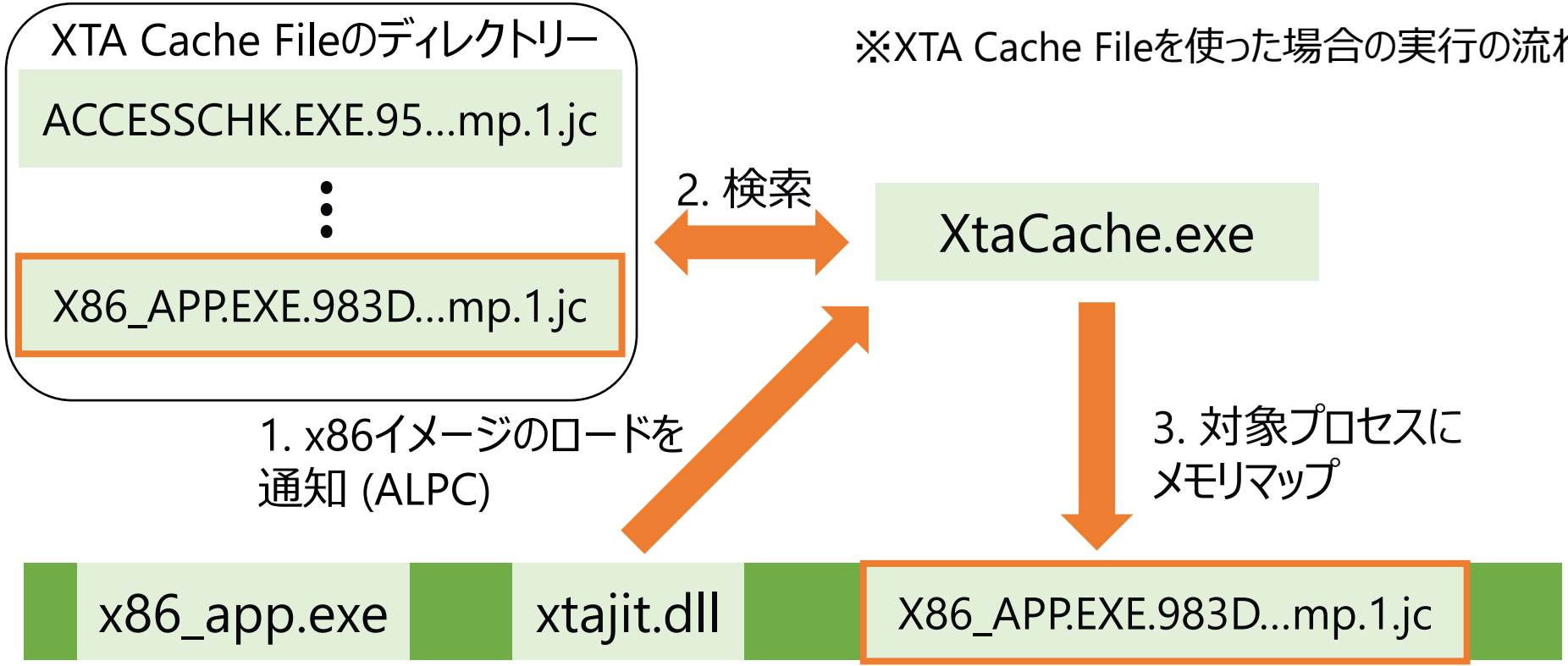
※XTA Cache Fileを使った場合の実行の流れ



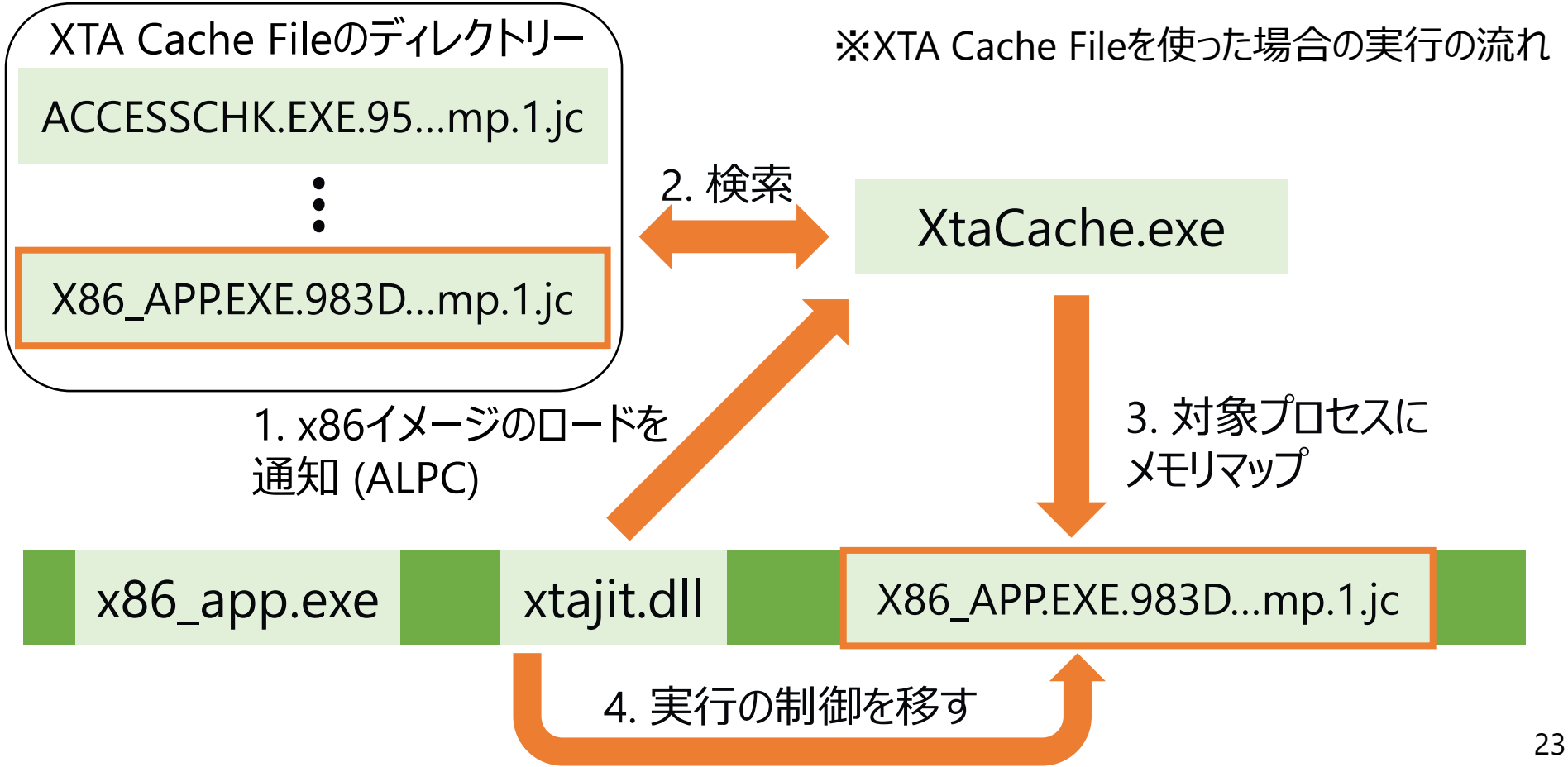
x86/x64エミュレーションの流れ



x86/x64エミュレーションの流れ



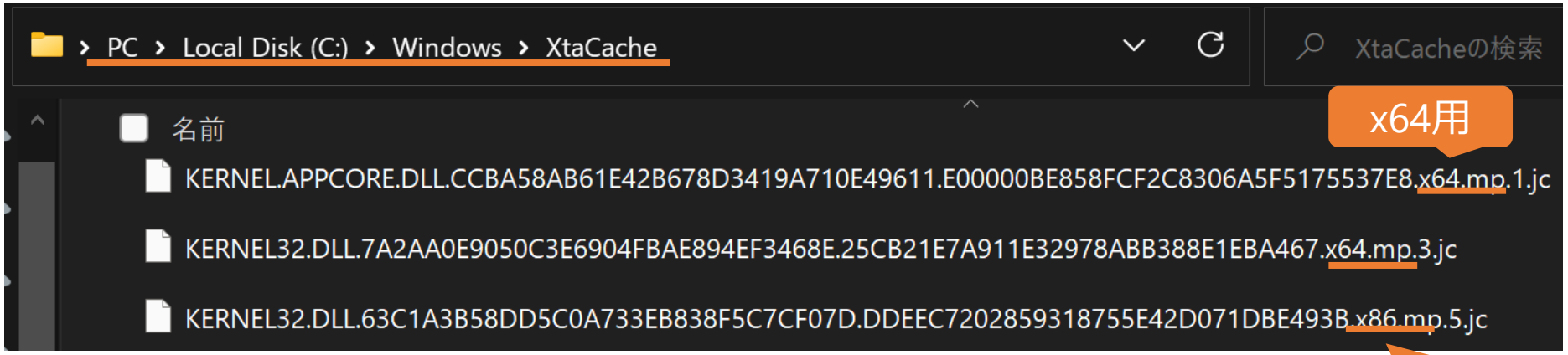
x86/x64エミュレーションの流れ



XTA Cache File

%SystemRoot%¥XtaCacheにXTA Cache Fileは存在

x86とx64のPEごとにそれぞれ別のXTA Cache Fileが作成



デフォルトの設定ではXtaCacheサービスにのみアクセスが許可

- しかし、管理者権限によりアクセス権を変更可能

XTA Cache Fileの構造

ファイルフォーマットの詳細はBlack Hat EU 2020の発表資料を参照

XTA Cache Fileのパーサーや中に含まれるARM64コードの解析ツールは公開済み

https://github.com/FFRI/XtaTools

```
Advc]0 0% 255 USER32.ĐLL.B762FE91071Đ23ĐA8720F34E3667A
- x86.00403db0:
0000b1c8      9dcf1fb8      str w29, [x28, -4]!
0000b1cc      fd031c2a      mov w29, w28
0000b1d0      9d4740b8      ldr w29, [x28], 4
0000b1d4      295d4311      add w9, w9, 0xd7, lsl 12
0000b1d8      29412f11      add w9, w9, 0xbd0
0000b1dc      23fbff97      bl 0x9e68
0000b1e0      20021fd6      br x17
```

XTA Cache Fileの逆アセンブル結果

https://github.com/FFRI/radare2

ここでは発表を理解する上で最低限必要な事項のみ説明

XTA Cache Fileの構造

Header

シグネチャー・後続のブロックへのオフセットの値などを含む

BLCK Stub

エミュレーターDLLへ戻る処理などを含む

Translated code

x86/x64からARM64への変換結果のコードを含む

キャッシュファイルの更新回数だけ繰り返し

⋮

```
str w29, [x28, -4]!
mov w29, w28
movz w27, 0x1
mov w28, w29
ldr w29, [x28], 4
```

難読化されておらず、そのままARM64の機械語が含まれる

Address pairs

x86/x64とARM64コードのRVAでのアドレスペア

NT path name

x86/x64 PEのNT path名

XTA Cache Fileの構造

Header

BLCK Stub

Translated code

キャッシュファイルの更新回数
だけ繰り返し

Address pairs

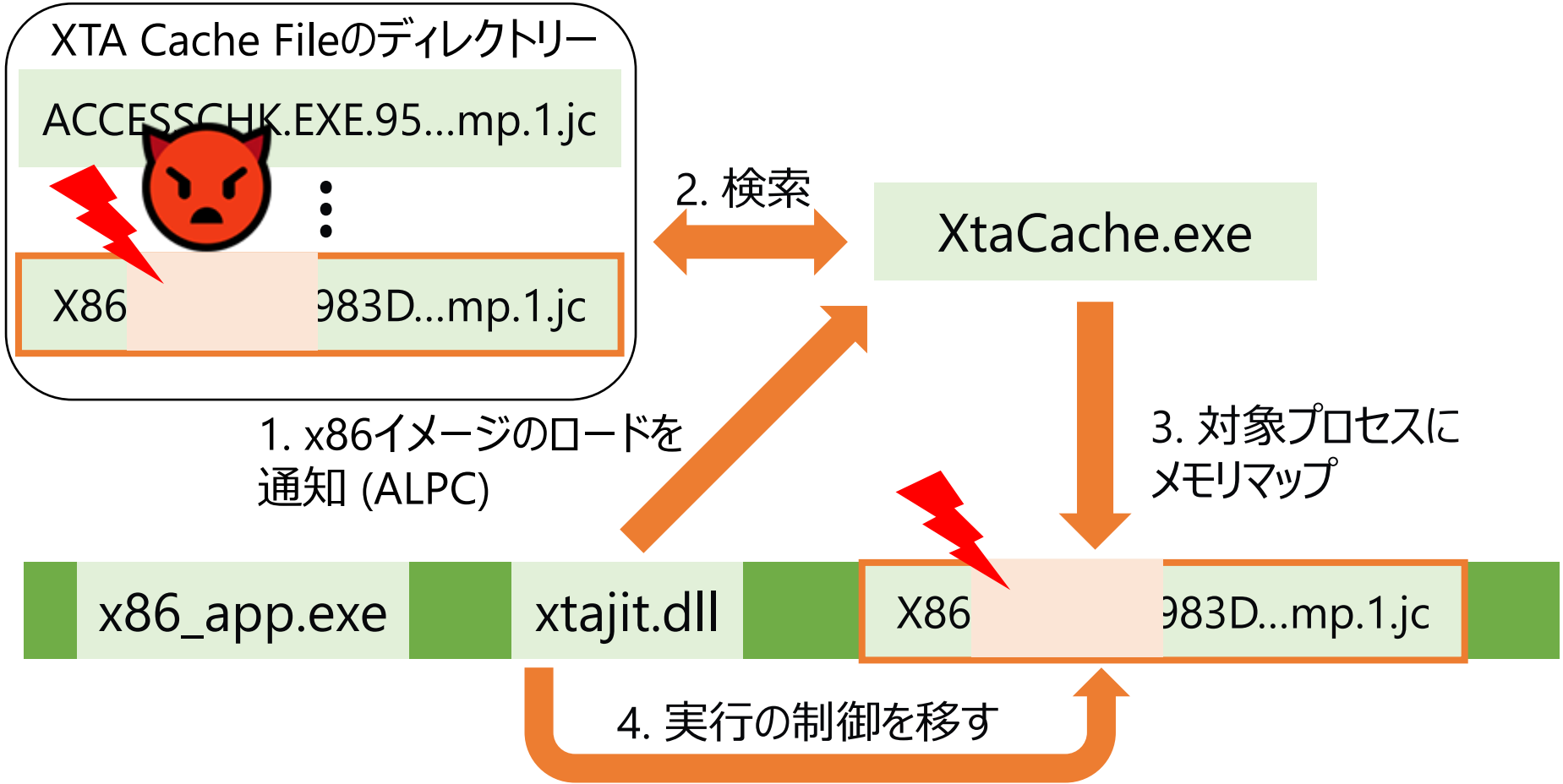
NT path name

このARM64コードを改ざんした場合
何が起きるか？

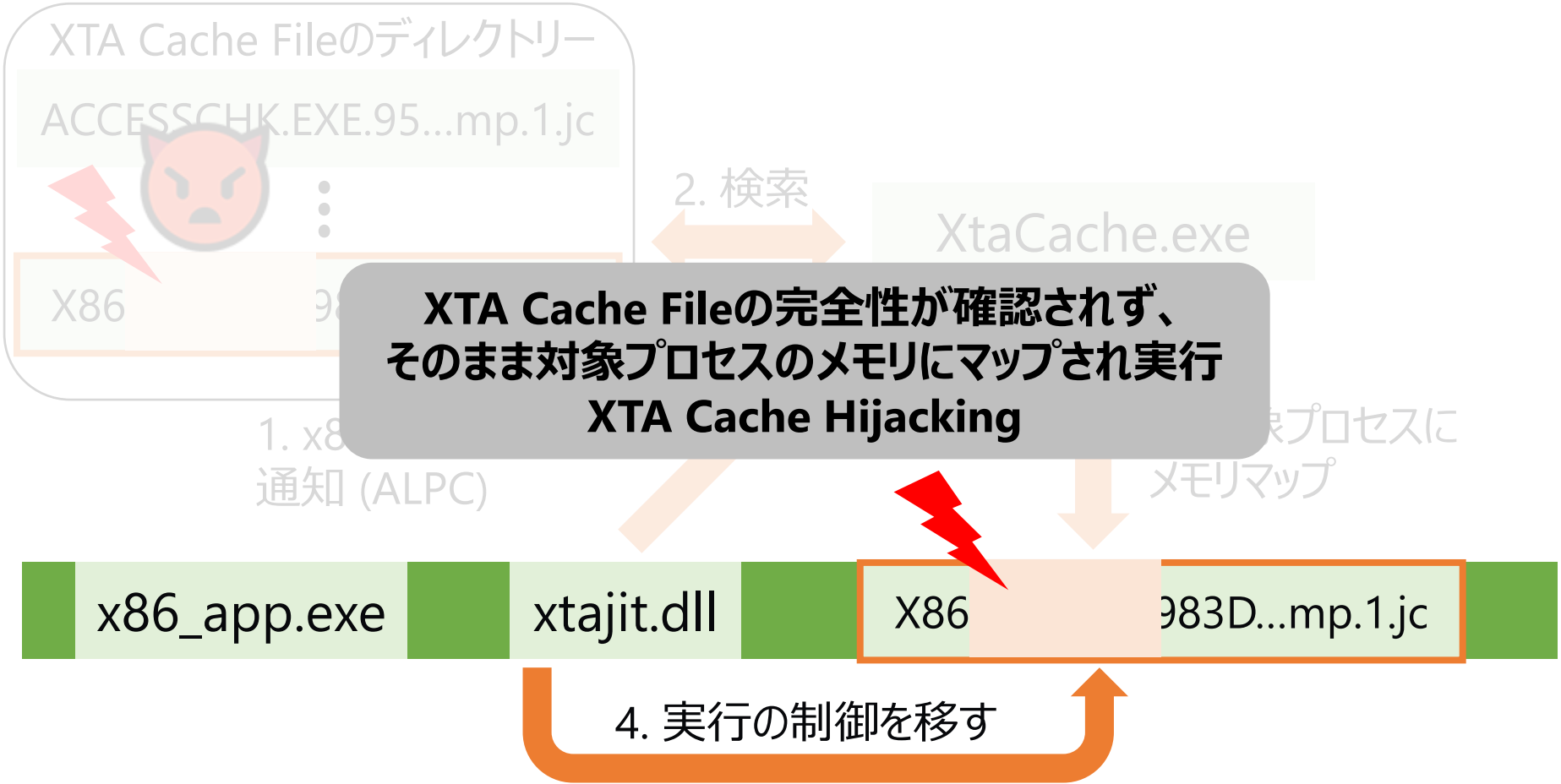
```
str w29, [x28, -4]!
mov w29, w28
movz w27, 0x1
mov w28, w29
ldr w29, [x28], 4
```

**難読化されておらず、そのまま
ARM64の機械語が含まれる**

改ざんされた場合のエミュレーションの流れ



改ざんされた場合のエミュレーションの流れ



XTA Cache Hijackingの特徴

3つの特徴を備えるコードインジェクション手法

- 検知困難: 対象プロセスのハンドルの取得なしに行えるため
  - 改ざんされたXTA Cache Fileのコードが通常のエミュレーションを介して実行されるのみ
- 追跡困難: 元となるx86/x64のPEファイルには痕跡がないため
  - XTA Cache Fileの存在を知らない場合、追跡が困難に
- 永続性: コードインジェクションの結果がファイルとして永続化されるため
  - 再起動後も同じアプリケーションを実行すればコードインジェクションが実行される

MITRE ATT&CK Matrix上では…

Defense Evasion・Credential Access・PersistenceのTacticsを実現するTechnique

しかしながら、**利用に際して管理者権限が必要**…

XTA Cache Hijackingの特徴

3つの特徴を備えるコードインジェクション手法

- 検知困難: 対象プロセスのハンドルの取得なしに行えるため
  - 改ざんされたXTA Cache Fileのコードが通常のエミュレーションを介して実行されるのみ
- 追跡困難: 元となるx86/x64のPEファイルには痕跡がないため
  - XTA Cache Fileの存在を知らない場合、追跡が困難に
- 永続性: コードインジェクションの結果がファイルとして永続化されるため
  - 再起動後も同じアプリケーションを実

管理者権限を取得してまでこの手法を利用する価値は本当にあるのか？

MITRE ATT&CK Matrix上では…

Defense Evasion・Credential Access・PersistenceのTacticsを実現するTech

しかしながら、**利用に際して管理者権限が必要**…

XTA Cache Hijackingの特徴

この手法に特有の特徴

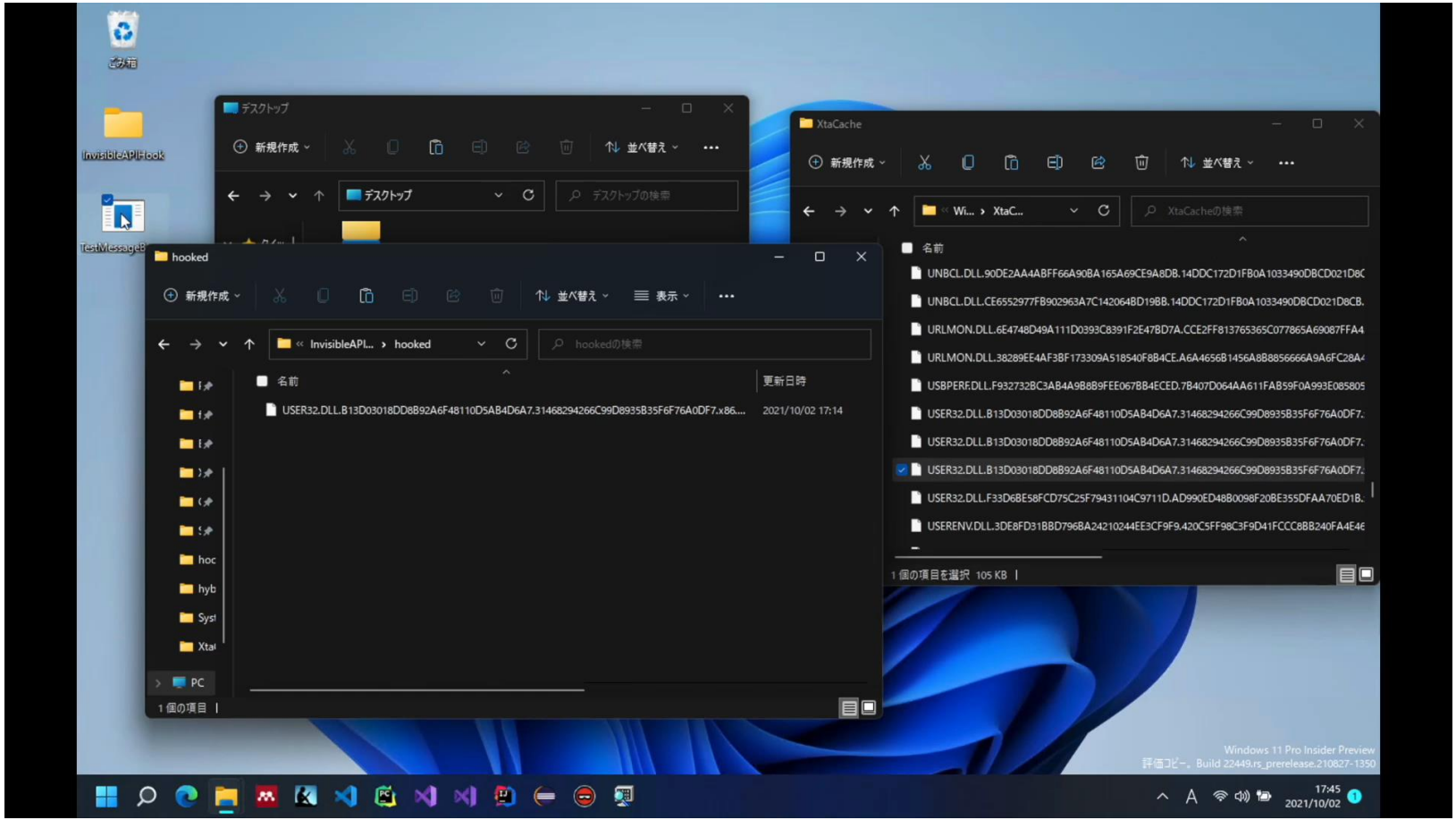
Invisible Execution

- x86/x64のコードレベルでの実行の痕跡を隠蔽しつつ、別のコードを実行可能

Invisible Executionを活用する例: Invisible API hooking

- Code hookingの場合、フックされていることの痕跡がコードに残る
- XTA Cache Hijackingにより**この痕跡を残すことなくCode hookingを行うことが可能**

デモ


ごみ箱
InvisibleAPIHook
TestMessageB
デスクトップ
新規作成
並べ替え
デスクトップ
デスクトップの検索
XtaCache
新規作成
並べ替え
Wi... › XtaC...
XtaCacheの検索
名前
UNBCL.DLL.90DE2AA4ABFF66A908A165A69CE9A8DB.14DDC172D1FB0A1033490DBCD021D8C
UNBCL.DLL.CE6552977FB902963A7C142064BD19BB.14DDC172D1FB0A1033490DBCD021D8CB.
URLMON.DLL.6E4748D49A111D0393C8391F2E47BD7A.CCE2FF813765365C077865A69087FFA4
URLMON.DLL.38289EE4AF3BF173309A518540F8B4CE.A6A4656B1456A8B8856666A9A6FC28A4
USBPERF.DLL.F932732BC3AB4A9B8B9FEE067BB4ECED.7B407D064AA611FAB59F0A993E085805
USER32.DLL.B13D03018DD8B92A6F48110D5AB4D6A7.31468294266C99D8935B35F6F76A0DF7.
USER32.DLL.B13D03018DD8B92A6F48110D5AB4D6A7.31468294266C99D8935B35F6F76A0DF7.
USER32.DLL.B13D03018DD8B92A6F48110D5AB4D6A7.31468294266C99D8935B35F6F76A0DF7.
USER32.DLL.F33D6BE58FCD75C25F79431104C9711D.AD990ED48B0098F20BE355DFAA70ED1B.
USERENV.DLL.3DE8FD31B8D796BA24210244EE3CF9F9.420C5FF98C3F9D41FCCC8BB240FA4E4E
1 個の項目を選択 105 KB
hooked
新規作成
並べ替え
表示
InvisibleAPI... › hooked
hookedの検索
名前
更新日時
USER32.DLL.B13D03018DD8B92A6F48110D5AB4D6A7.31468294266C99D8935B35F6F76A0DF7.x86.... 2021/10/02 17:14
hoc
hyb
Syst
Xtal
PC
1 個の項目
Windows 11 Pro Insider Preview
評価コピー。Build 22449.rs_prerelease.210827-1350
17:45
2021/10/02

XTA Cache Hijackingの対策

XtaCacheディレクトリーのアクセス権の変更の監視・制限

XTA Cache Fileを編集するにはXtaCacheディレクトリーのアクセス権の変更が必要

- デフォルトではXtaCacheサービス以外はアクセスできないため

XtaCacheディレクトリーのアクセス権の変更は通常まず起きない

- ここを監査対象に入れることで、XTA Cache Hijackingが使われたことを追跡可能
- アクセス権の変更を制限することで本手法から防御することが可能

ここまでのまとめ

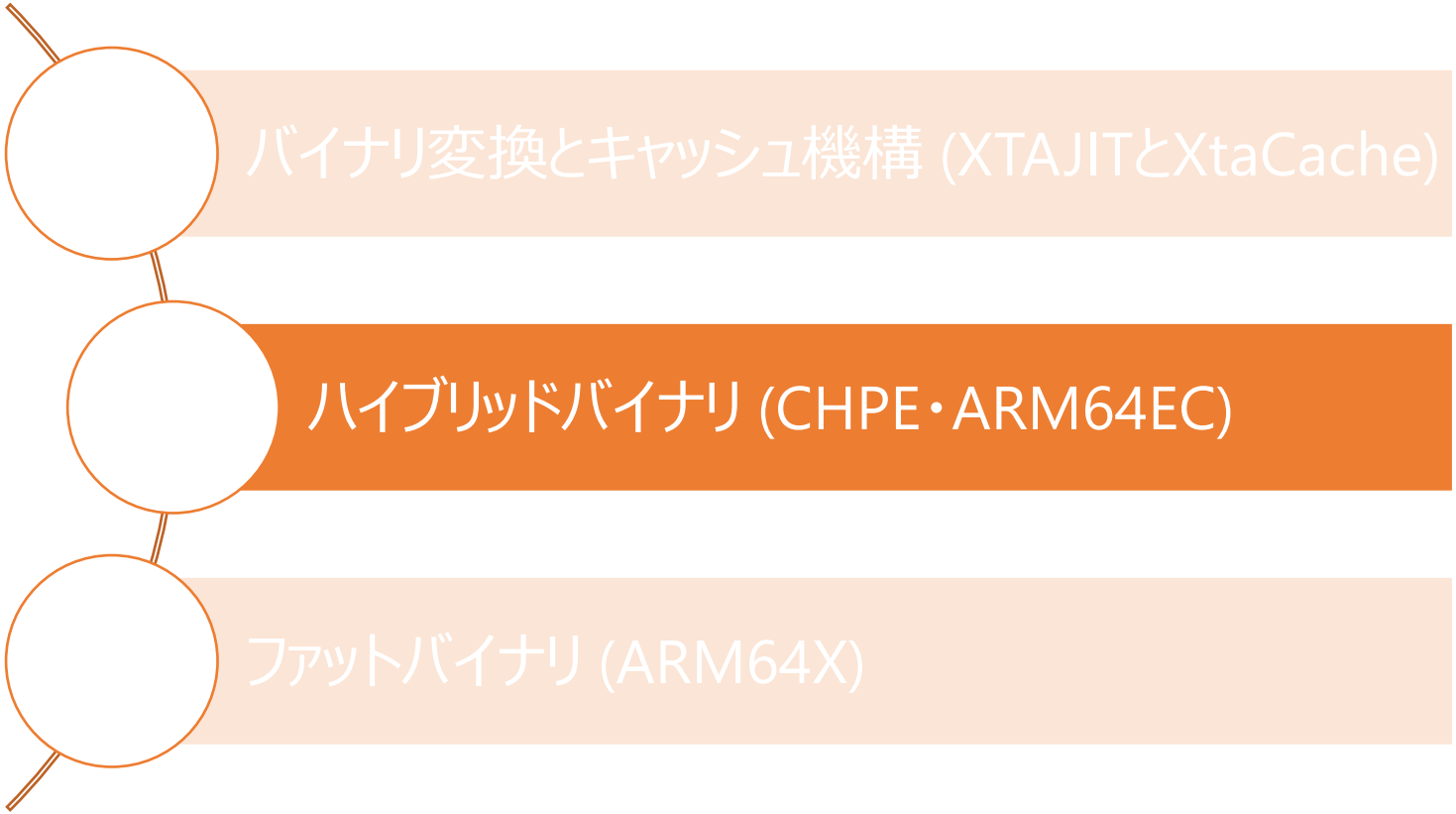
XTAJIT及びXTA Cache Fileを解析し以下の詳細を明らかにした

- x86/x64エミュレーション実行の流れ
- XTA Cache Fileの構造 (今回はARM64コードが難読化なしにそのまま存在する点のみ説明)

XTA Cache Hijackingというコードインジェクション手法を提案した

- 検知困難・追跡困難・持続性という性質を備える
- その他、Invisible Executionというこの手法特有の性質を備える

本研究で対象とする互換性テクノロジー



ハイブリッドバイナリ

Compiled Hybrid PE (CHPE)

x86とARM64の双方のコードを含むPE

CHPEの例:

- %SystemRoot%¥SyChpe32以下のシステムDLL (kernel32.dll, user32.dll など)
- ARM版Windows向けOfficeのEXE

x86エミュレーションでは、基本的にCHPEのシステムDLLが使われる

- SyChpe32にない場合、SysWOW64にあるシステムDLLが代わりに利用される

CHPEの特徴

x86 PEとの互換性を保ちつつ、ARM64ネイティブとほぼ同等のパフォーマンスを実現

CHPEは見かけ上x86 PEとして振る舞う

- 表層情報はx86であり、export関数の逆アセンブル結果もx86
  - export関数はexport thunkと呼ばれる
- CHPEは1プロセス内でx86 PEと混在して利用することが可能

x86 (func0のexport thunk)

```
mov edi, edi
push ebp
...
jmp #func0
```

export thunkはARM64コードへのjump stub

- jump先のARM64コードに関数の本体が含まれる

ARM64 (func0の関数の本体)

```
#func0
stp x29,x30,[sp, #-0x10]!
mov x29, sp
...
```

JITバイナリ変換はexport thunkについてのみ実行

- 関数全体について行う必要がなく、JITバイナリ変換量を削減
- そのため、パフォーマンス自体はARM64ネイティブで実行するときとほぼ同等に

ARM64 Emulation Compatible (ARM64EC)

ARM64EC (※ABI・ビルドアーキテクチャを総称してこのように呼称)

基本的にはCHPEをx64向けに利用可能な形にしたもの

CHPEと同様で…

- x64とARM64の双方のコードを含み、x64のコードはARM64コードへのjump stub
- ARM64ECとx64のPEを1つのプロセスで混在して利用可能

CHPEとの大きな違いは、SDKが公開されている点

- そのため、サードパーティーベンダーのARM64移行においても利用可能

https://blogs.windows.com/windowsdeveloper/2021/06/28/announcing-arm64ec-building-native-and-interoperable-apps-for-windows-11-on-arm/

CHPE/ARM64ECにおけるAPI呼び出し

CHPE/ARM64ECから外部のDLLの関数の呼び出しには、以下の2つの場合が存在

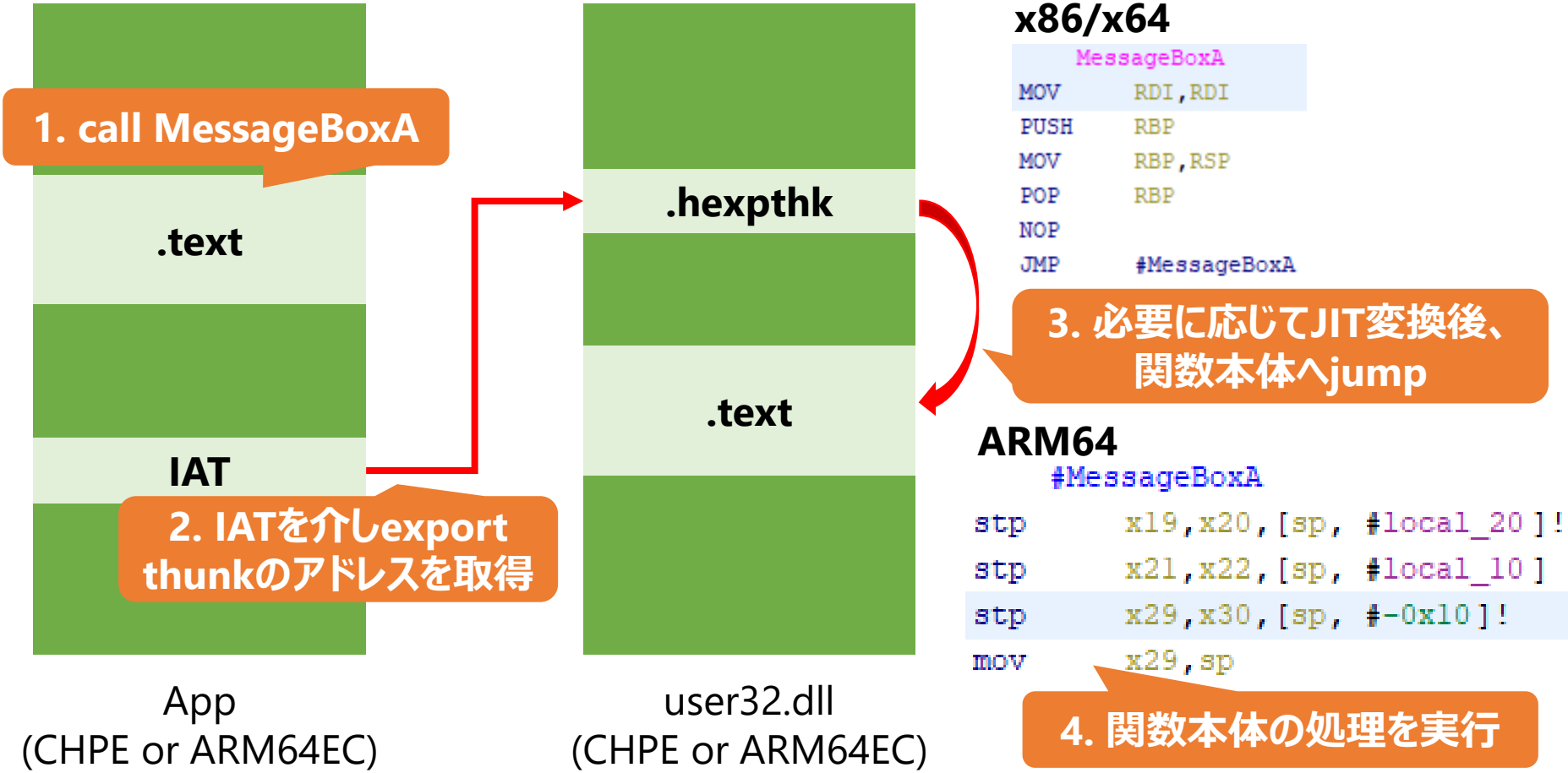
x86/x64のDLLの関数の呼び出し (CHPE/ARM64EC -> x86/x64)

- JITバイナリ変換 (or XTA Cache File)を使って処理を実行
- 2つの呼び出し規約に違いがあるため、呼び出し規約の変更も必要に

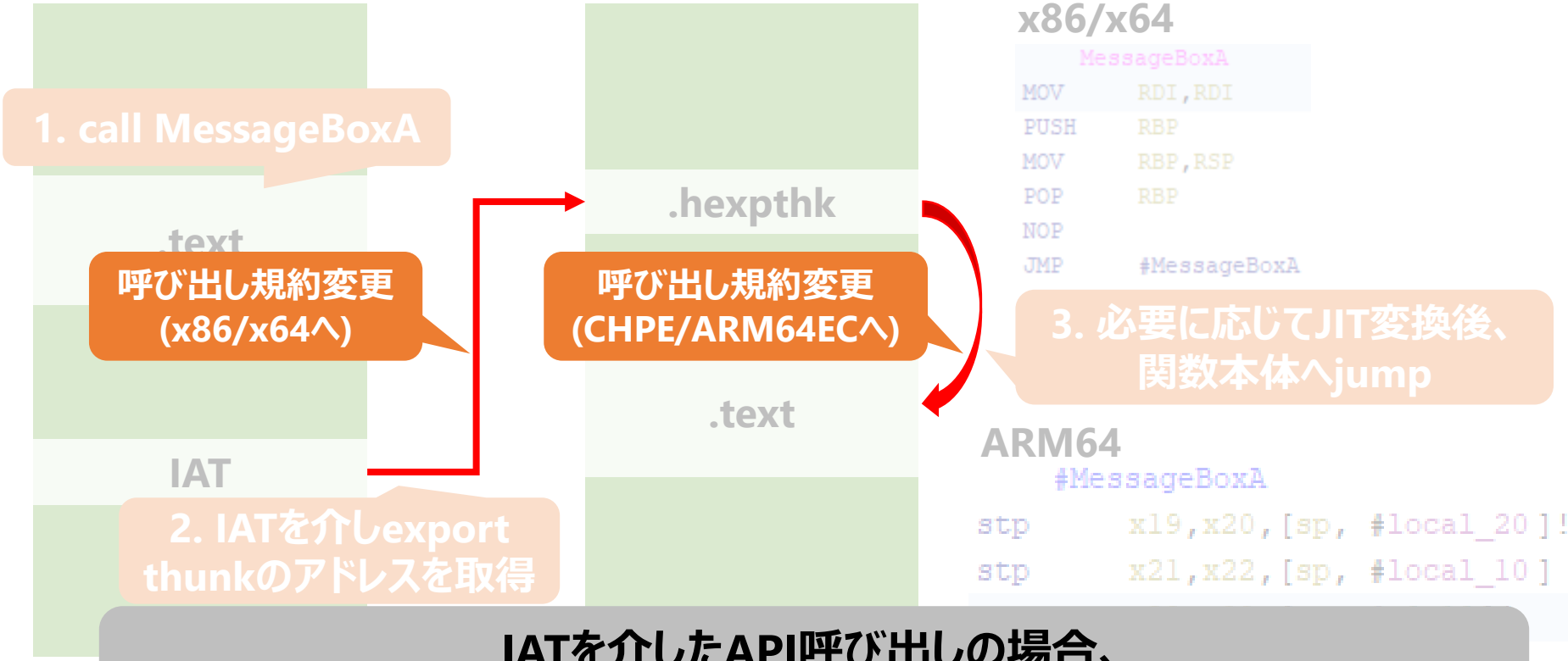
CHPE/ARM64ECのDLLの関数の呼び出し (CHPE/ARM64EC -> CHPE/ARM64EC)

- 呼び出し規約の変更やJITバイナリ変換は一見不要に思えるが…

CHPE/ARM64ECにおけるAPI呼び出しの流れ



CHPE/ARM64ECにおけるAPI呼び出しの流れ



IATを介したAPI呼び出しの場合、
export thunkの実行が経由されるため、呼び出し規約の変更が行われる

API呼び出しの最適化

export thunkは実行をスキップできるケースが多い

export thunkに含まれるコードのほとんどがホットパッチ用のコード

```
                    MessageBoxA
69e03db0   8b ff              MOV             EDI,EDI
69e03db2   55                 PUSH            EBP
69e03db3   8b ec              MOV             EBP,ESP
69e03db5   5d                 POP             EBP
69e03db6   90                 NOP
69e03db7   e9 c4 7b 0d 00     JMP             #MessageBoxA@16
```

ホットパッチ用のコード

Code hookingなどで先頭のコードの書き換えを行わない限り、このコードは実質何もしない

そういった特別なケース以外での実行をスキップできればAPI呼び出しの高速化につながる

- export thunkのJIT変換や呼び出し規約の変換処理を削減できるため

export thunkの実行をスキップして関数を呼び出せないか？

Hybrid Auxiliary IAT

Hybrid Auxiliary IAT: CHPE・ARM64ECに存在するもう一つIAT

Hybrid Auxiliary IATは必要に応じてexport thunkの実行のスキップを可能に

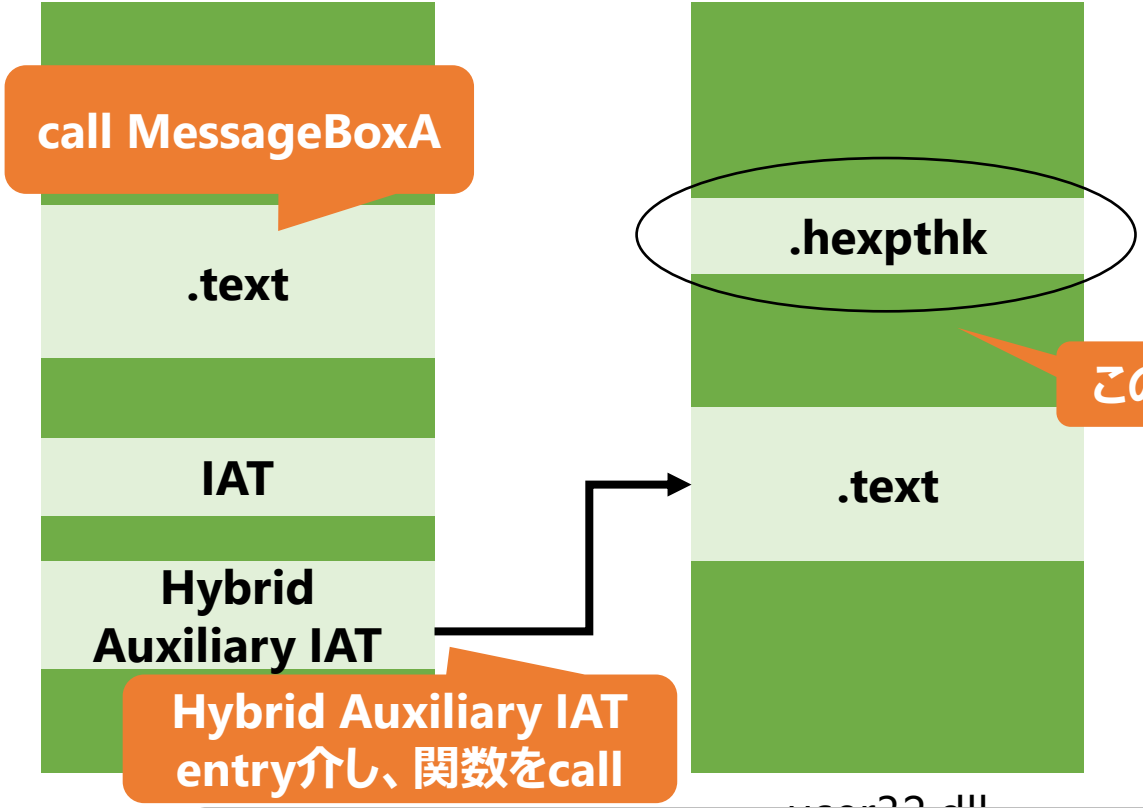
```
                    __imp_GetCurrentThreadId                XREF[1]:
                    .idata$9
                    __hybrid_auxiliary_iat
140008000 a0 20 00 40     addr      __impchk_GetCurrentThreadId
          01 00 00 00
```

プログラムローダーはIATの内容を参照し、Hybrid Auxiliary IATを実行時に変更※

- export thunkの実行が不要である場合、export thunkのjump先の実アドレスで上書き
- これにより、export thunkを経由しない形での関数呼び出しが可能に

※ntdll!#LdrpFastForwardAuxiliaryIat関数がこれを実行

Hybrid Auxiliary IATによるAPI呼び出しの流れ



x86/x64

```
MessageBoxA
MOV     RDI,RDI
PUSH    RBP
MOV     RBP,RSP
POP     RBP
NOP
JMP     #MessageBoxA
```

この部分の実行はスキップ

ARM64

```
#MessageBoxA
stp     x19,x20,[sp, #local_20]!
stp     x21,x22,[sp, #local_10]
stp     x29,x30,[sp, #-0x10]!
mov     x29,sp
```

CHPE d

export thunkの実行をスキップした上でのAPI呼び出しが可能に

CHPE・ARM64ECで使える2つのIAT hooking

古典的なIAT hooking

Hybrid Auxiliary IATがあるため、使えないように思われるが問題なく機能

- IATの権限変更を検知して、IATを参照した形でのAPI呼び出しに変更されるため

Hybrid Auxiliary IAT hooking

Hybrid Auxiliary IATエントリーを書き換えることによるIAT hooking

- IATエントリーを書き換えずに関数フックできる点が特徴
  - IATエントリーの内容からIAT hookingされていることを判別できない

対策: Hybrid Auxiliary IAT hooking

Hybrid Auxiliary IAT解析用のWinDbg拡張コマンド

https://github.com/FFRI/ProjectChameleon/tree/master/hybrid_aux_iat

```
0:024:ARM64EC> !dump powershell
Image Base is 00007ff79dc60000
Image Import Descriptor is 00007ff79dc78224
Image Load Config Directory is 00007ff79dc761b0
Module: OLE32.dll 00007ffea45e0000
                Name                IAT            Aux IAT     Aux IAT copy
    PropVariantClear 00007ffea45e27c0 00007ffea49cede0 00007ff79dc6c220
      CoUninitialize 00007ffea45e1af0 00007ffea48a9ae0 00007ff79dc6c320
      CoTaskMemAlloc 00007ffea45e19d0 00007ffea48aecf0 00007ff79dc6c260
      CoInitializeEx 00007ffea45e15a0 00007ffea48a94c0 00007ff79dc6c5a0
        CoInitialize 00007ffea3661100 00007ffea375b8a0 00007ff79dc6c1e0
    CoCreateInstance 00007ffea45e11c0 00007ffea496a770 00007ff79dc6c340
```

ここまでのまとめ

CHPE・ARM64ECの特徴と用途についてまとめた

API呼び出しにおいて、呼び出し規約の変換・JIT変換など多くのステップが必要な点も触れた

Hybrid Auxiliary IATとは何かについて明らかにした

呼び出し規約の変換・JIT変換を場合によりスキップし、API呼び出しの高速化を可能に

Hybrid Auxiliary IATの書き換えによる新しいAPI hooking手法を提案した

IATの内容からフックされていることを判別できないという特徴を持つ

本研究で対象とする互換性テクノロジー

バイナリ変換とキャッシュ機構 (XTAJITとXtaCache)

ハイブリッドバイナリ (CHPE・ARM64EC)

ファットバイナリ (ARM64X)

ARM版Windowsにおけるファットバイナリ

ARM64X

ARM64ネイティブとARM64EC向けの双方のコードを含む

- ファイルフォーマットは従来のPEと同じ
  - 新規に別のファイルフォーマットが用意されているわけではない
- 表層情報ではARM64と表示

ARM64Xとして提供されているファイルの例

- %SystemRoot%System32以下のシステムDLL
- cmd.exeやdllhost.exeなど一部のシステムEXE

ARM64Xの特徴

EXEの場合、親プロセスのアーキテクチャによって実行されるコードが変化

親プロセスのアーキテクチャ	実行されるコード
x86	ARM64
x64	ARM64EC
ARM64	ARM64
ARM64EC	ARM64EC

ARM64Xの特徴

DLLの場合、ARM64EC・x64・ARM64のすべてのプロセスからロード可能

表層情報はARM64だが、ARM64ECとx64プロセスからもロード可能

- x64プロセスがロードする場合 (x64 Chromeブラウザ実行時)

```
ModLoad: 00007ff7`aa6e0000 00007ff7`aa953000   chrome.exe
ModLoad: 00007ffd`89530000 00007ffd`8992b000   ntdll.dll
ModLoad: 00007ffd`87fd0000 00007ffd`880c4000   C:\WINDOWS\System32\xtajit64.dll
ModLoad: 00007ffd`868f0000 00007ffd`86a4c000   C:\WINDOWS\System32\KERNEL32.DLL
ModLoad: 00007ffd`852f0000 00007ffd`858e1000   C:\WINDOWS\System32\KERNELBASE.dll
```

- ARM64ネイティブプロセスがロードする場合 (ARM64 Edgeブラウザ実行時)

```
ModLoad: 00007ff7`6f120000 00007ff7`6f3bc000   msedge.exe
ModLoad: 00007ffd`89530000 00007ffd`8992b000   ntdll.dll
ModLoad: 00007ffd`868f0000 00007ffd`86a4c000   C:\WINDOWS\System32\KERNEL32.DLL
ModLoad: 00007ffd`852f0000 00007ffd`858e1000   C:\WINDOWS\System32\KERNELBASE.dll
ModLoad: 00007ffd`5f0a0000 00007ffd`5f19c000   C:\Program Files (x86)\Microsoft\Ed
```

DLLとしては同じものをロード

ARM64Xの特徴

ARM64EC・x64・ARM64のすべてのプロセスからロード可能

表層情報はARM64だが、ARM64ECとx64プロセスからもロード可能

- x64プロセスがロードする場合 (x64 Chromeブラウザ実行時)

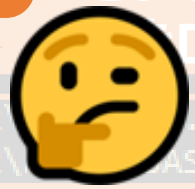
```
ModLoad: 00007ff7`aa6e0000 00007ff7`aa953000   chrome.exe
ModLoad: 00007ffd`89530000 00007ffd`8992b000   ntdll.dll
ModLoad: 00007ffd`87fd0000 00007ffd`880c4000   C:\WINDOWS\System32\xtajit64.dll
Mod                                            XERNEL32.DLL
Mod                                            XERNELBASE.dll
```

DLLのアーキテクチャ情報とプロセスのアーキテクチャ情報は通常一致しなければロードできないのでは？

- ARM

```
ModLoad: 00007ff7`6f120000 00007ff7`6f3bc000   msedge.exe
ModLoad: 00007ffd`89530000 00007ffd`8992b000   ntdll.dll
ModLoad: 00007ffd`868f0000 00007ffd`86a4c000   C:\WINDOWS\System32\                .DLL
ModLoad: 00007ffd`852f0000 00007ffd`858e1000   C:\WINDOWS\System32\                ASE.dll
ModLoad: 00007ffd`5f0a0000 00007ffd`5f19c000   C:\Program Files (x86)\Microsoft\Ed
```

LLとしては同じ
ロード



ARM64Xの特徴

ロードされたDLLの表層情報をWinDbgで調査

ARM64プロセス

```
0:000> !lmi ntdll
Loaded Module Info: [ntdll]
         Module: ntdll
   Base Address: 00007ff8cd190000
     Image Name: ntdll.dll
   Machine Type: 43620 (ARM64)
     Time Stamp: 29de4a9f (This is
           Size: 3f6000
       CheckSum: 3ed2bd
```

ntdll.dllのMachine Type はARM64

ARM64EC (or x64) プロセス

```
0:016:ARM64EC> !lmi ntdll
Loaded Module Info: [ntdll]
         Module: ntdll
   Base Address: 00007ff8cd190000
     Image Name: C:\WINDOWS\SYSTEM32\ntdll.dll
   Machine Type: 34404 (X64)
     Time Stamp: 29de4a9f (This is a reproducib
           Size: 3f6000
       CheckSum: 3ed2bd
```

ntdll.dllのMachine Type がx64？

同じファイルのはずが、DLLをロードした後で表層情報が変化？
ARM64EC (or x64) プロセスが使うときのみ、表層情報が変化するのでは？

ARM64Xに含まれる新しい再配置エントリー

ARM64とARM64ECの切り替えを行うために新しい再配置エントリーが追加

IMAGE_DYNAMIC_RELOCATION_ARM64X

- Dynamic Value Relocation Table (DVRT)の1つとして追加
- 以下ではDVRT ARM64Xと表記

対象プロセスのメモリにマップする前で適用され、種々の情報を動的に書き換え

- PEヘッダー(エントリーポイント・Export DirectoryのRVA・Machine Typeなど)
- コードセクションに含まれるAPI呼び出しの関数オフセット

この適用はカーネル側で (nt!MiApplyConditionalFixups) で行われる

DVRT ARM64Xのデータ構造

DVRT ARM64Xのデータ構造の詳細については以下の記事を参照

Discovering a new relocation entry of ARM64X in recent Windows 10 on Arm

3種類の再配置エントリーが存在

- Zero fill: 指定したアドレスの2/4/8バイトを0クリア
- Assign value: 指定したアドレスの2/4/8バイトを、指定した値で上書き
- Delta: 指定したアドレスに含まれる4バイトのデータに、4 or 8を足し引き

DVRT ARM64Xのデータ構造

DVRT ARM64Xのデータ構造の詳細については以下の記事を参照

[Discovering a new relocation entry of ARM64X in recent Windows 10 on Arm]

単純にオフセットを足すだけでなく、
Arbitrary Write可能な再配置エントリー
が存在する点が特徴※

3種類の再配置エントリーが存在

- Zero fill: 指定したアドレスの2/4/8バイト
- **Assign value: 指定したアドレスの2/4/8バイトを、指定した値で上書き**
- Delta: 指定したアドレスに含まれる4バイトのデータに、4 or 8を足し引き

※ARM64Xのイメージ内に限定される

DVRT ARM64Xの例: アーキテクチャ情報の書き換え

```
                                    RELOCATION_BLOCK
1803f4e04 00 00 00 00               ddw         0h
1803f4e08 30 00 00 00               ddw         30h
1803f4e0c ec 50                     dw          50ECh
1803f4e0e 64 86                     dw          8664h

1800000ec 64 aa 0c 00 9f  IMAGE_FILE_HEADER
          4a de 29 00 00
          00 00 00 00 00...
   1800000ec 64 aa           dw          AA64h       Machine
   1800000ee 0c 00           dw          Ch          NumberOfSections
   1800000f0 9f 4a de 29     ddw         29DE4A9Fh   TimeDateStamp
```

0x1800000ECにある2バイトのデータを0x8664で上書きすることを意味

0xAA64から0x8664へアーキテクチャ情報が上書きされる

DLLに含まれるアーキテクチャ情報がDVRT ARM64Xによりx64に変化
これにより、x64・ARM64ECプロセスからもロード可能に

ARM64X Relocation Obfuscation

DVRT ARM64Xを利用した難読化ができないか？

?

junk/fake PE header

junk/fake code

junk/fake IAT

DVRT ARM64X for decoding

再配置適用によりjunk/fakeデータを上書き

valid PE header

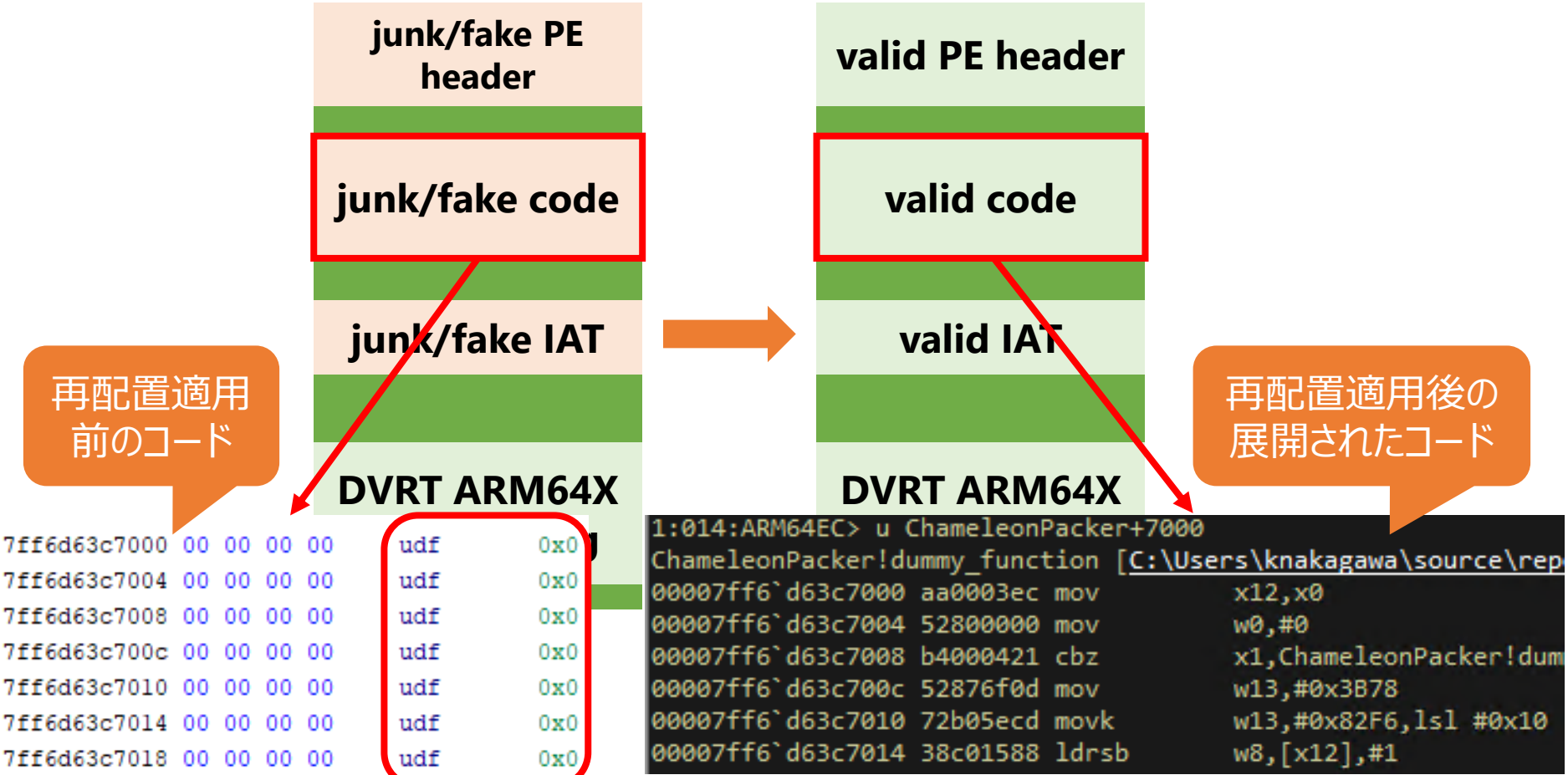
valid code

valid IAT

DVRT ARM64X for decoding

実行時には有効なコード・ヘッダーが展開され、正常に実行

悪用例: パッカー

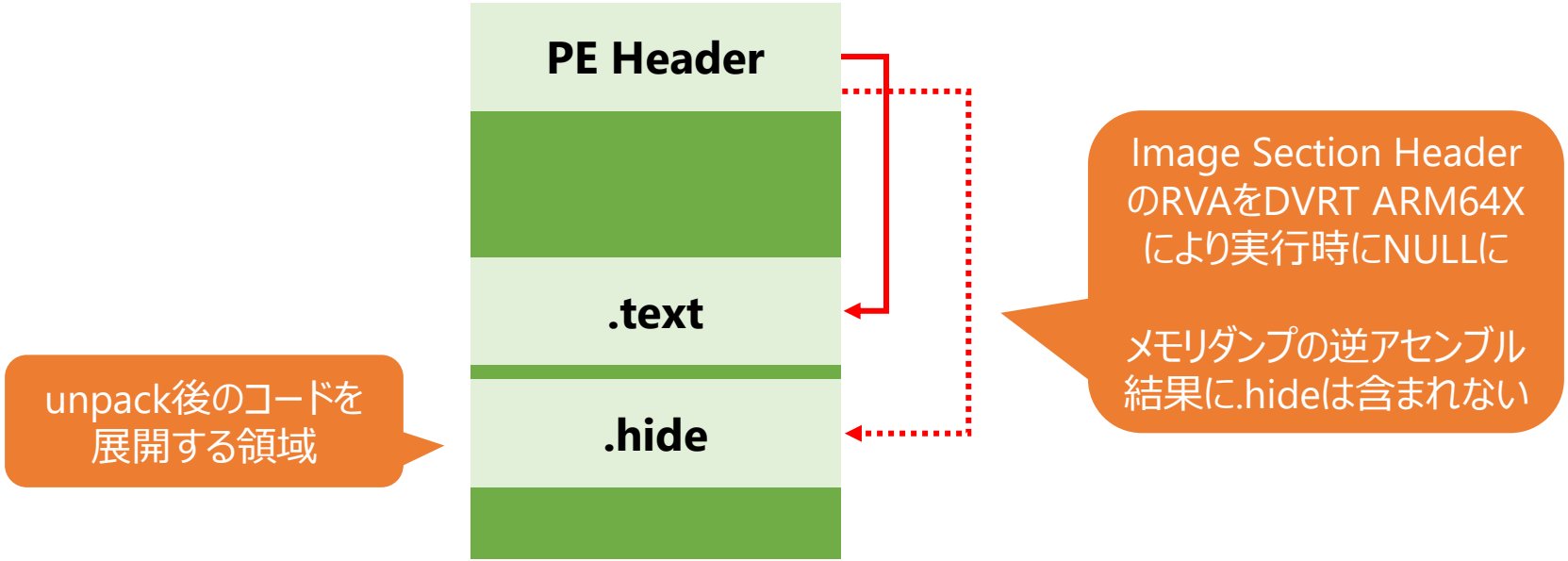

junk/fake PE header
junk/fake code
junk/fake IAT
DVRT ARM64X
valid PE header
valid code
valid IAT
DVRT ARM64X
再配置適用前のコード
再配置適用後の展開されたコード
7ff6d63c7000 00 00 00 00 udf 0x0
7ff6d63c7004 00 00 00 00 udf 0x0
7ff6d63c7008 00 00 00 00 udf 0x0
7ff6d63c700c 00 00 00 00 udf 0x0
7ff6d63c7010 00 00 00 00 udf 0x0
7ff6d63c7014 00 00 00 00 udf 0x0
7ff6d63c7018 00 00 00 00 udf 0x0
1:014:ARM64EC> u ChameleonPacker+7000
ChameleonPacker!dummy_function [C:\Users\knakagawa\source\rep
00007ff6`d63c7000 aa0003ec mov x12,x0
00007ff6`d63c7004 52800000 mov w0,#0
00007ff6`d63c7008 b4000421 cbz x1,ChameleonPacker!dum
00007ff6`d63c700c 52876f0d mov w13,#0x3B78
00007ff6`d63c7010 72b05ecd movk w13,#0x82F6,lsl #0x10
00007ff6`d63c7014 38c01588 ldrsb w8,[x12],#1

悪用例: パッカー

Q. ダンプしてしまえば解析は容易に行えるのでは？

A. メモリダンプ後の解析の妨害も可能

PEのセクションヘッダーをDVRT ARM64Xで編集して、逆アセンブル結果を騙すことが可能



悪用例: パッカー

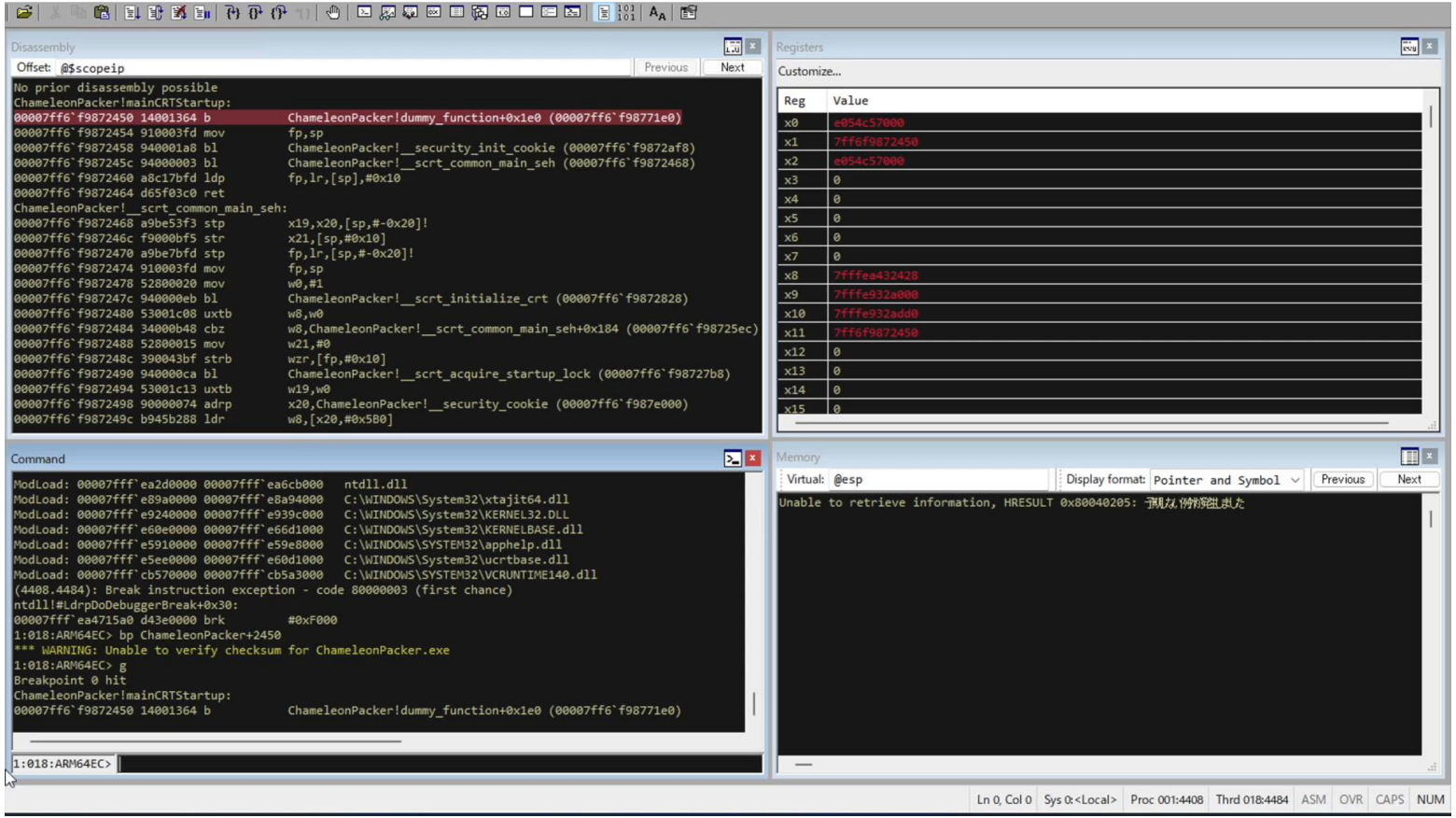
さらなる解析の妨害

ARM64XはARM64ネイティブとARM64EC向けのコードの双方を含む

- ARM64ECプロセスを実行中、ARM64ネイティブ向けのコードは実行されない

ARM64ネイティブ向けのコードが含まれている領域にあえてコードを展開し実行すると…

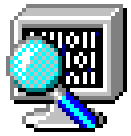
デモ


Disassembly
Offset: @$scopeip
Previous
Next
No prior disassembly possible
ChameleonPacker!mainCRTStartup:
00007ff6`f9872450 14001364 b ChameleonPacker!dummy_function+0x1e0 (00007ff6`f98771e0)
00007ff6`f9872454 910003fd mov fp,sp
00007ff6`f9872458 940001a8 bl ChameleonPacker!__security_init_cookie (00007ff6`f9872af8)
00007ff6`f987245c 94000003 bl ChameleonPacker!__scrt_common_main_seh (00007ff6`f9872468)
00007ff6`f9872460 a8c17bfd ldp fp,lr,[sp],#0x10
00007ff6`f9872464 d65f03c0 ret
ChameleonPacker!__scrt_common_main_seh:
00007ff6`f9872468 a9be53f3 stp x19,x20,[sp,#-0x20]!
00007ff6`f987246c f9000bf5 str x21,[sp,#0x10]
00007ff6`f9872470 a9be7bfd stp fp,lr,[sp,#-0x20]!
00007ff6`f9872474 910003fd mov fp,sp
00007ff6`f9872478 52800020 mov w0,#1
00007ff6`f987247c 940000eb bl ChameleonPacker!__scrt_initialize_crt (00007ff6`f9872828)
00007ff6`f9872480 53001c08 uxtb w8,w0
00007ff6`f9872484 34000b48 cbz w8,ChameleonPacker!__scrt_common_main_seh+0x184 (00007ff6`f98725ec)
00007ff6`f9872488 52800015 mov w21,#0
00007ff6`f987248c 390043bf strb wzr,[fp,#0x10]
00007ff6`f9872490 940000ca bl ChameleonPacker!__scrt_acquire_startup_lock (00007ff6`f98727b8)
00007ff6`f9872494 53001c13 uxtb w19,w0
00007ff6`f9872498 90000074 adrp x20,ChameleonPacker!__security_cookie (00007ff6`f987e000)
00007ff6`f987249c b945b288 ldr w8,[x20,#0x5B0]
Registers
Customize...
Reg
Value
x0 e054c57000
x1 7ff6f9872450
x2 e054c57000
x3 0
x4 0
x5 0
x6 0
x7 0
x8 7fffea432428
x9 7fffe932a000
x10 7fffe932add0
x11 7ff6f9872450
x12 0
x13 0
x14 0
x15 0
Command
ModLoad: 00007fff`ea2d0000 00007fff`ea6cb000 ntdll.dll
ModLoad: 00007fff`e89a0000 00007fff`e8a94000 C:\WINDOWS\System32\xtajit64.dll
ModLoad: 00007fff`e9240000 00007fff`e939c000 C:\WINDOWS\System32\KERNEL32.DLL
ModLoad: 00007fff`e60e0000 00007fff`e66d1000 C:\WINDOWS\System32\KERNELBASE.dll
ModLoad: 00007fff`e5910000 00007fff`e59e8000 C:\WINDOWS\SYSTEM32\apphelp.dll
ModLoad: 00007fff`e5ee0000 00007fff`e60d1000 C:\WINDOWS\System32\ucrtbase.dll
ModLoad: 00007fff`cb570000 00007fff`cb5a3000 C:\WINDOWS\SYSTEM32\VCRUNTIME140.dll
(4408.4484): Break instruction exception - code 80000003 (first chance)
ntdll!#LdrpDoDebuggerBreak+0x30:
00007fff`ea4715a0 d43e0000 brk #0xF000
1:018:ARM64EC> bp ChameleonPacker+2450
*** WARNING: Unable to verify checksum for ChameleonPacker.exe
1:018:ARM64EC> g
Breakpoint 0 hit
ChameleonPacker!mainCRTStartup:
00007ff6`f9872450 14001364 b ChameleonPacker!dummy_function+0x1e0 (00007ff6`f98771e0)
1:018:ARM64EC>
Memory
Virtual: @esp
Display format: Pointer and Symbol
Previous
Next
Unable to retrieve information, HRESULT 0x80040205: 予期なｴﾗｰが発生しました
Ln 0, Col 0
Sys 0:<Local>
Proc 001:4408
Thrd 018:4484
ASM
OVR
CAPS
NUM

悪用例: パッカー

WinDbgによる動的解析の妨害

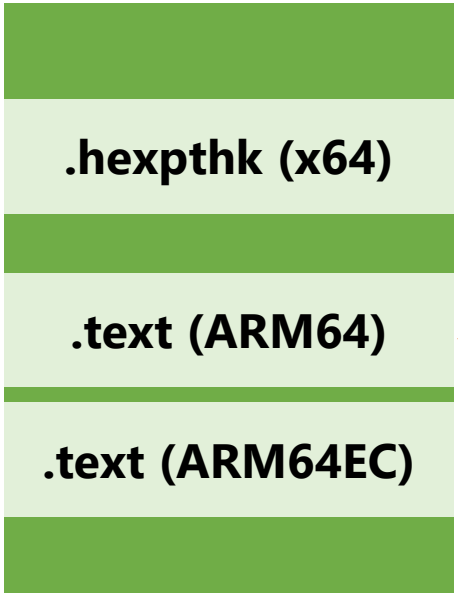
This is ARM64



Disassembly

Offset: @$scopeip

```
No prior disassembly possible
00007ff6`f9877209 61ec40c7 ???
00007ff6`f987720d c7426567 ???
00007ff6`f9877211 786ff040 ???
00007ff6`f9877215 40c70041 ???
00007ff6`f9877219 6c654808 ???
00007ff6`f987721d 0c40c76c ???
00007ff6`f9877221 0000006f ???
00007ff6`f9877225 fffe86e8 ???
```



ARM64ECプロセスでは
x64として解釈され実行

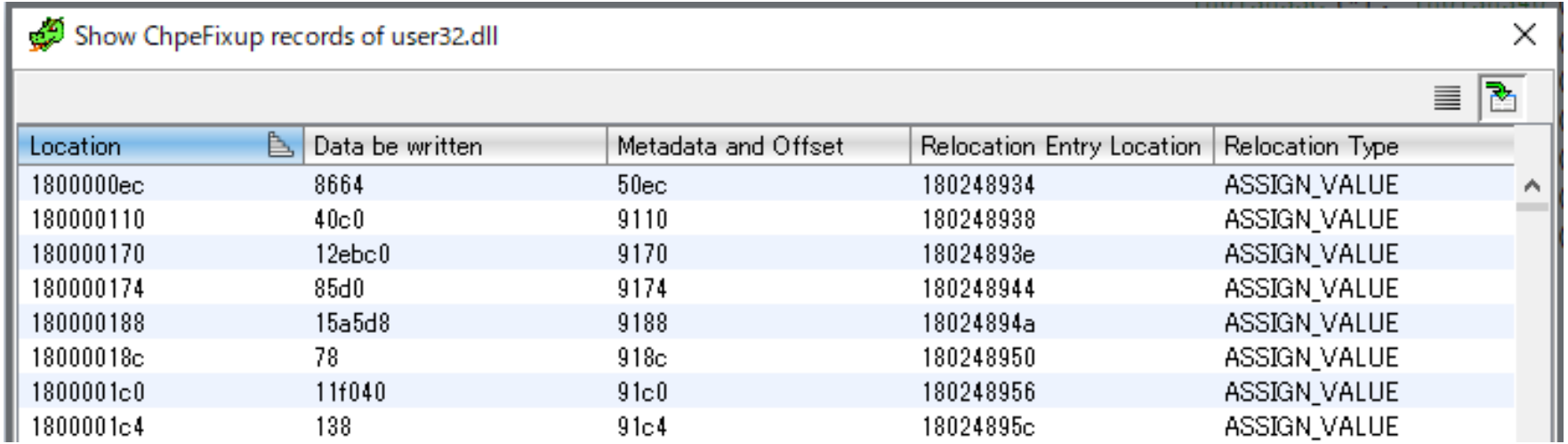
対策: ARM64X解析用のGhidraスクリプト

ARM64X解析用のGhidraスクリプト

ARM64XにDVRT ARM64Xの再配置を適用しファイルとして保存

- DVRT ARM64Xがパッカーとして使われていた場合、アンパック後の結果をファイルとして保存可能

その他、再配置エントリーの一覧を表示する機能も実装



Location	Data be written	Metadata and Offset	Relocation Entry Location	Relocation Type
1800000ec	8664	50ec	180248934	ASSIGN_VALUE
180000110	40c0	9110	180248938	ASSIGN_VALUE
180000170	12ebc0	9170	18024893e	ASSIGN_VALUE
180000174	85d0	9174	180248944	ASSIGN_VALUE
180000188	15a5d8	9188	18024894a	ASSIGN_VALUE
18000018c	78	918c	180248950	ASSIGN_VALUE
1800001c0	11f040	91c0	180248956	ASSIGN_VALUE
1800001c4	138	91c4	18024895c	ASSIGN_VALUE

ここまでのまとめ

ARM64Xとは何かとその特徴についてまとめた

ARM64XはARM64ネイティブとARM64EC向けの双方のコードを含むバイナリ

- 利用するプロセス・親プロセスに応じて実行されるコードが変化する特徴を持つ

ARM64Xに導入された新しい再配置エントリーを明らかにした

IMAGE_DYNAMIC_RELOCATION_ARM64X (DVRT ARM64X)と呼ばれる

- ARM64Xイメージ内でのArbitrary Write可能な再配置エントリーが存在する点が特徴

DVRT ARM64Xによる新しい難読化手法を提案した

ダンプ後の静的解析やWinDbgの動的解析を困難とする耐解析手法にもなる点を示した

全体のまとめとメッセージ

ARM版Windowsの互換性テクノロジーの解析結果を提示

これらの互換性テクノロジーを悪用した攻撃手法について紹介

発表されて間もないもので、他にも様々な攻撃手法が存在する可能性がある

今後もさらなる研究が必要

今回発表した攻撃手法のコンセプト自体は広く適用可能なもの

例えば、キャッシュ機構は、バイナリ変換を実装する際に高速化のために実装するのが自然

- 今後、類似の互換性テクノロジーが登場した際にも、同様の攻撃手法が適用出来る可能性は高い

ARMベースのノートPCの本格的な普及を前にして、
十分な対策が行われることを期待

ツール・PoCコードへのリンク

XTA Cache File関連

https://github.com/FFRI/XtaTools (XTA Cache HijackingのPoCコード)

https://github.com/FFRI/radare2 (XTA Cache Fileの解析用のradare2)

Black Hat EU 2020発表スライド (ファイルフォーマット詳細やXTA Cache Hijackingの詳細)

ARM64EC・ARM64X関連

https://github.com/FFRI/ProjectChameleon (PoCコード・ツール・解析ドキュメント)

https://ffri.github.io/ProjectChameleon/ (解析結果をまとめたドキュメント)

Thank you!

質問・コメントは以下の連絡先まで

Twitter DM: @FFRI_Research

email: research-feedback@ffri.jp