# 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

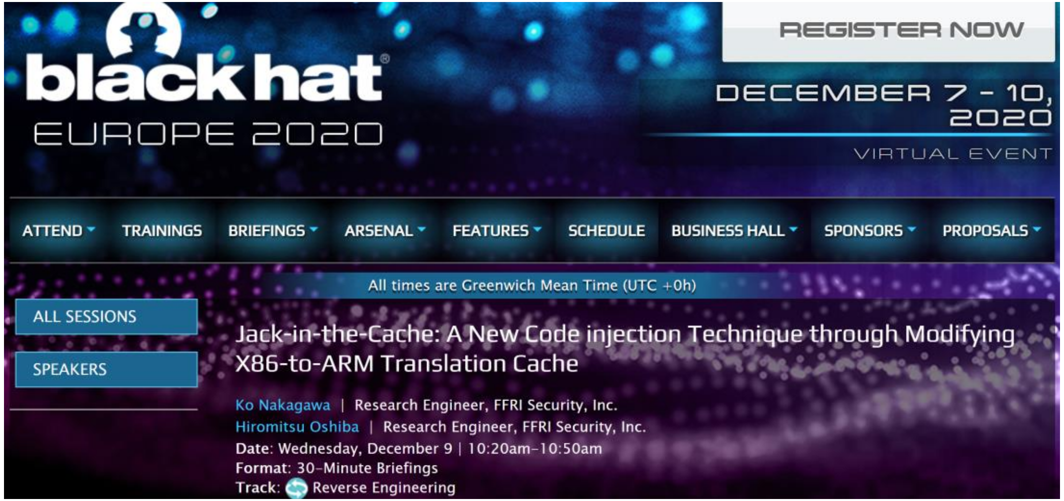
# About me

Joined FFRI Security, Inc. after graduating.

- Working as a research engineer at the basic research lab.
- Recently reverse-engineering compatibility technology of Windows on ARM and M1 Mac.
- Black Hat EU 2020 Briefings Speaker

Maxwell's Equations
$\nabla \cdot B = 0$
$\nabla \cdot D = \rho$
$\nabla \times E = -\frac{\partial B}{\partial t}$
$\nabla \times H = \frac{\partial D}{\partial t} + J$

GitHub: https://github.com/kohnakagawa



All times are Greenwich Mean Time (UTC +0h)

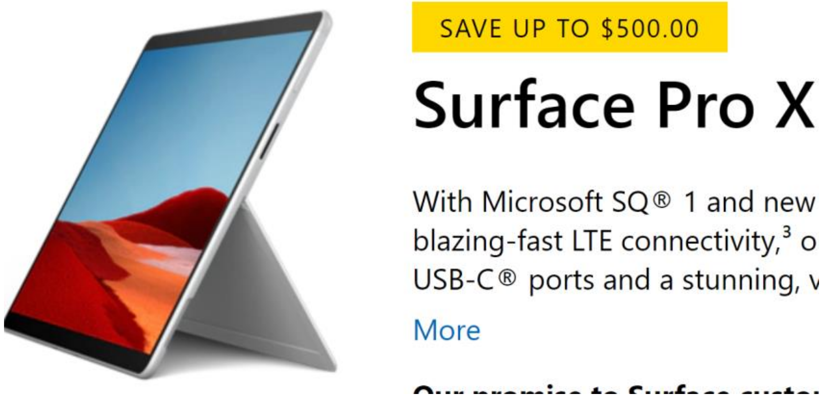
Jack-in-the-Cache: A New Code injection Technique through Modifying X86-to-ARM Translation Cache

Ko Nakagawa | Research Engineer, FFRI Security, Inc.
Hiromitsu Oshiba | Research Engineer, FFRI Security, Inc.
Date: Wednesday, December 9 | 10:20am–10:50am
Format: 30-Minute Briefings
Track: Reverse Engineering

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

# The Rise of ARM

Surface Pro X Windows on ARM



SAVE UP TO $500.00

Surface Pro X

With Microsoft SQ® 1 and new blazing-fast LTE connectivity,3 o USB-C® ports and a stunning, v

More

Our promise to Surface custo

[Surface Pro X – Ultra-thin & Always Connected 2-in-1 Laptop – Microsoft Surface]()

M1 Mac

[MacBook Pro 13-inch - Apple]()

**ARM Based Laptops are being released one after another.**

※ARM notation in accordance with Microsoft's Documentation Standards

# Why an ARM Based Laptop?

ARM has superior power usage related functionality.

Many have a longer battery life compared to previous devices※.

- Always Connected PCs, laptops that can be used like a smartphone

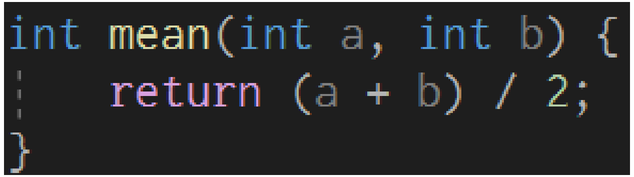
In the new normal, work will not be restricted by time of day or location

- Long battery life is crucial.

**ARM based laptops will continue to rise in demand.**

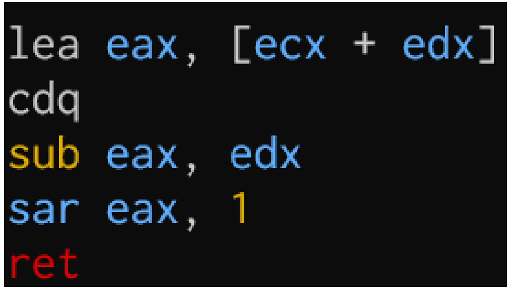
※Windows on ARM Benchmarked > Impressions, Superb Battery Life, What Needs to Improve | TechSpot

# The Application Compatibility Problem



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

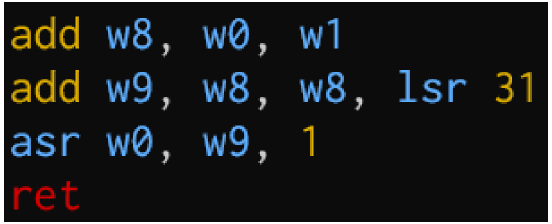
**x86/x64**



```
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
```

**We cannot use existing software for x86/x64 on ARM-based laptops.**

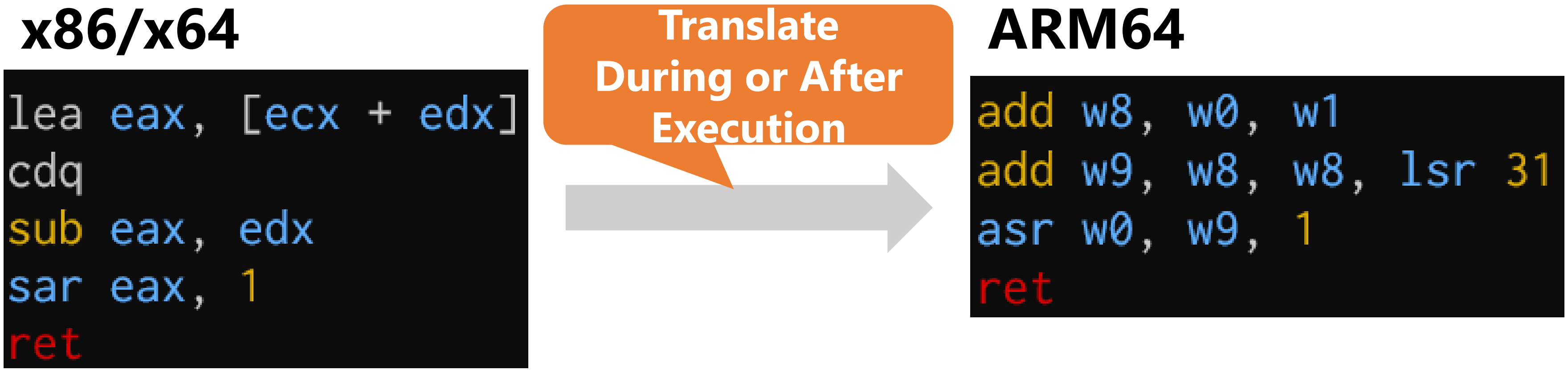
# The Solution to The Incompatibility Problem

## Binary Translation and Caching Mechanism

Translate x86/x64 to ARM64

The translation is resource-intensive, a caching mechanism is used for acceleration.

- Cache the execution result as a file and reuse in subsequent executions.



**x86/x64**

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

**Translate During or After Execution**

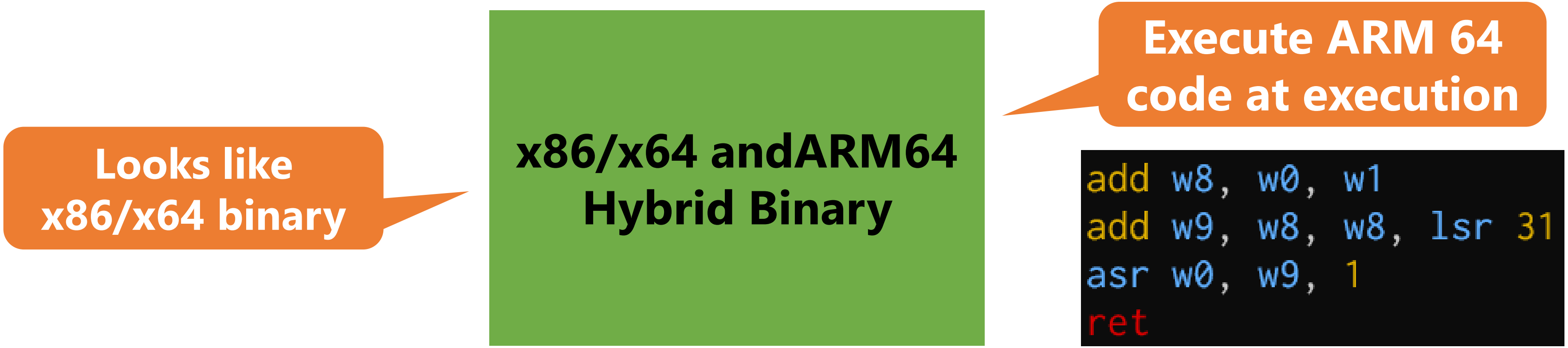
**ARM64**

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

# The Solution to The Incompatibility Problem

## Hybrid Binary

Maintain compatibility with current binary while allowing code execution at fast speeds close to native code execution.



Looks like x86/x64 binary

**x86/x64 andARM64 Hybrid Binary**

Execute ARM 64 code at execution

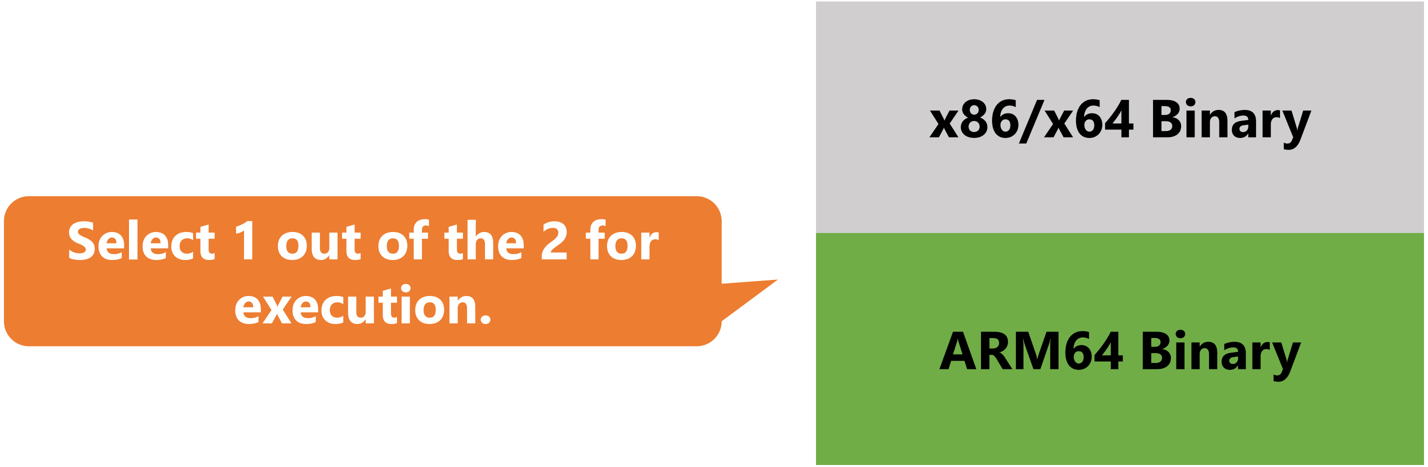
```
add w8, w0, w1
add w9, w8, w8, lsr 31
asr w0, w9, 1
ret
```

# The Solution to The Incompatibility Problem

## Fat Binary

A binary that combines multiple binaries for various architectures.

- Allows using a single binary for multiple uses or multiple platforms.



x86/x64 Binary

ARM64 Binary

Select 1 out of the 2 for execution.

# Windows and macOS Implementation

| Compatibility Technology | Windows Implementation | macOS Implementation |
|---|---|---|
| Binary Translation and Caching Mechanism | XTAJIT and XtaCache | Rosetta 2 |
| Hybrid Binary | CHPE・ARM64EC | N/A |
| Fat Binary | ARM64X | Universal 2 |

**Both Windows on ARM and M1 Mac have implemented compatibility technologies.**

# Problem: Malicious Usage of Compatibility Technologies

There have been cases where compatibility technologies have been used for malicious intent.

Ex: Application Shimming

The possibility of malicious usage of compatibility technologies implemented in Windows on ARM and M1 Mac?

-> **As far as we know, not even discussed.**

**Details of the compatibility technologies aren’t being disclosed to begin with.**

Information disclosed by Microsoft or Apple is very limited and there aren’t that many reverse engineering findings neither.

# Purpose of this research

To discover attack methods leveraging the malicious use of the newly implemented compatibility technologies.

Specifically…

- Details of the compatibility technologies. (※ Including acceleration technologies often included in compatibility technologies.)
- Malicious use of such technologies.

To discover/disclose the above 2 aims.

**We hope that this research will lead to increased security research of compatibility technologies.**

# Compatibility technologies examined

| Compatibility Technology | Windows Implementation | macOS Implementation |
|---|---|---|
| Binary Translation and Caching Mechanism | XTAJIT and XtaCache | Rosetta 2 |
| Hybrid Binary | CHPE･ARM64EC | N/A |
| Fat Binary | ARM64X | Universal 2 |

**3 Compatibility technologies implemented in Windows on ARM are the subject.**

# Compatibility technologies examined

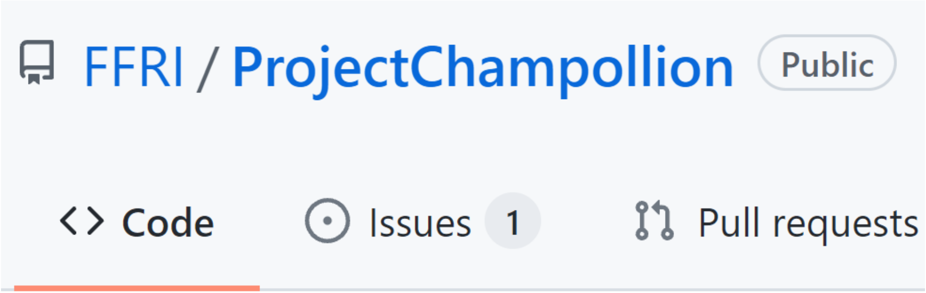
Regarding macOS…

| Compatibility Technology | Windows Implementation | macOS Implementation |
|---|---|---|
| Binary Translation and Caching Mechanism | XTAJIT and XtaCache | Rosetta 2 |
| Hybrid Binary | CHPE･ARM64EC | N/A |
| Fat Binary | ARM64X | Universal 2 |

**3 Compatibility technologies implemented in the Windows on ARM are the subject.**

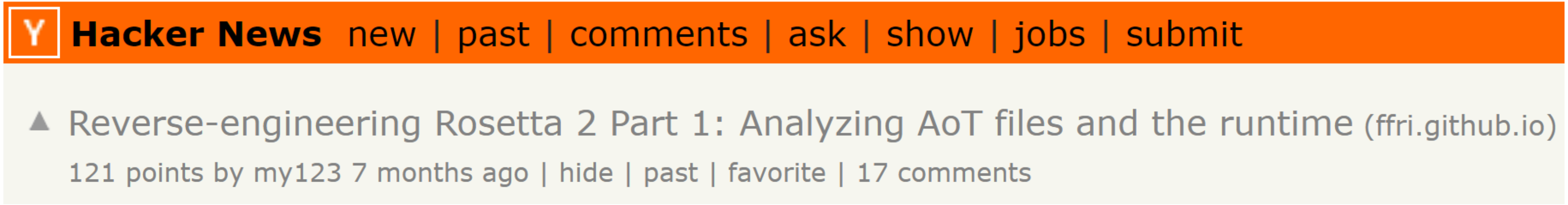
# Project Champollion

Repository of macOS Rosetta 2 related reverse engineering results.



FFRI / ProjectChampollion Public

https://github.com/FFRI/ProjectChampollion



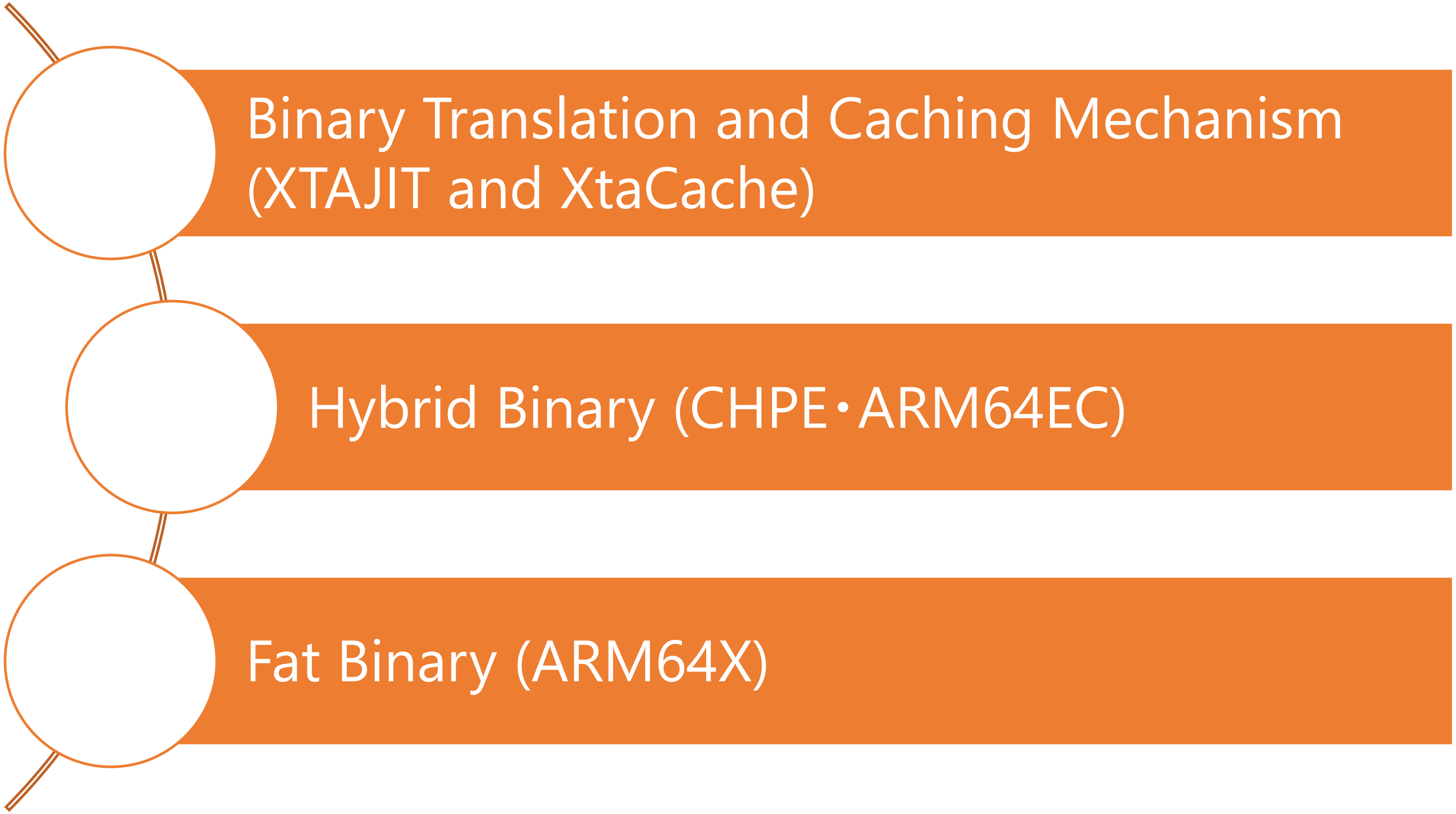
Y
Hacker News new | past | comments | ask | show | jobs | submit

▲ Reverse-engineering Rosetta 2 Part 1: Analyzing AoT files and the runtime (ffri.github.io)

121 points by my123 7 months ago | hide | past | favorite | 17 comments

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

# Compatibility technologies examined

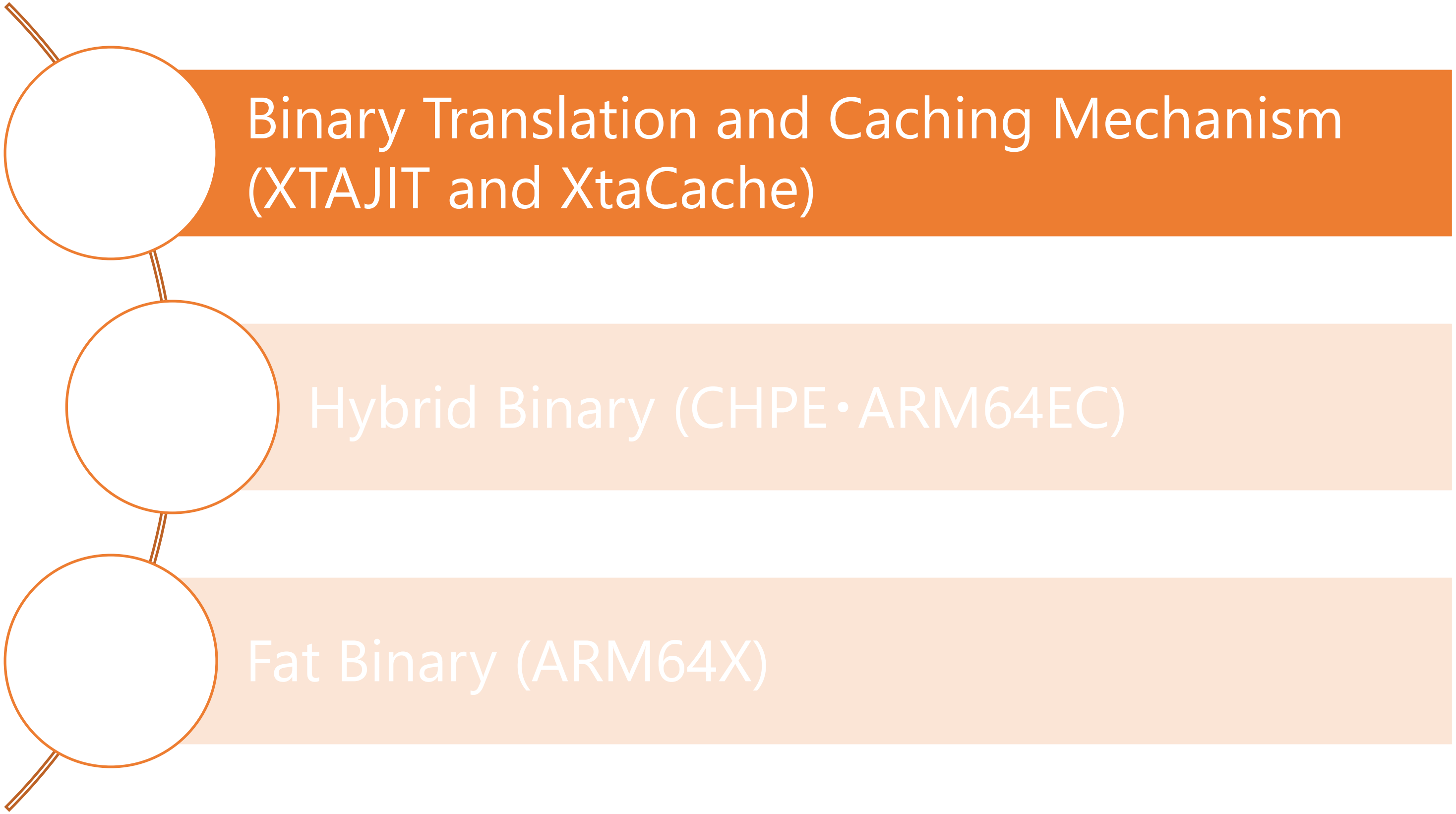


Binary Translation and Caching Mechanism (XTAJIT and XtaCache)

Hybrid Binary (CHPE・ARM64EC)

Fat Binary (ARM64X)

# Compatibility technologies examined



- Binary Translation and Caching Mechanism (XTAJIT and XtaCache)
- Hybrid Binary (CHPE・ARM64EC)
- Fat Binary (ARM64X)

# Binary translation and Caching Mechanism in Windows on ARM

## Binary Translation: XTAJIT (※XTA is suspected an acronym for X86-To-ARM)

JIT Binary translation of x86/x64 code to ARM 64 code upon execution.

- Windows 10 Insider Preview and Windows 11 also support x64

## Caching Mechanism: XtaCache

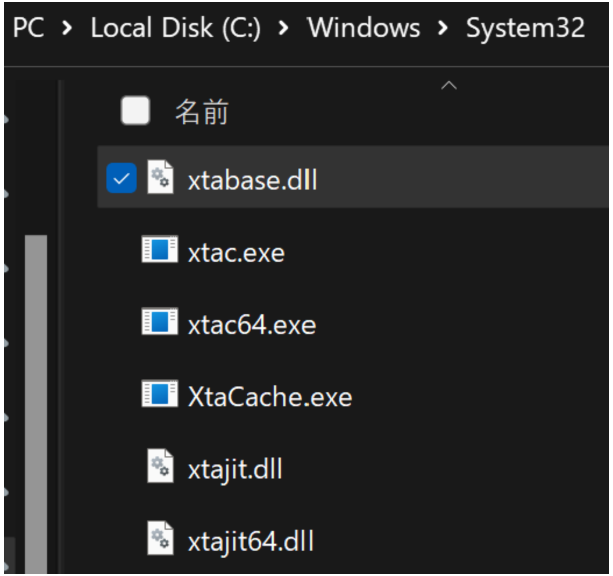
Translation results are saved as a XTA Cache File and reused in subsequent re-executions.

- Decrease JIT binary translation overhead and accelerate execution of application after the 2nd time.

# X86/x64 Emulation

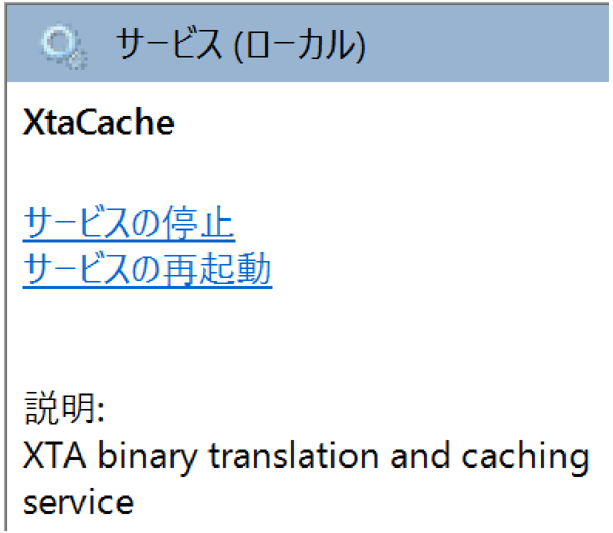
## EXE and DLLs related to x86/x64 emulation

- xtajit.dll/xtajit64.dll: x86/x64 emulator DLL
- xtac.exe/xtac64.exe: Compiler to create XTA Cache File
- XtaCache.exe: Management service of XTA Cache File



PC › Local Disk (C:) › Windows › System32

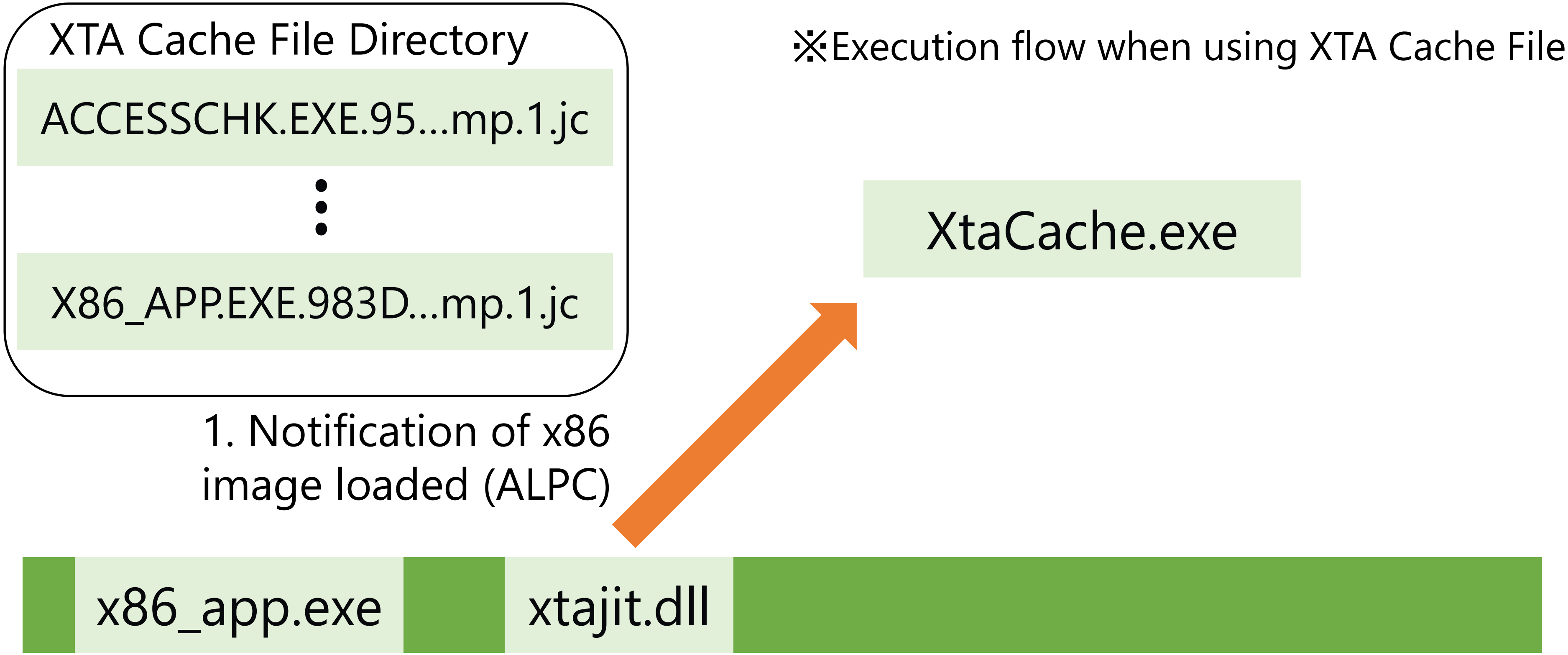
☑ xtabase.dll
xtac.exe
xtac64.exe
XtaCache.exe
xtajit.dll
xtajit64.dll



XtaCache

説明:
XTA binary translation and caching service

# X86/x64 Emulation Flow



※Execution flow when using XTA Cache File

XTA Cache File Directory

ACCESSCHK.EXE.95…mp.1.jc

⋮

X86_APP.EXE.983D…mp.1.jc

XtaCache.exe

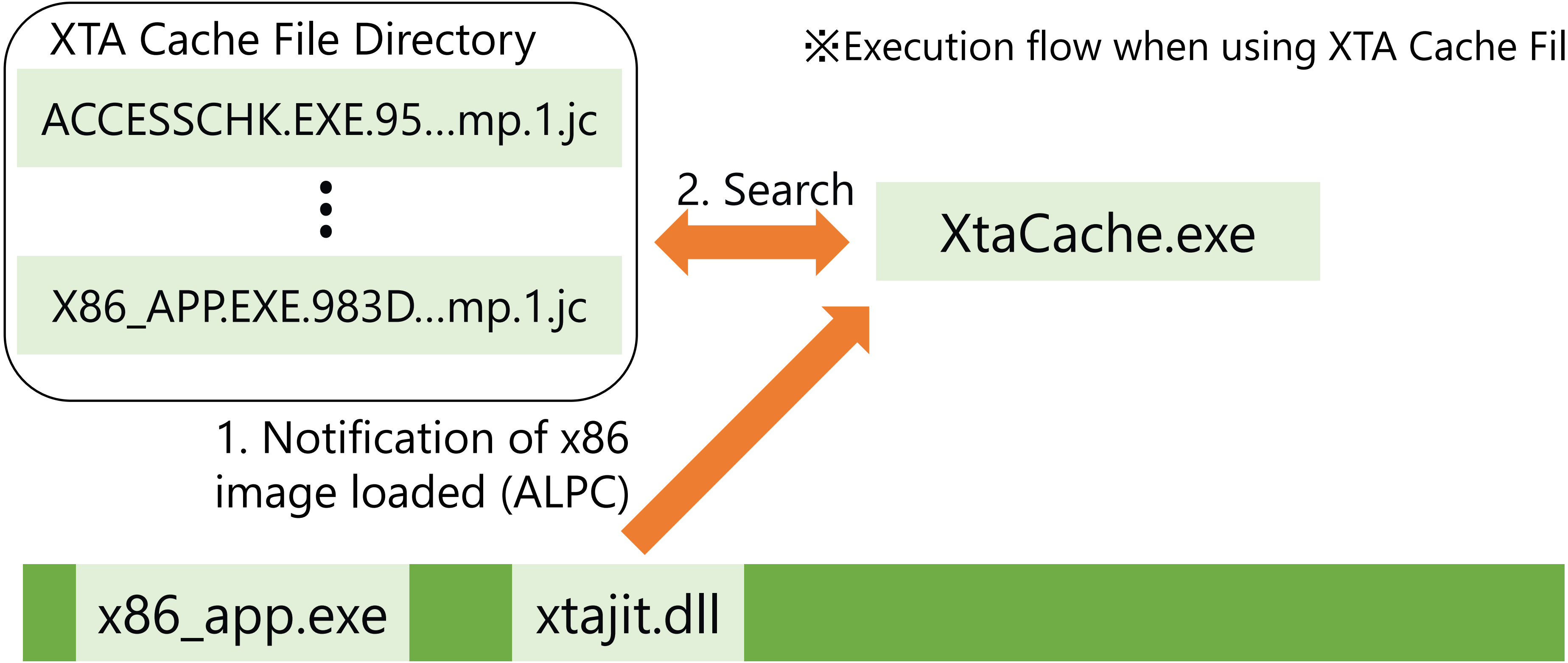
1. Notification of x86 image loaded (ALPC)

x86_app.exe

xtajit.dll

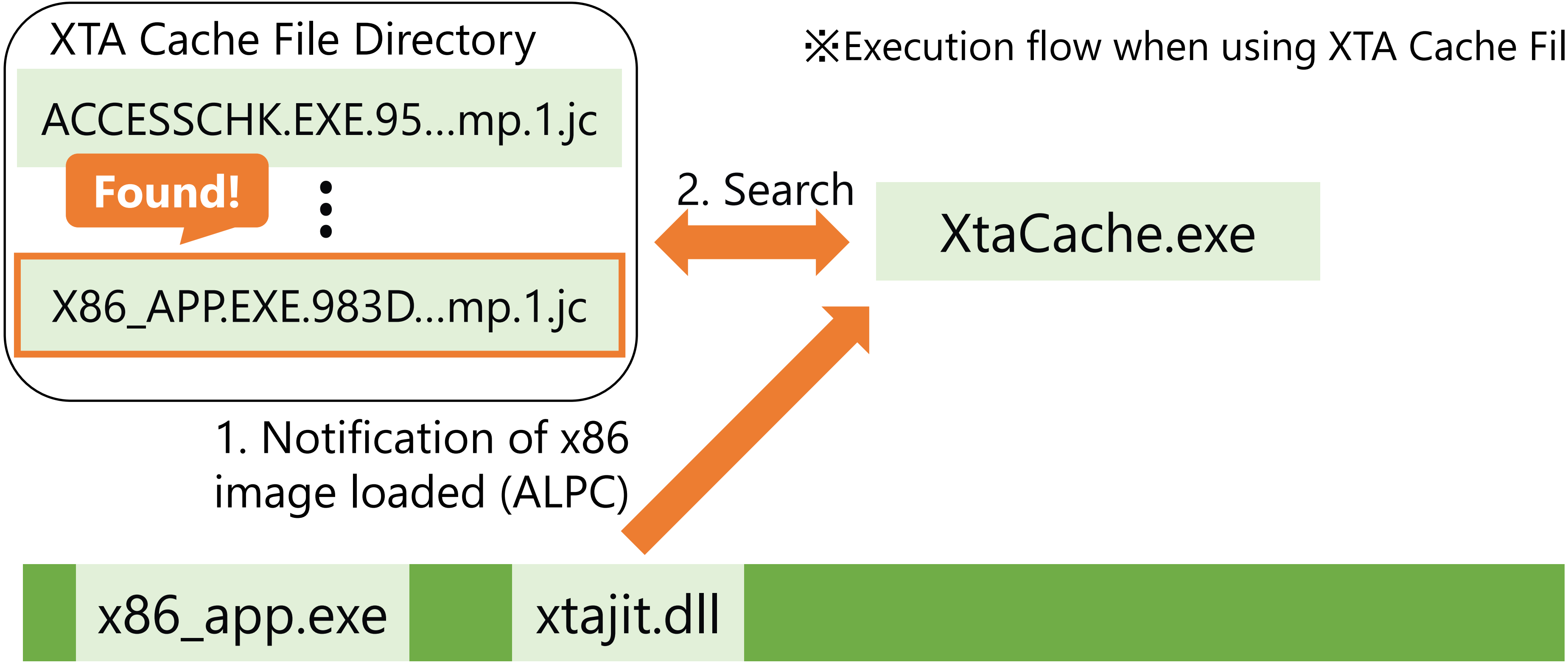
# X86/x64 Emulation Flow

※Execution flow when using XTA Cache File



# X86/x64 Emulation Flow

※Execution flow when using XTA Cache File



XTA Cache File Directory

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

Found!

⋮

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

2. Search

XtaCache.exe

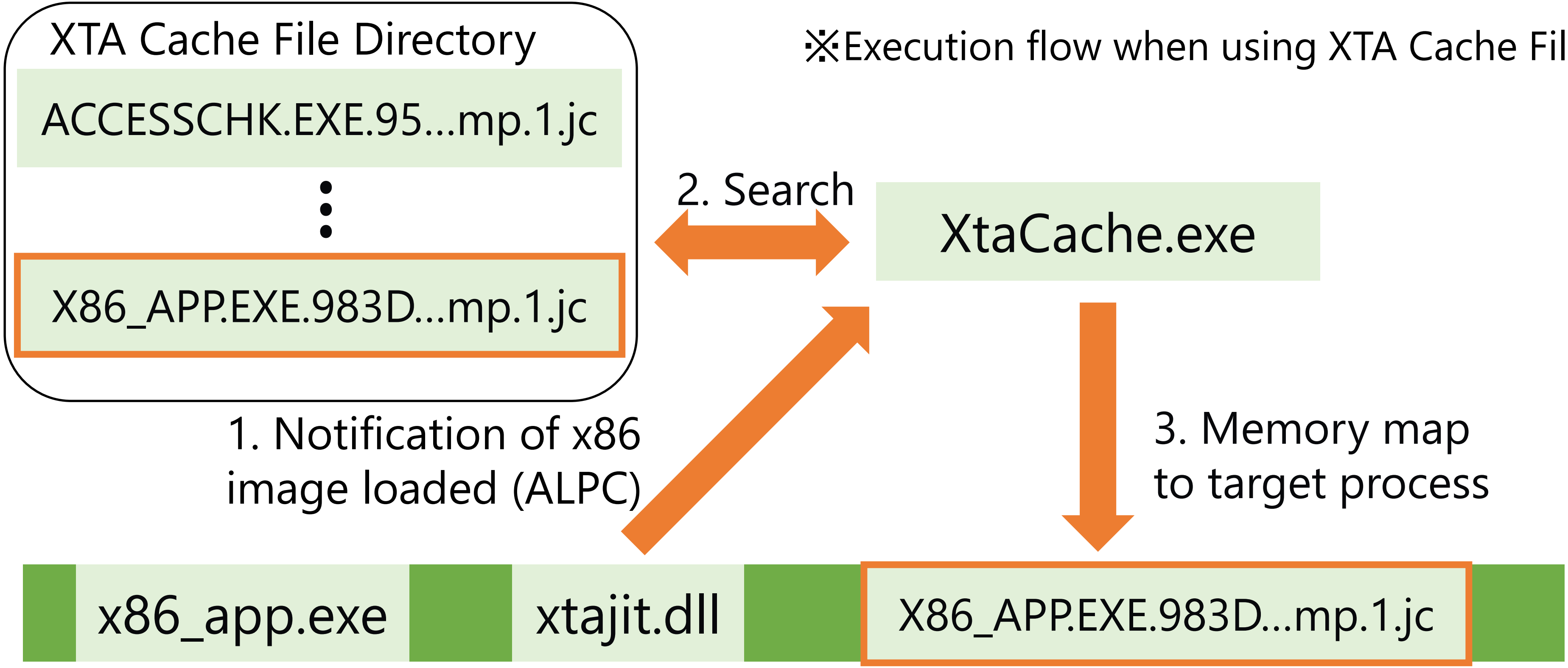
1. Notification of x86 image loaded (ALPC)

x86_app.exe

xtajit.dll

# X86/x64 Emulation Flow

※Execution flow when using XTA Cache File



XTA Cache File Directory

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

⋮

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

2. Search

XtaCache.exe

1. Notification of x86 image loaded (ALPC)

3. Memory map to target process

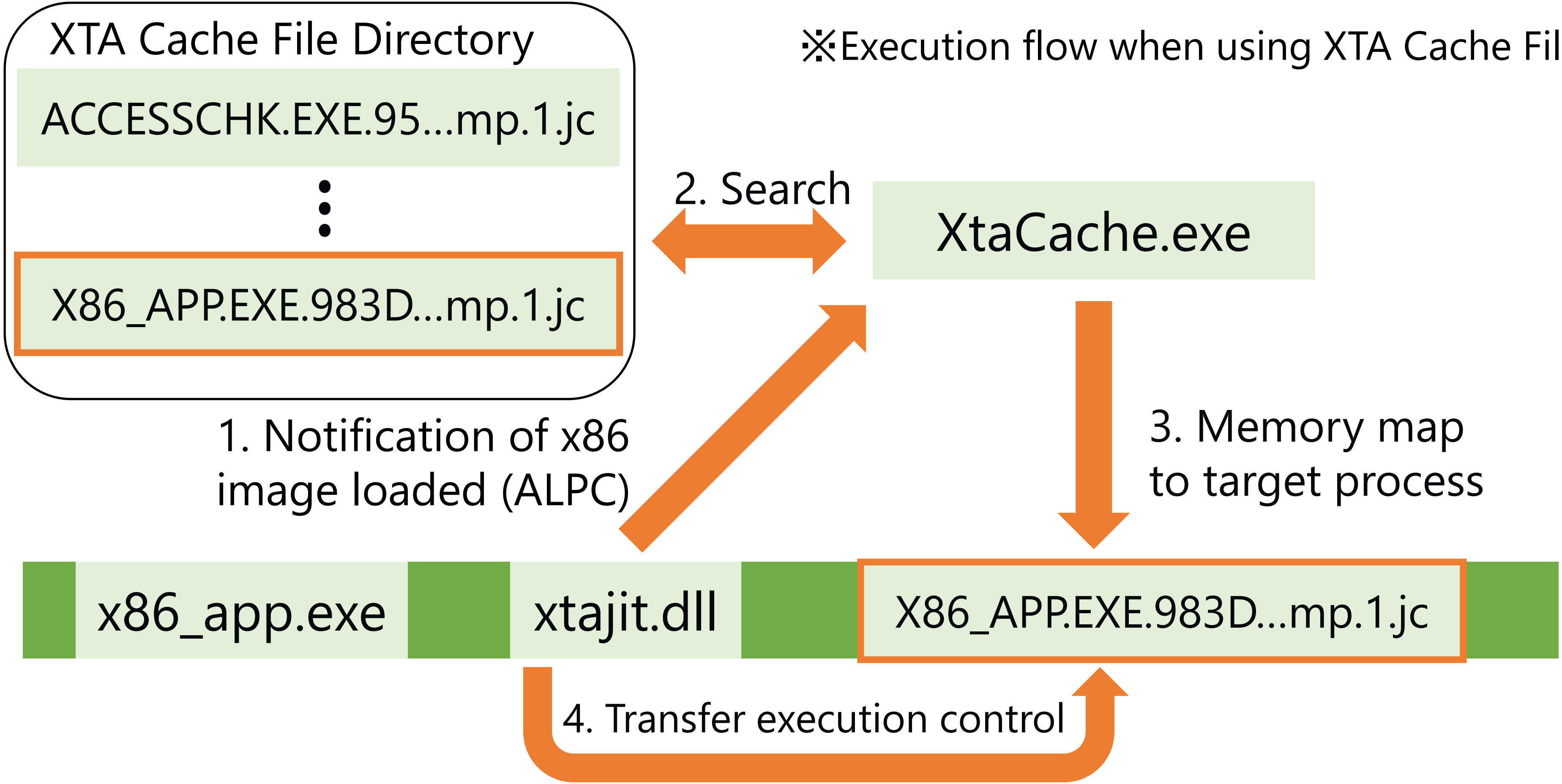
x86_app.exe

xtajit.dll

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

# X86/x64 Emulation Flow

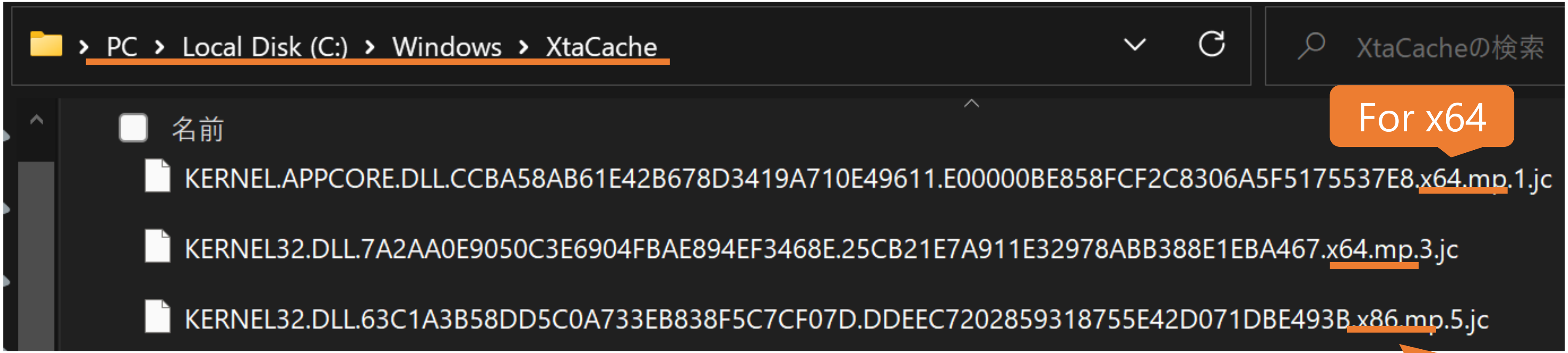
※Execution flow when using XTA Cache File


XTA Cache File Directory
ACCESSCHK.EXE.95...mp.1.jc
X86_APP.EXE.983D...mp.1.jc
2. Search
XtaCache.exe
1. Notification of x86 image loaded (ALPC)
3. Memory map to target process
x86_app.exe
xtajit.dll
X86_APP.EXE.983D...mp.1.jc
4. Transfer execution control

# XTA Cache File

XTA Cache File exists in %SystemRoot%¥XtaCache

XTA Cache File is created for each x86 and x64 PE



KERNEL.APPCORE.DLL.CCBA58AB61E42B678D3419A710E49611.E00000BE858FCF2C8306A5F5175537E8.x64.mp.1.jc

KERNEL32.DLL.7A2AA0E9050C3E6904FBAE894EF3468E.25CB21E7A911E32978ABB388E1EBA467.x64.mp.3.jc

KERNEL32.DLL.63C1A3B58DD5C0A733EB838F5C7CF07D.DDEEC7202859318755E42D071DBE493B.x86.mp.5.jc

For x64

For x86

Only the XtaCache service can access these files by default.

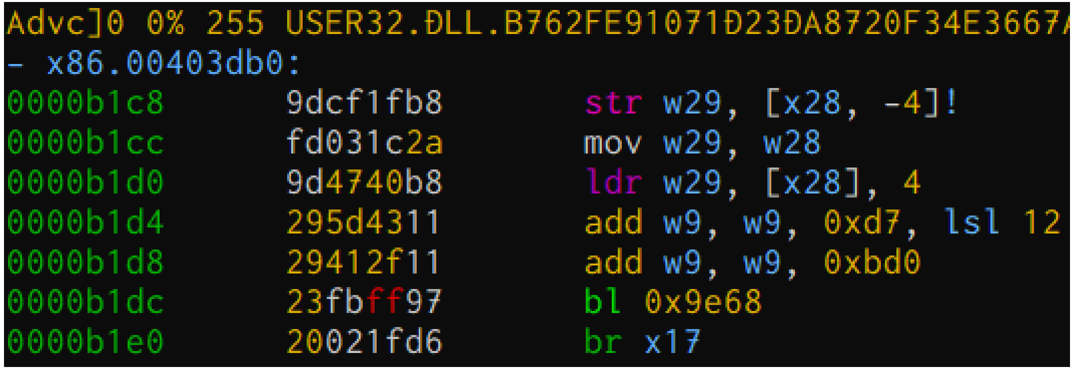
- But this settings can be changed by administrator access privileges.

# XTA Cache File Structure

Refer to Black Hat EU 2020 presentation material for more details about the file format

Parser for XTA Cache File and analysis tool for contained ARM64 code are already public

https://github.com/FFRI/XtaTools



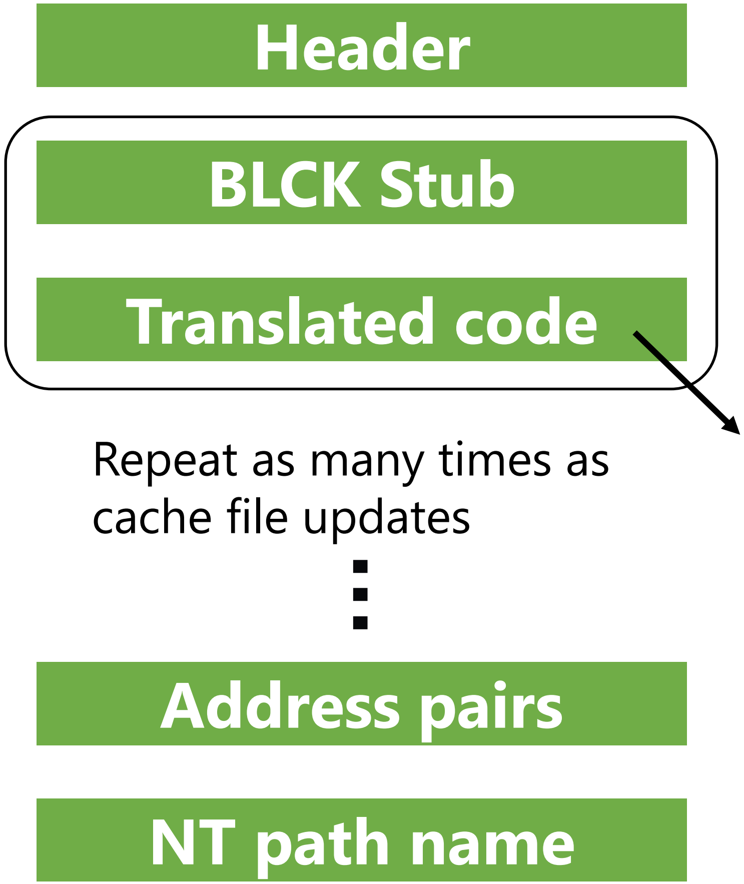
```
Advc]0 0% 255 USER32.DLL.B762FE91071D23DA8720F34E3667
- x86.00403db0:
0000b1c8       9dcf1fb8       str w29, [x28, -4]!
0000b1cc       fd031c2a       mov w29, w28
0000b1d0       9d4740b8       ldr w29, [x28], 4
0000b1d4       295d4311       add w9, w9, 0xd7, lsl 12
0000b1d8       29412f11       add w9, w9, 0xbd0
0000b1dc       23fbff97       bl 0x9e68
0000b1e0       20021fd6       br x17
```

ARM64 instructions in XTA cache file

https://github.com/FFRI/radare2

**We will only cover what is necessary to understand this research in this presentation**

# XTA Cache File Structure



Header

Contains Signature / Offset for following block etc.

BLCK Stub

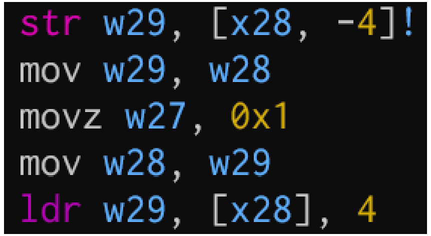
Includes code for returning to the emulator DLL

Translated code

Includes x86/x64 to ARM64 translated code

Repeat as many times as cache file updates

⋮



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

Not obfuscated , raw ARM 64 assembly

Address pairs

x86/x64 and ARM64 code RVA address pair

NT path name

NT path name for x86/x64 PE

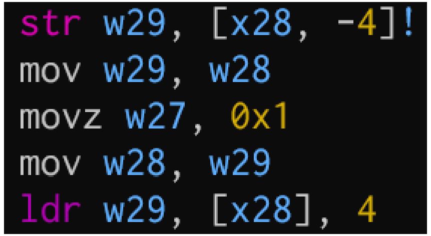
# XTA Cache File Structure

**Header** — et for following block etc.

What happens when we modify this ARM64 code ?

**BLCK Stub** — Includes code fo urnin the emulator DLL

**Translated code** — Includes x86/x64 to ARM nslation result code



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

**Not obfuscated , raw ARM 64 assembly**

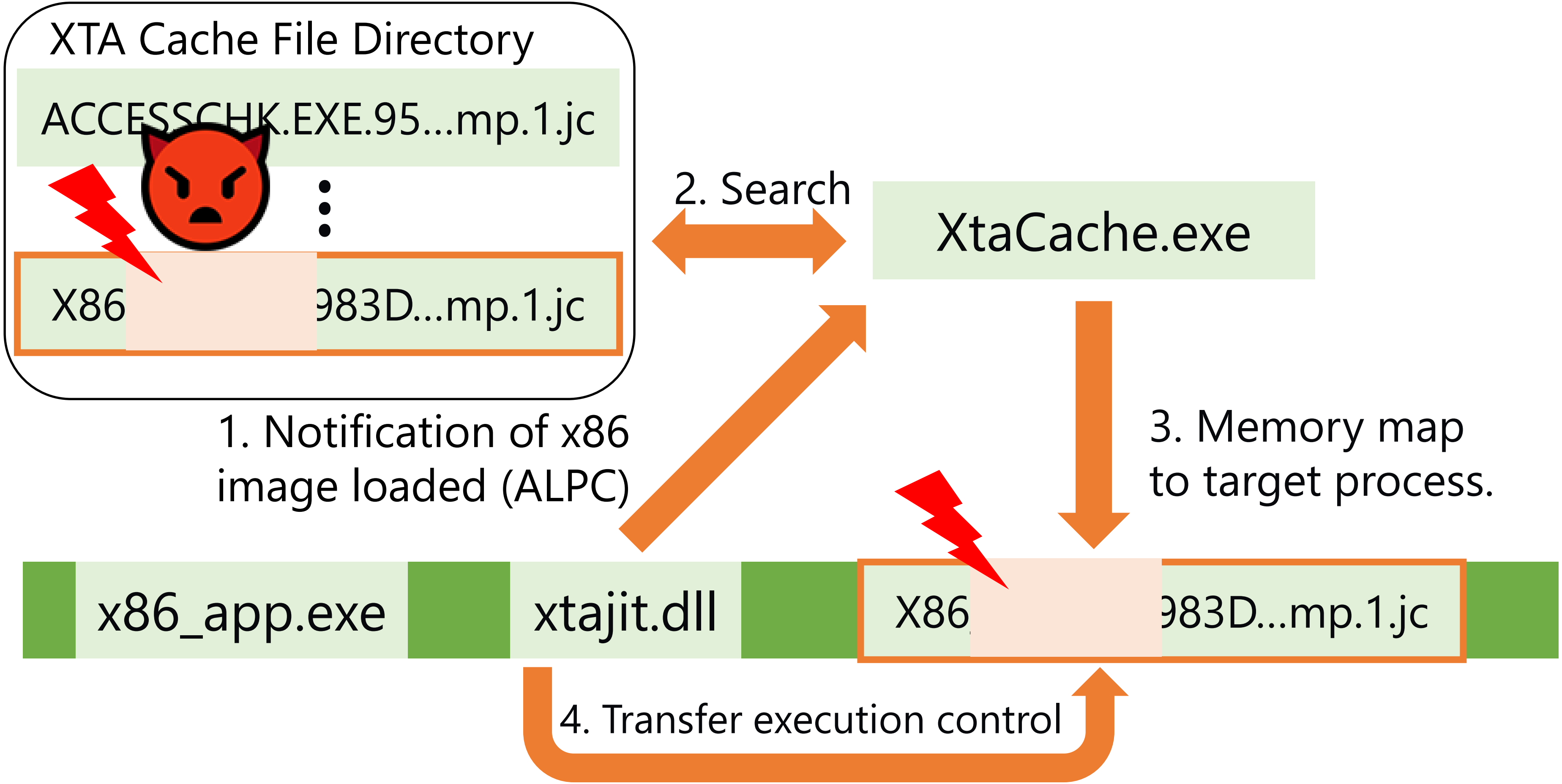
Repeat as many times as cache file updates

⋮

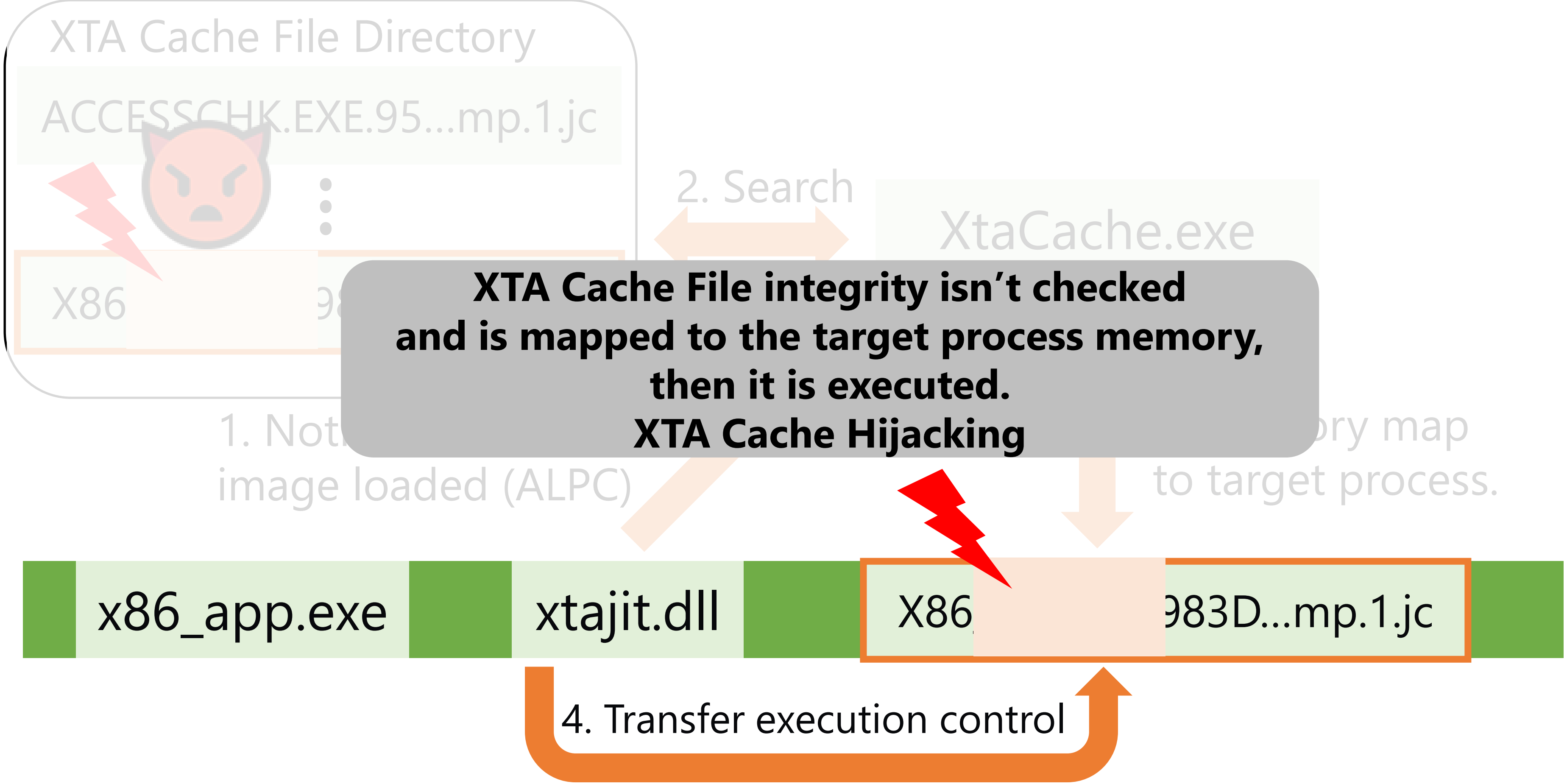
**Address pairs** — x86/x64 and ARM64 code RVA address pair

**NT path name** — NT path name for x86/x64 PE

# Execution flow when XTA cache file is modified



# Execution flow when XTA cache file is modified



**XTA Cache File integrity isn't checked and is mapped to the target process memory, then it is executed.**

**XTA Cache Hijacking**

# XTA Cache Hijacking Characteristics

Code injection method with 3 characteristics

- Difficult to detect: Can be done without getting the handle for the target process.
  - The modified XTA Cache File’s code gets executed through the normal emulation process.
- Difficult to trace: No evidence in the original x86/x64 PE file.
  - If we do not know the XTA Cache File, it is hard to trace.
- Persistent: The code injection result becomes persistent as a file.
  - The same code injection will get executed after a reboot if the same application is executed.

In the MITRE ATT&CK …

This Technique enables Defense Evasion・Credential Access・Persistence

However, **it does require administrator privilege to use…**

# XTA Cache Hijacking Characteristics

Code injection method with 3 characteristics

- Difficult to detect: Can be done without getting the handle for the target process.
  - The modified XTA Cache File’s code gets executed through the normal emulation process.
- Difficult to trace: No evidence of the original x86/x64 PE file.
  - If we do not know the XTA Cache File, it is hard to trace.
- Persistent: The code injection res
  - The same code injection will ge executed.

Is it worth to use this technique by gaining administrator privilege ?

🤔

In the MITRE ATT&CK …

This Technique enables Defense Evasion･Credential Access･Persistence

However, **it does require administrator privilege to use…**

# XTA Cache Hijacking Characteristics

Unique characteristics of this method

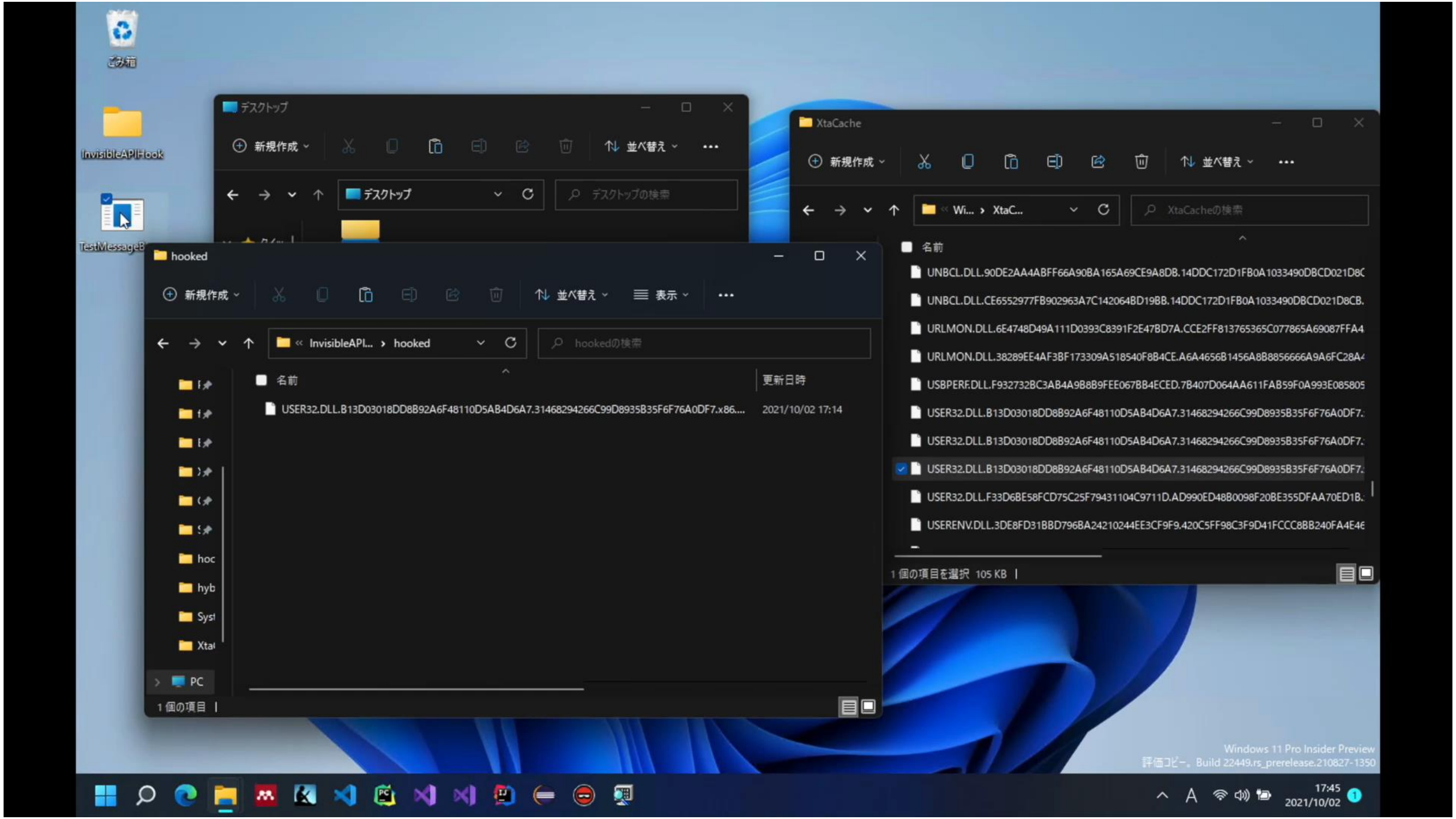
Invisible Execution

- Can execute a different code while hiding its execution at the x86/x64 code level.

Example of Invisible Execution: Invisible API hooking

- When code hooking, evidence of the hook are left.
- By using XTA Cache Hijacking, **we can code hook without leaving this evidence at the x86/x64 code level.**

# Demo



hooked:

| 名前 | 更新日時 |
|---|---|
| USER32.DLL.B13D03018DD8B92A6F48110D5AB4D6A7.31468294266C99D8935B35F6F76A0DF7.x86.... | 2021/10/02 17:14 |

1 個の項目

XtaCache:

```
名前
UNBCL.DLL.90DE2AA4ABFF66A90BA165A69CE9A8DB.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.3DE8FD31BBD796BA24210244EE3CF9F9.420C5FF98C3F9D41FCCC8BB240FA4E46
```

1 個の項目を選択 105 KB

# Countermeasures to XTA Cache Hijacking

Monitor access privilege changes of the XtaCache directory

To edit the XTA Cache File, we need to edit ACL of XtaCache directory.

- Because only the XtaCache service has access by default.

Usually changes to the access privileges to the XtaCache directory do not happen.

By adding this to the monitoring process, XTA Cache Hijacking can be traced.

- Defense is possible by limiting access privilege changes.

# In Summary

By analyzing XTAJIT and XTA Cache, revealed the following details

- x86/x64 Emulation flow
- XTA Cache File Structure (Only explained that the ARM64 code does not get obfuscated and exists in raw format)

Proposed a new code injection method, called XTA Cache Hijacking

- Hard to detect / Hard to trace / Persistent characteristics
- Also, a unique characteristic of this attack , Invisible Execution

# Compatibility technologies examined

- Binary Translation and Caching Mechanism (XTAJIT and XtaCache)
- Hybrid Binary (CHPE・ARM64EC)
- Fat Binary (ARM64X)

# Hybrid Binary

## Compiled Hybrid PE (CHPE)

PE contains both x86 and ARM64 code

Example of CHPE:

- System DLLs under %SystemRoot%¥SyChpe32 (such as kernel32.dll and user32.dll)
- Office EXE for Windows on ARM

X86 Emulation uses CHPE system DLLs

- When there is no SyChpe32, system DLLS under SysWOW64 are used instead.

# CHPE Characteristics

Enable near ARM64 native performance while maintaining x86 PE compatibility

- CHPE acts as a x86 PE
  - Surface analysis information is x86, disassembly result of the export function is also x86
  - export function is called export thunk
- export thunk is the jump stub to the ARM64 code
  - The actual function code is included into the ARM64 code at the jump destination
- JIT Binary translation only executes for export thunk
  - No need to JIT binary translation against the whole function
  - Therefore, performances is nearly equal to that of native ARM64 execution
    - – By CHPE-ing system DLL improve performance.

x86 (export thunk of func0)

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

ARM64 (actual code of func0)

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

# ARM64 Emulation Compatible (ARM64EC)

ARM64EC (※ABI・Build Architecture are collectively called like this)

Basically, a CHPE for x64

It has the following characteristics like CHPE:

- Includes both x64 and ARM64 code, x64 code is a jump stub to ARM64 code
- Allows for ARM64EC and x64 PE to be mixed and used in one process

The major difference with CHPE, the SDK is publicly available

- Third-party vendor can also use ARM64EC for ARM64 transition

June 28, 2021 | Windows Developers

Announcing ARM64EC: Building Native and Interoperable Apps for Windows 11 on ARM

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

# CHPE・ARM64EC API Calls

There are the two following cases to call an external DLL from CHPE/ARM64EC

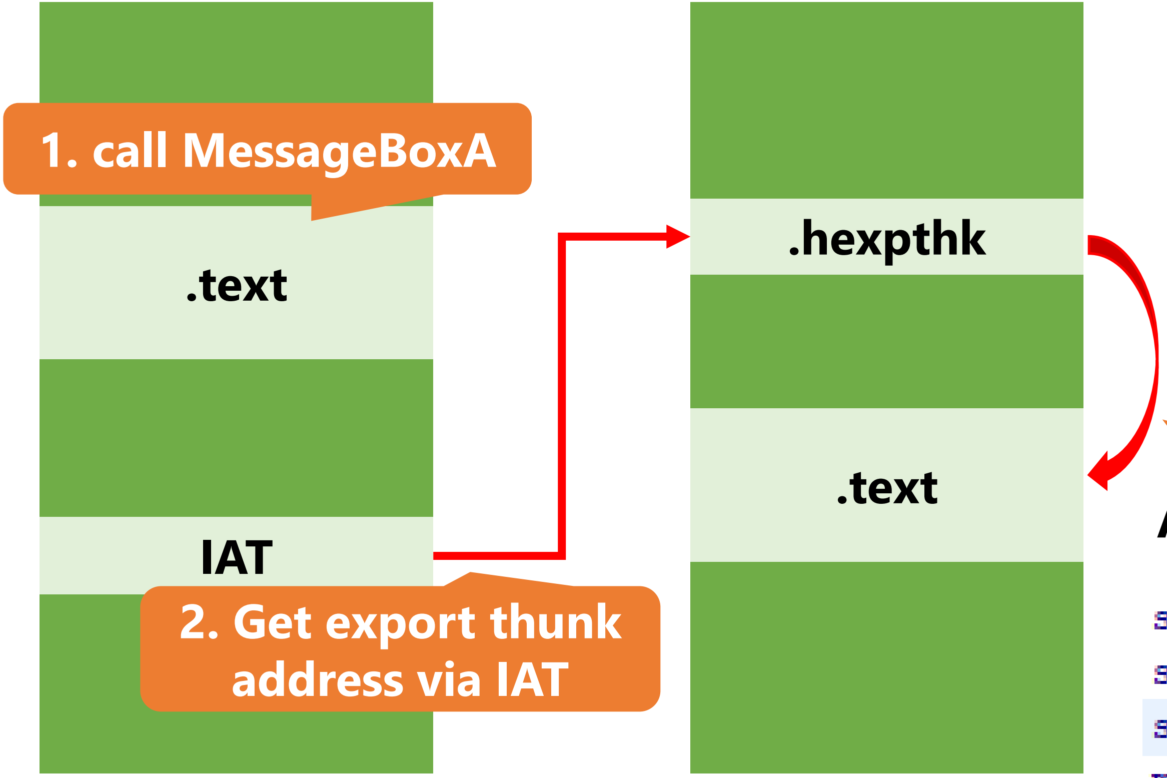
Function calls for x86/x64 DLLs (CHPE/ARM64EC -> x86/x64)

- It is executed by JIT Binary translation (or XTA cache File)
- Because of the difference in the conventions, calling convention changes happen

Function calls for CHPE/ARM64EC DLLs (CHPE/ARM64EC -> CHPE/ARM64EC)

- Since the two calling conventions are the same, it seems that there is no need for changes to the calling conventions, however…

# CHPE/ARM64EC API call flow



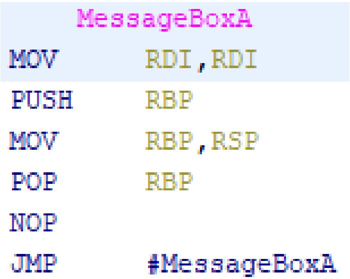
1. call MessageBoxA

2. Get export thunk address via IAT

App
(CHPE or ARM64EC)

user32.dll
(CHPE or ARM64EC)

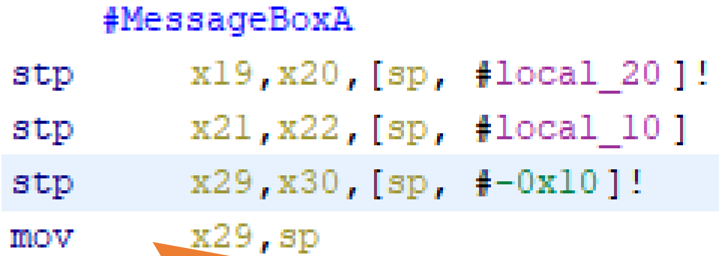
**x86/x64**



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

3. After JIT translation, if necessary, jump to the function body

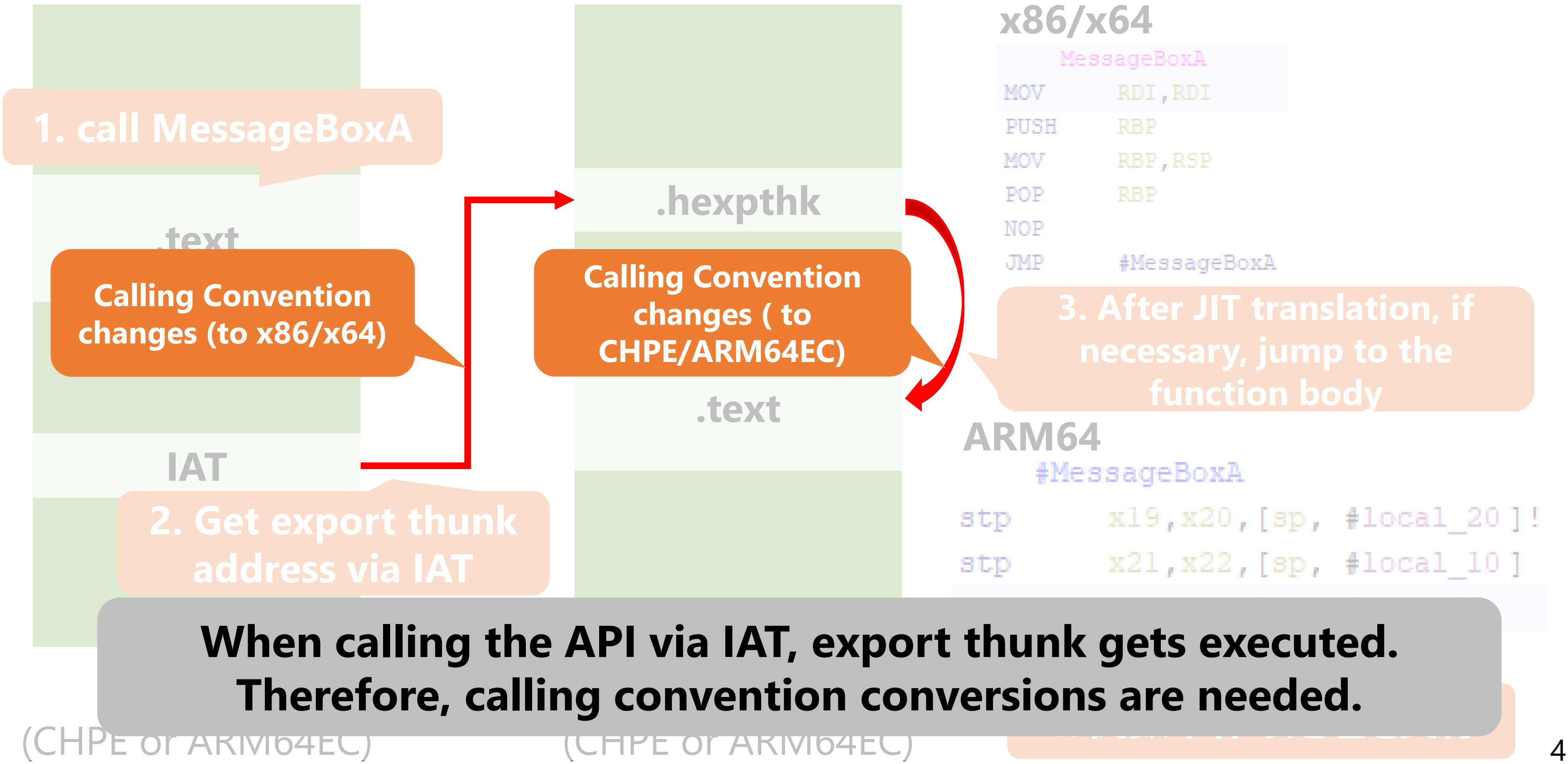
**ARM64**



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

4. Execute function body

# CHPE/ARM64EC API call flow


1. call MessageBoxA
.text
Calling Convention changes (to x86/x64)
IAT
2. Get export thunk address via IAT
(CHPE or ARM64EC)
.hexpthk
Calling Convention changes ( to CHPE/ARM64EC)
.text
(CHPE or ARM64EC)
3. After JIT translation, if necessary, jump to the function body
x86/x64
ARM64

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

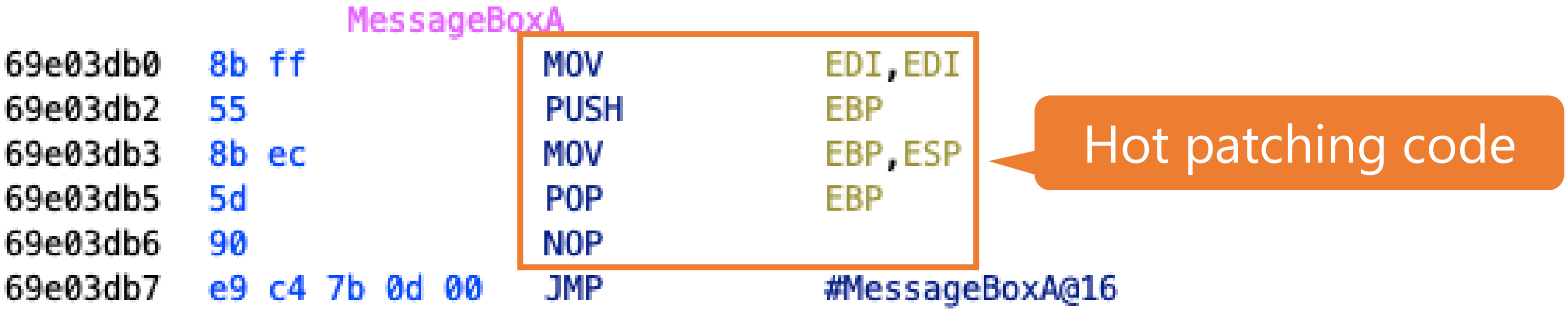
```
     #MessageBoxA
stp        x19,x20,[sp, #local_20]!
stp        x21,x22,[sp, #local_10]
```

**When calling the API via IAT, export thunk gets executed.**
**Therefore, calling convention conversions are needed.**

# Optimizing API calls

In many cases export thunk execution can be skipped.

Most of the code included in export thunk are for hot patching.



```
                         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
```

Hot patching code

Unless the code is changed by code hooking, this code does nothing.

If execution can be skipped except in special cases, it will lead to faster API calls.

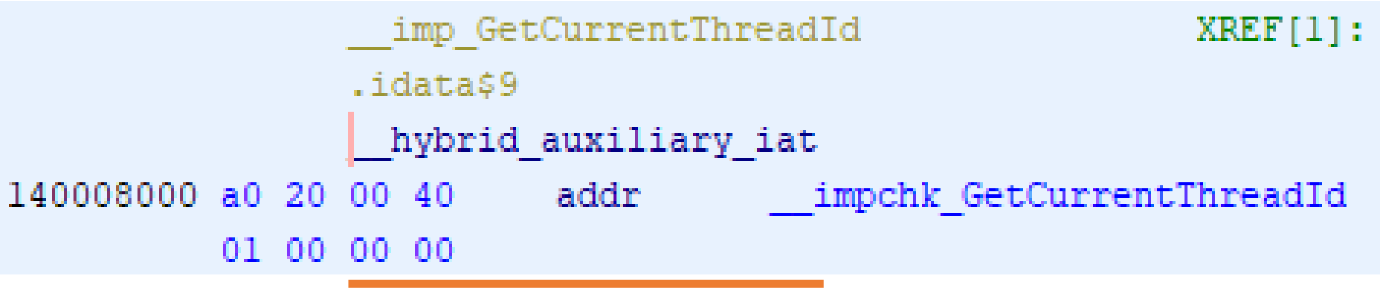
- It will allow the reducing of export thunk JIT translating and calling convention conversions.

**Can a function be called while skipping the execution of export thunk ?**

# Hybrid Auxiliary IAT

The other IAT that is in Hybrid Auxiliary IAT: CHPE・ARM64EC

When necessary, Hybrid Auxiliary IAT can skip the execution of export thunk



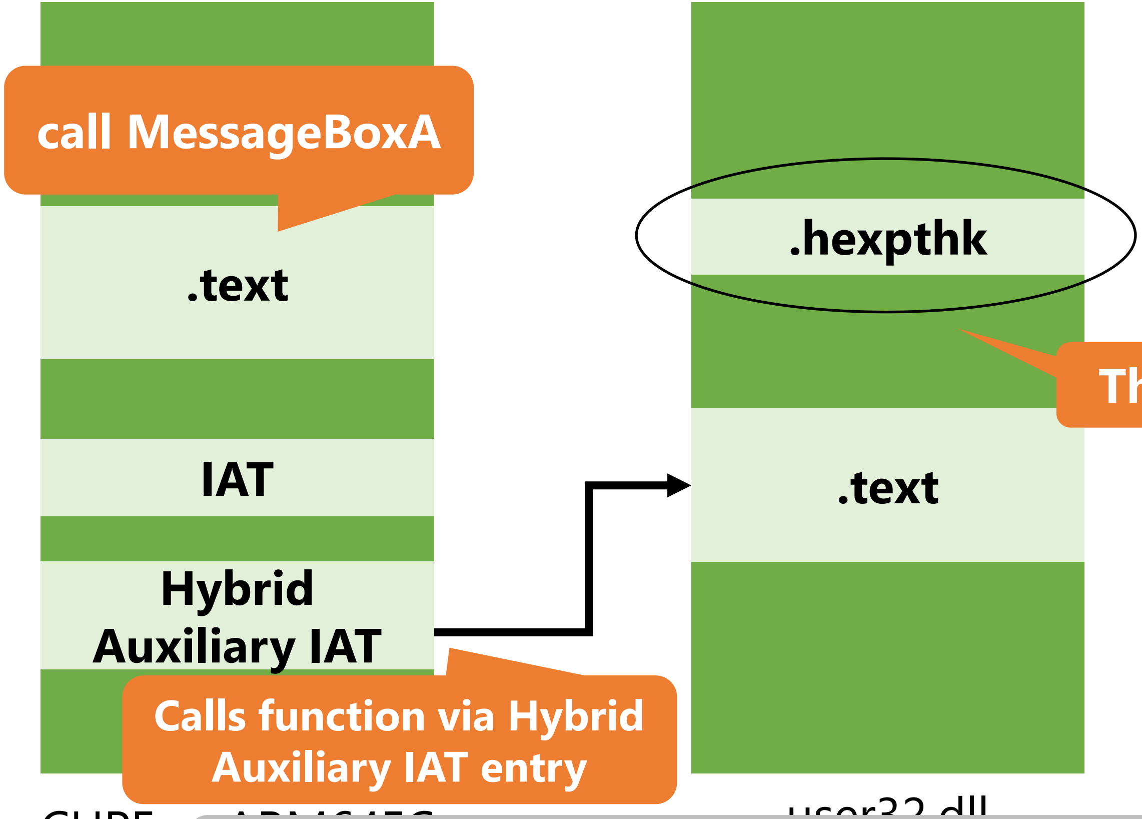
```
                        __imp_GetCurrentThreadId                       XREF[1]:
                        .idata$9
                        __hybrid_auxiliary_iat
140008000 a0 20 00 40       addr          __impchk_GetCurrentThreadId
          01 00 00 00
```

The program loader refers to the contents of the IAT and changes the Hybrid Auxiliary IAT at runtime. ※

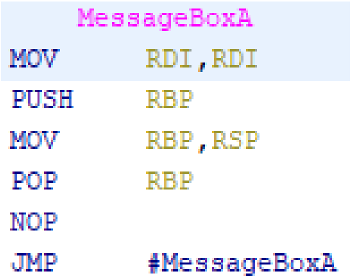
- When execution of export thunk is not needed, overwrites the Hybrid Auxiliary IAT entry by the actual address of the jump destination of export thunk
- This allows function calls without going through export thunk

※ntdll!#LdrpFastForwardAuxiliaryIat function does this

# API calls via Hybrid Auxiliary IAT



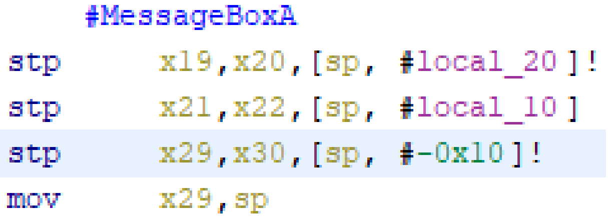
**x86/x64**



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

This part gets skipped

**ARM64**



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

**Can call the API while skipping the export thunk execution**

# Two types of hooking available in CHPE･ARM64EC

## Classic IAT hooking

It works fine.

- Because it detects protection changes of IAT and changes the API call to refer to IAT

## Hybrid Auxiliary IAT hooking

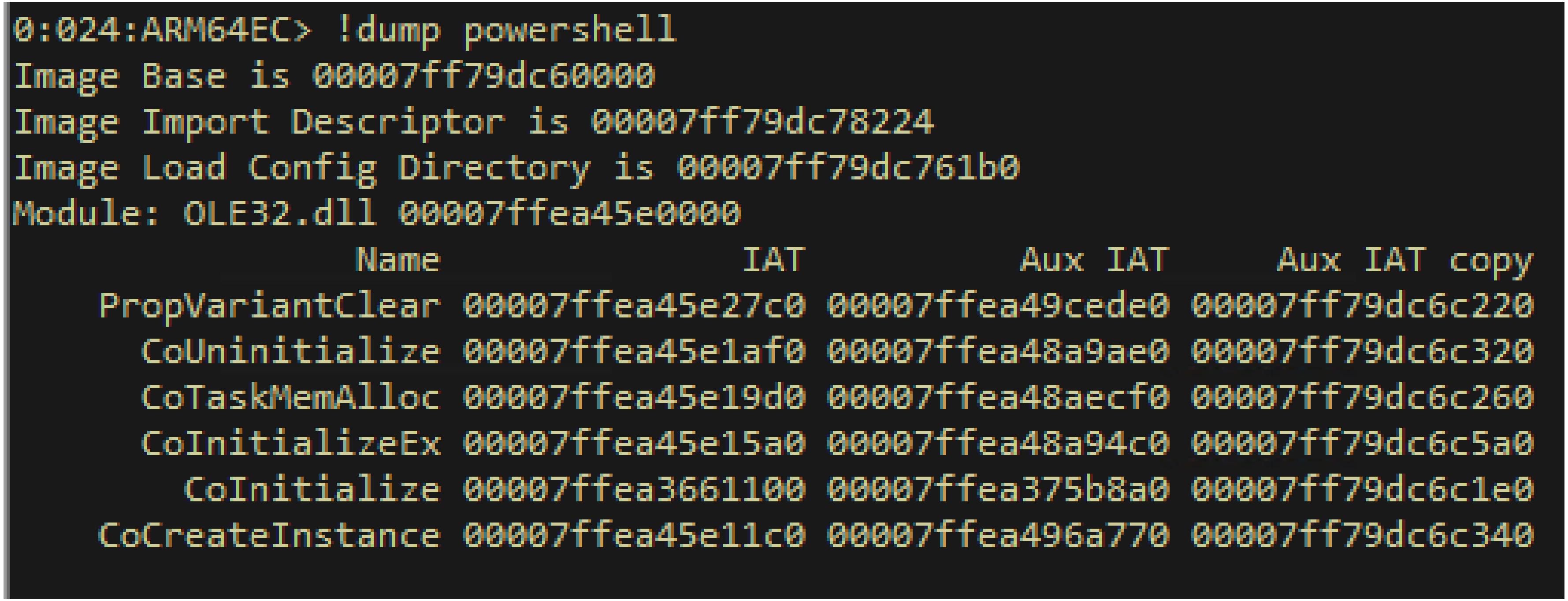
IAT hooking by modifying the Hybrid Auxiliary IAT entry

- Unique because it allows hooking without modifying IAT entries
  - Cannot detect if IAT is hooked by simply dumping IAT entries

# Countermeasures: Hybrid Auxiliary IAT hooking

WindDbg extension command for analyzing Hybrid Auxiliary IAT

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
```

# In Summary

Summarized CHPE・ARM64EC Characteristics and use cases.

Also pointed out the fact that there are many steps such as calling convention conversions, JIT translations when calling an API.

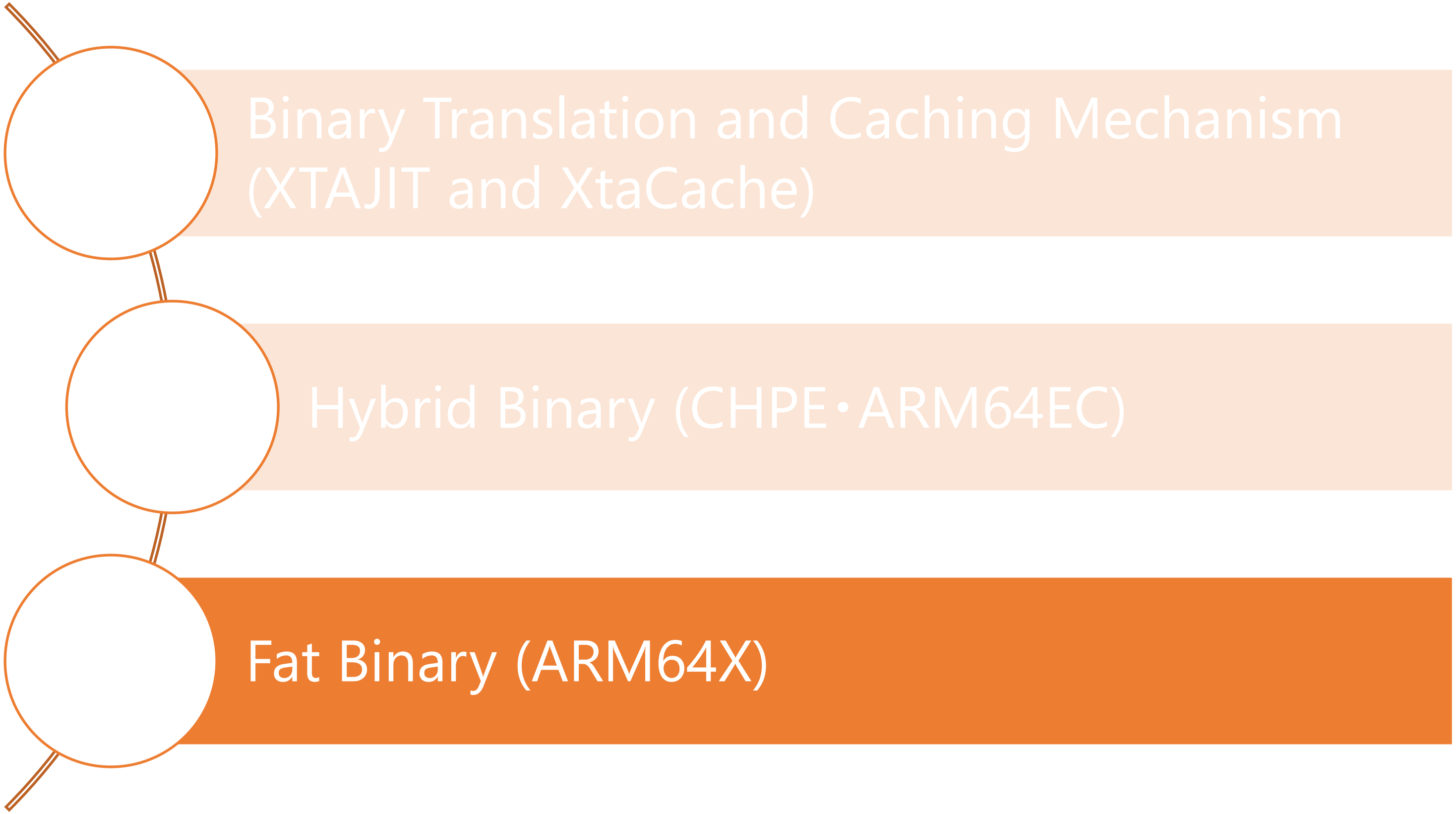
Clarified what Hybrid Auxiliary IAT is

Enables acceleration of API calls sometimes by skipping calling convention conversions / JIT binary translations.

Proposed a new API hooking method by modifying Hybrid Auxiliary IAT

Unique characteristic includes the inability to determine a hook based on IAT

# Compatibility technologies examined



Binary Translation and Caching Mechanism (XTAJIT and XtaCache)

Hybrid Binary (CHPE・ARM64EC)

Fat Binary (ARM64X)

# Fat binary for Windows on ARM

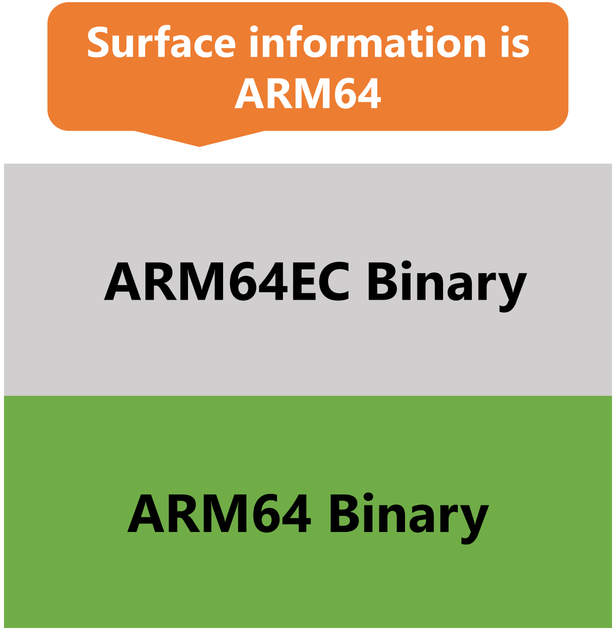
ARM64X

Contains code for both ARM64 native and ARM64EC

- File format is the same as the traditional PE
  - No new file format is prepared for ARM64X
- Surface information is ARM64

Examples of files provided as ARM64X

- System DLLs under %SystemRoot%System32
- Some of the System EXE such as cmd.exe or dllhost.exe



Surface information is ARM64

# ARM64X Characteristics

In case of EXE, the code that gets executed changes based on parent process architecture

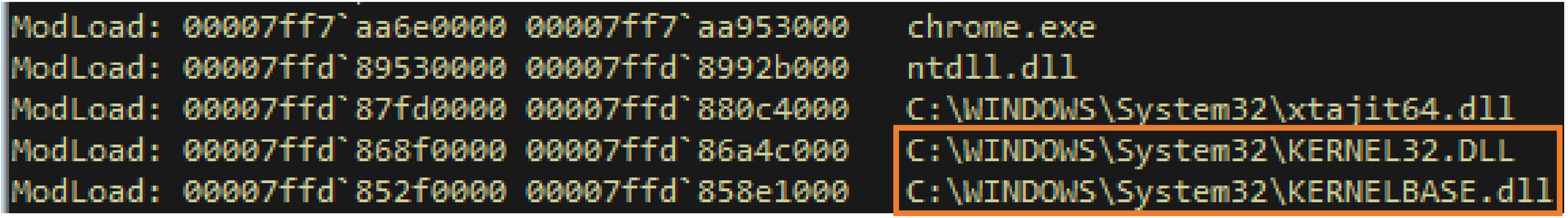
| Parent process architecture | Executed code |
|---|---|
| x86 | ARM64 |
| x64 | ARM64EC |
| ARM64 | ARM64 |
| ARM64EC | ARM64EC |

# ARM64X Characteristics

## Can be loaded from all processes for ARM64EC・x64・ARM64

Surface information is ARM64, but can be loaded from both ARM64EC and x64 processes

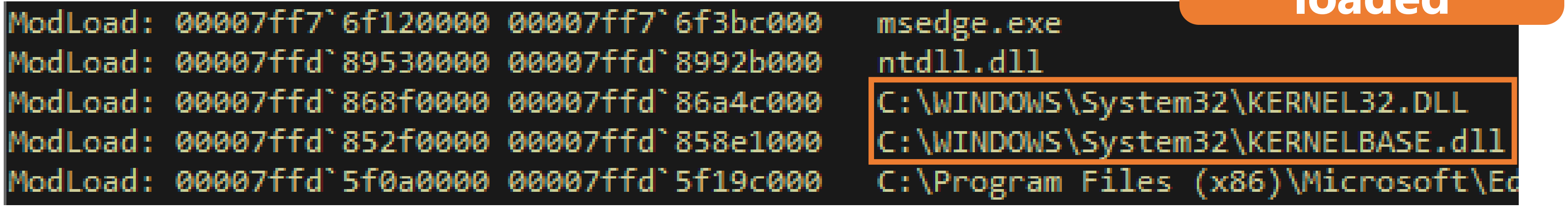
- When loaded from x64 process (x64 Chrome browser)



```
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
```

- When ARM64 native process does the loading (ARM64 Edge browser)

**Same DLL gets loaded**



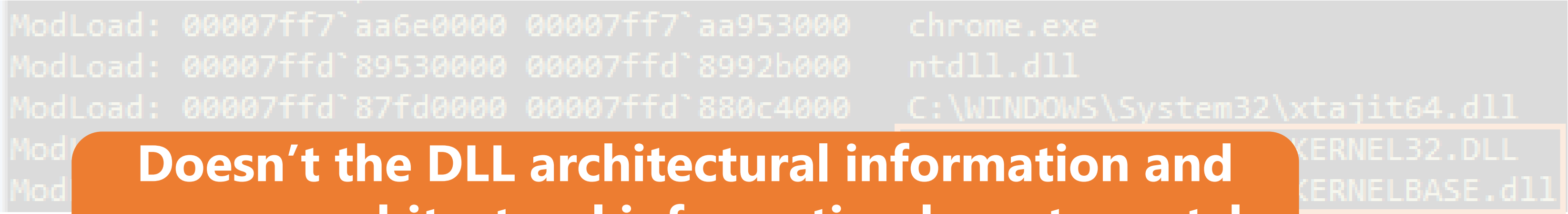
```
ModLoad: 00007ff7`6f120000 00007ff7`6f3bc000   msedge.exe
ModLoad: 00007ffd`89530000 00007ffd`8992b000   ntdll.dll
ModLoad: 00007ffd`868f0000 00007ffd`86a4c000   C:\WINDOWS\System32\KERNEL32.DLL
ModLoad: 00007ffd`852f0000 00007ffd`858e1000   C:\WINDOWS\System32\KERNELBASE.dll
ModLoad: 00007ffd`5f0a0000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

# ARM64X Characteristics

Can be loaded from all processes for ARM64EC･x64･ARM64

Surface information is ARM64, but can be loaded from both ARM64EC and x64 processes

- When loaded from x64 process (x64 Chrome browser)

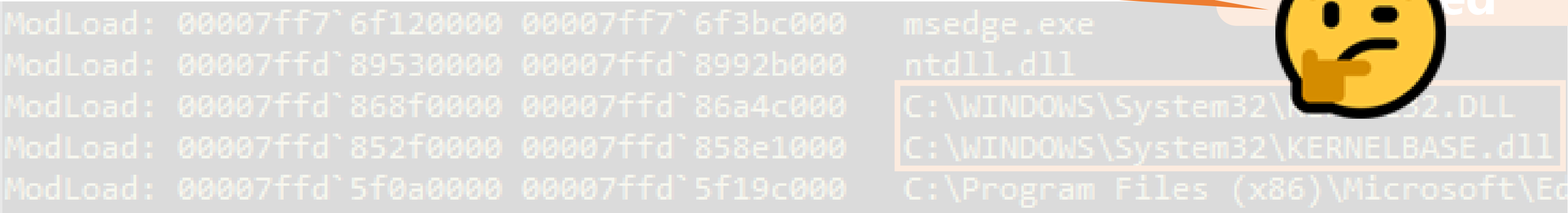


```
ModLoad: 00007ff7`aa6e0000 00007ff7`aa953000   chrome.exe
ModLoad: 00007ffd`89530000 00007ffd`8992b000   ntdll.dll
ModLoad: 00007ffd`87fd0000 00007ffd`880c4000   C:\WINDOWS\System32\xtajit64.dll
Mod                                                              KERNEL32.DLL
Mod                                                              KERNELBASE.dll
```

**Doesn’t the DLL architectural information and process architectural information have to match to load usually ?**

- Whe
(ARM

me DLL gets
ed

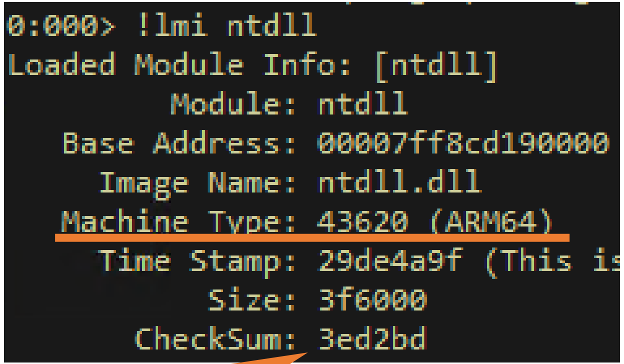


```
ModLoad: 00007ff7`6f120000 00007ff7`6f3bc000   msedge.exe
ModLoad: 00007ffd`89530000 00007ffd`8992b000   ntdll.dll
ModLoad: 00007ffd`868f0000 00007ffd`86a4c000   C:\WINDOWS\System32\          32.DLL
ModLoad: 00007ffd`852f0000 00007ffd`858e1000   C:\WINDOWS\System32\KERNELBASE.dll
ModLoad: 00007ffd`5f0a0000 00007ffd`5f19c000   C:\Program Files (x86)\Microsoft\Ed
```

# ARM64X Characteristics

Analyze surface information of loaded DLL with WinDbg

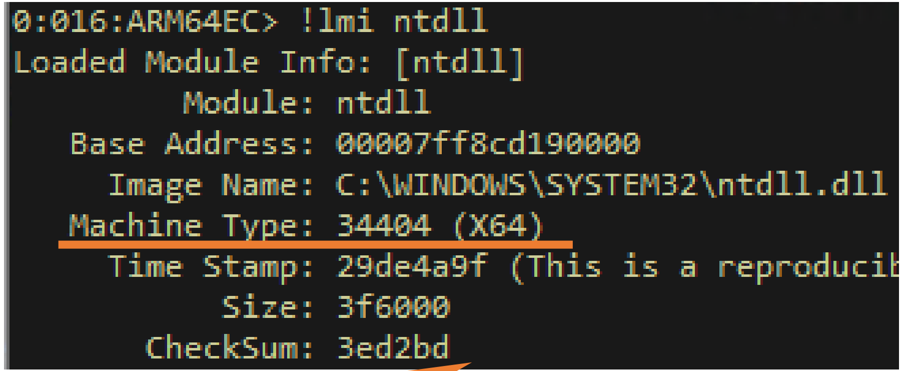
ARM64 Process



```
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
```

Machine Type for ntdll.dll is ARM64

ARM64EC (or x64) Process



```
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
```

**Machine Type for ntdll.dll is x64 ?**

**Should be the same file but surface information changed after the DLL was loaded ?**
**Only when it is used by ARM64EC (or x64) processes, does the surface information change ?**

# New relocation entries included in ARM64X

Newly added relocation entries to switch between ARM64 and ARM64EC

IMAGE_DYNAMIC_RELOCATION_ARM64X

- Added as one entry to Dynamic Value Relocation Table (DVRT)
- Noted as DVRT ARM64X after this

Applied prior to mapping the target process to memory, overwrites various information dynamically

- PE Header (Entry point, RVA of Export Directory, Machine Type etc.)
- Offset for API call included in the code section

This relocation is applied from the kernel side by nt!MiApplyConditionalFixups

# DVRT ARM64X Data Structure

For details of DVRT ARM64X's data structure refer to

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

3 types of relocation entries

- Zero fill: 0 clears 2/4/8 byte of specified address
- Assign value: overwrites 2/4/8 byte of specified address with specified value
- Delta: either add/subtract 4 or 8 from the 4 bytes of specified address data

# DVRT ARM64X Data Structure

For details of DVRT ARM64X's data structure refer to

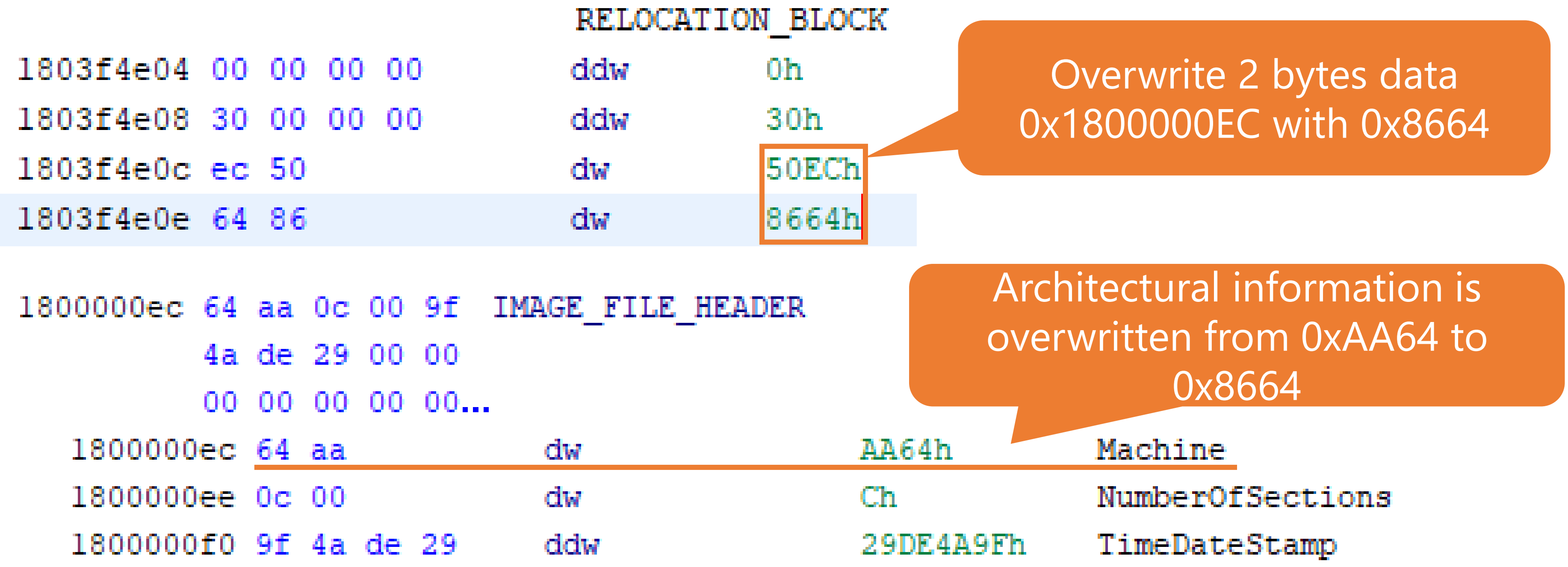
[Discovering a new relocation entry of ARM64X in recent Windows 10 on Arm]

Unlike base relocations, it can arbitrary-write the data in the image

3 types of relocation entries

- Zero fill: 0 clears 2/4/8 byte of sp
- **Assign value: overwrites 2/4/8 byte of specified address with specified value**
- Delta: either add/subtract 4 or 8 from the 4 bytes of specified address data

# DVRT ARM64X Example:
# Rewrite of architectural information



```
                          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
```

Overwrite 2 bytes data 0x1800000EC with 0x8664

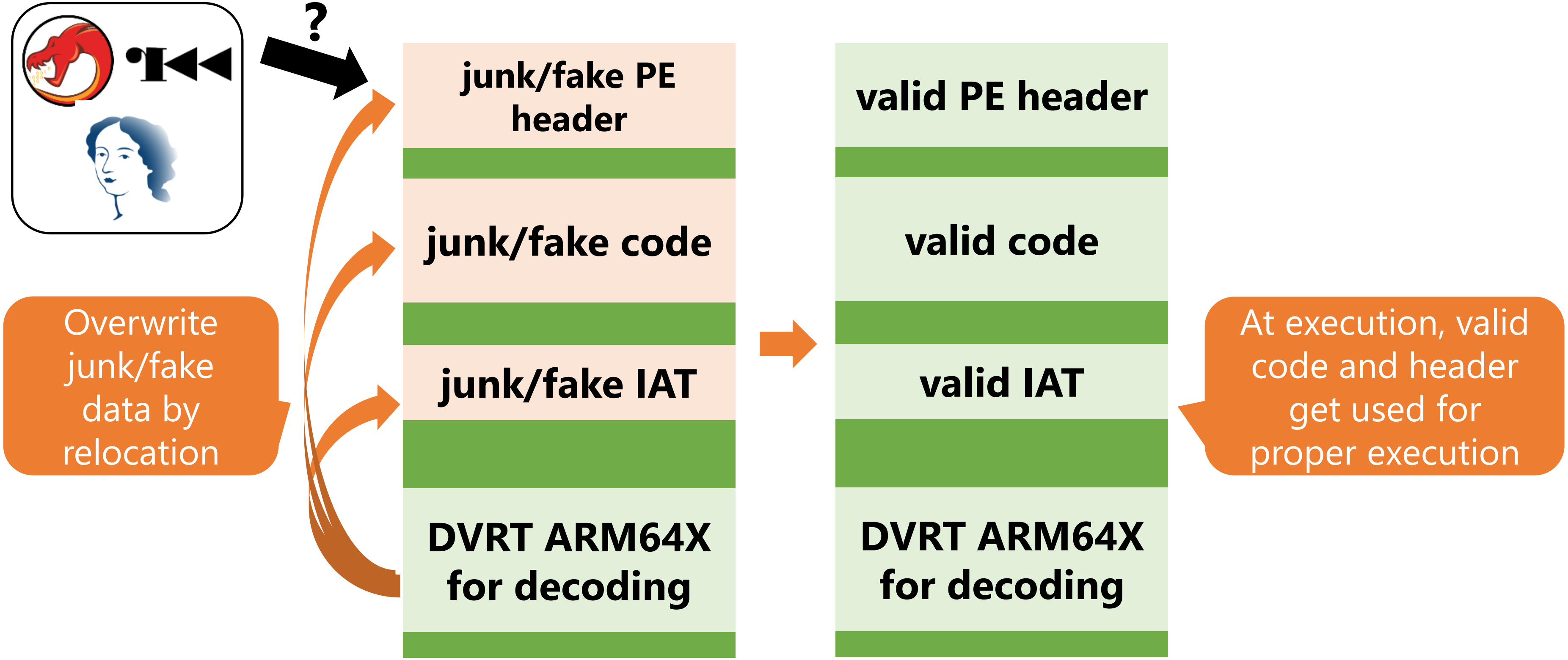
```
1800000ec 64 aa 0c 00 9f  IMAGE_FILE_HEADER
          4a de 29 00 00
          00 00 00 00 00...
   1800000ec 64 aa        dw          AA64h        Machine
   1800000ee 0c 00        dw          Ch           NumberOfSections
   1800000f0 9f 4a de 29  ddw         29DE4A9Fh    TimeDateStamp
```

Architectural information is overwritten from 0xAA64 to 0x8664

**Architectural information inside the DLL changes from ARM64 to x64 by DVRT ARM64X**
**This enables the loading from x64 / ARM64EC processes**

# ARM64X Relocation Obfuscation

Can DVRT ARM64X be used for obfuscation?



?

junk/fake PE header

junk/fake code

junk/fake IAT

DVRT ARM64X for decoding

valid PE header

valid code

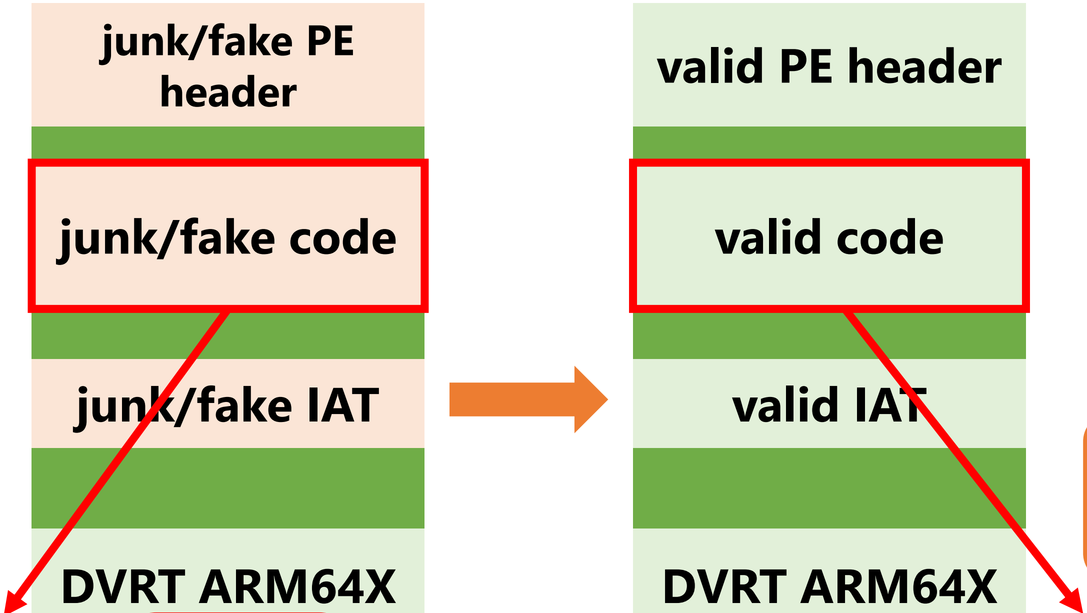
valid IAT

DVRT ARM64X for decoding

Overwrite junk/fake data by relocation

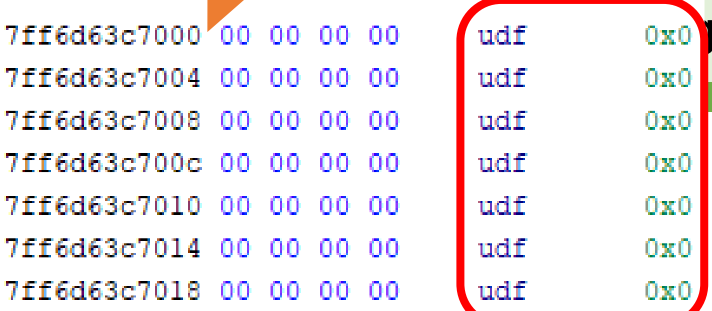
At execution, valid code and header get used for proper execution

# Usage example: Packer

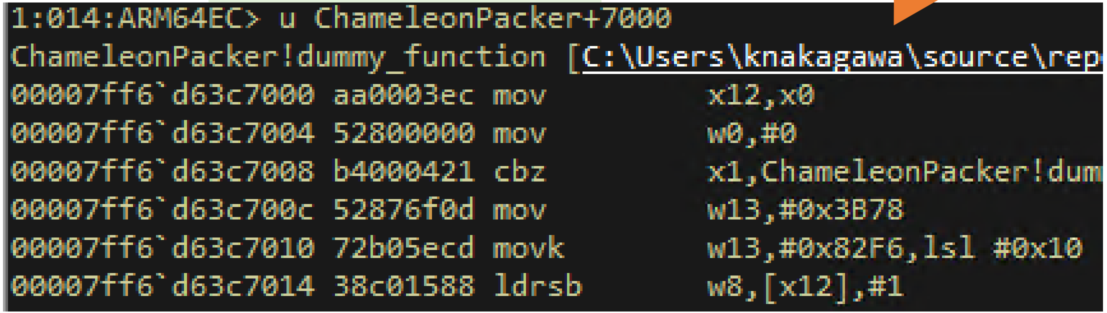


Code before applying relocation

Unpacked code after applying relocation



```
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
```



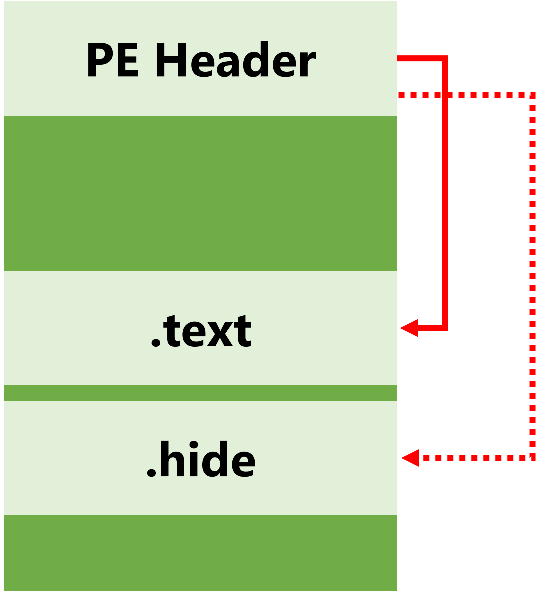
```
1:014:ARM64EC> u ChameleonPacker+7000
ChameleonPacker!dummy_function [C:\Users\knakagawa\source\rep
00007ff6`d63c7000 aa0003ec mov       x12,x0
00007ff6`d63c7004 52800000 mov       w0,#0
00007ff6`d63c7008 b4000421 cbz       x1,ChameleonPacker!dum
00007ff6`d63c700c 52876f0d mov       w13,#0x3B78
00007ff6`d63c7010 72b05ecd movk      w13,#0x82F6,lsl #0x10
00007ff6`d63c7014 38c01588 ldrsb     w8,[x12],#1
```

# Usage example: Packer

Q. If we dump it, we can easily analyze?

A. Possible to obstruct analysis after memory dump

By editing the PE section header with DVRT ARM64X fool the disassembly results.



Location for code used after unpacking

NULL the RVA of the Image Section Header with DVRT ARM64X at execution time

Disassembly result from a memory dump does not include .hide

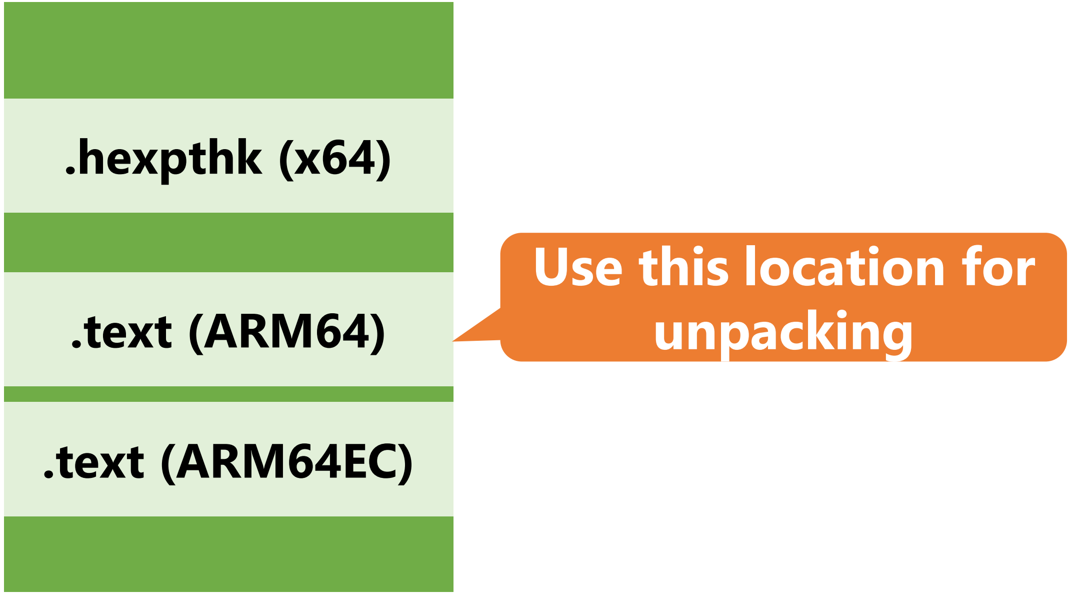
# Usage example: Packer

Further obstruction of analysis

ARM64X contains both ARM64 and ARM64EC code

- When executing ARM64EC process, ARM64 native code does not get executed

If you unpack code to the location where ARM64 native code is contained….



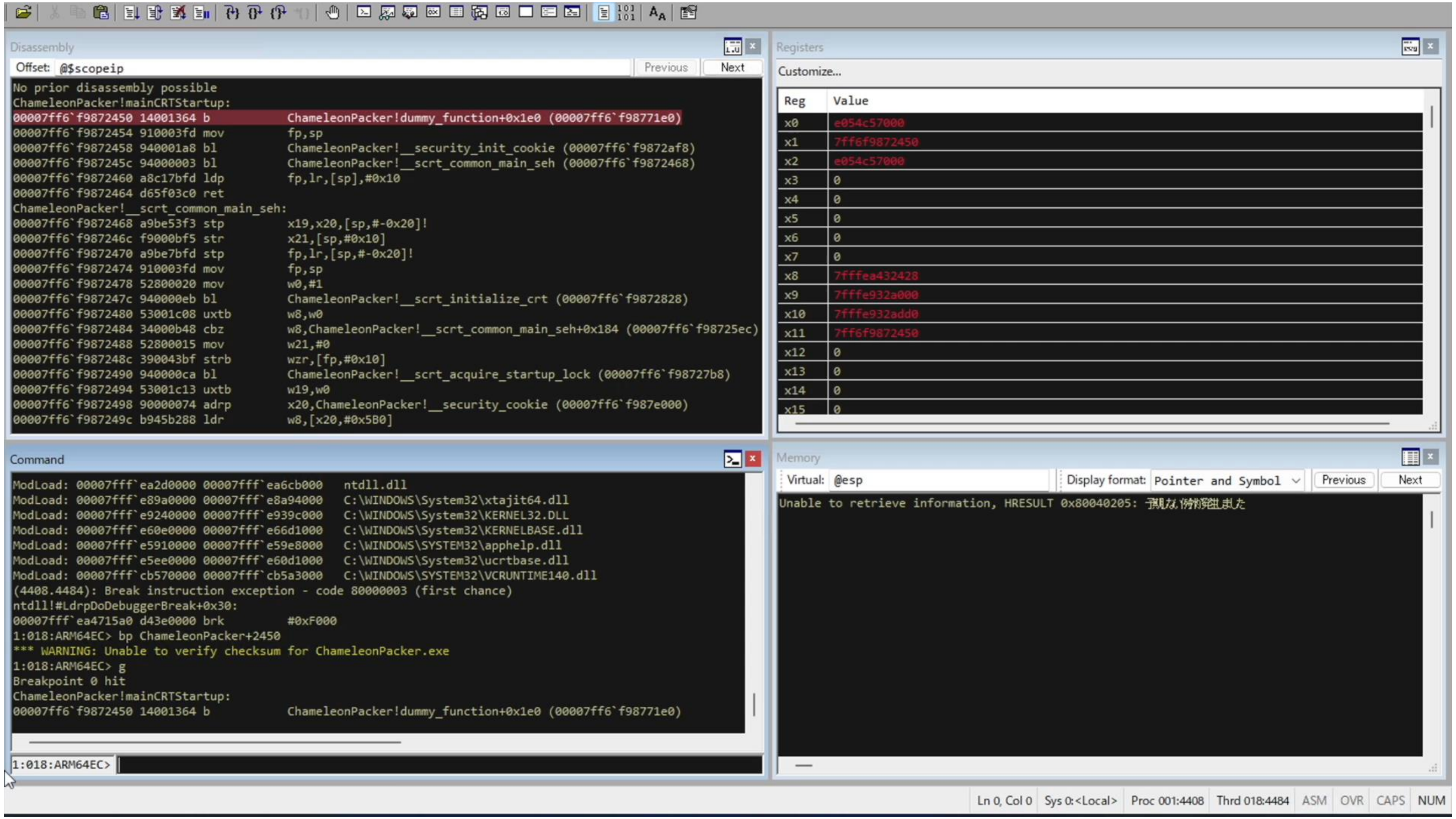
.hexpthk (x64)

.text (ARM64)

.text (ARM64EC)

Use this location for unpacking

# Demo



Disassembly

Offset: @$scopeip

```
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

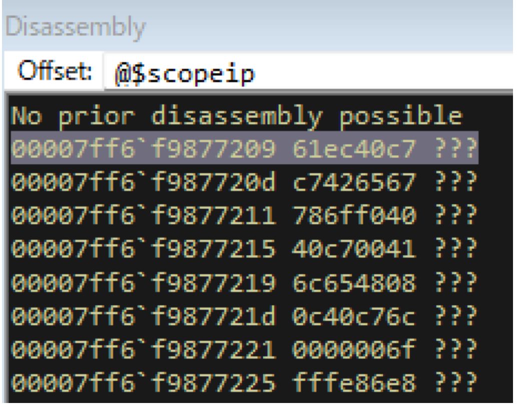
Display format: Pointer and Symbol

```
Unable to retrieve information, HRESULT 0x80040205: 予期しない例外が発生しました
```

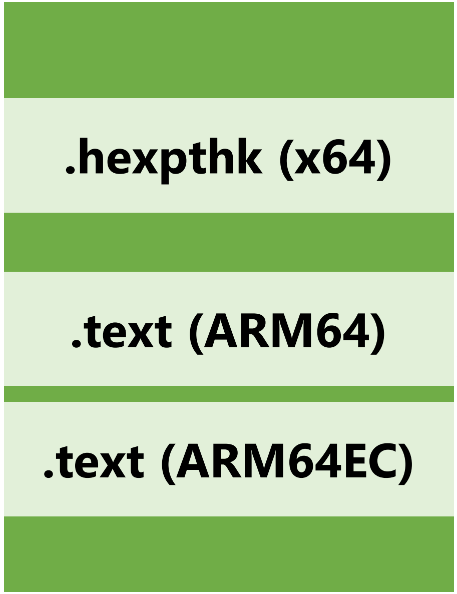
# Usage example: Packer

Obstruct dynamic analysis using WinDbg

This is ARM64



```
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 ???
```



Interpreted and executed as x64 in ARM64EC process

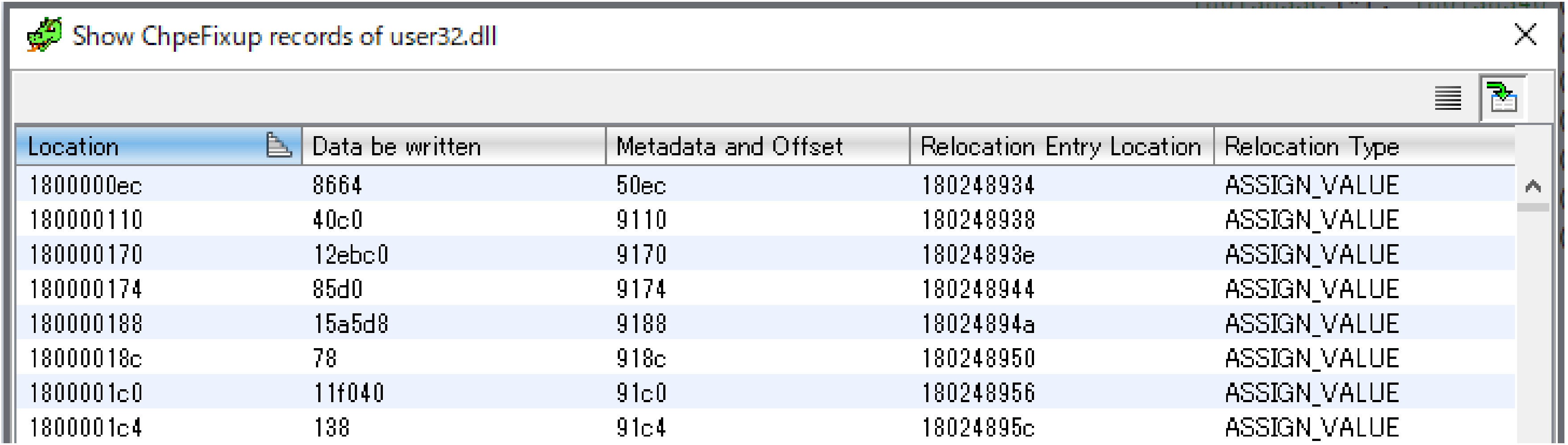
# Countermeasures: Ghidra script for ARM64X analysis

## Ghidra script for DVRT ARM64X analysis

Apply DVRT ARM64X relocation to ARM64X and save as a file

- If DVRT ARM64X is used as a packer, possible to save the unpacked result as a file

Also, can dump DVRT ARM64X relocation entries as follows



Show ChpeFixup records of user32.dll

| 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 |

# In Summary

## Summarized what is ARM64X and its characteristics

ARM64X is a fat binary containing both ARM64 native and ARM64EC code

- Characterized by code execution changes based on process used or parent process.

## Disclosed new relocation entry implemented in ARM64X

Called IMAGE_DYNAMIC_RELOCATION_ARM64X (DVRT ARM64X)

- Unique characteristics include relocation entries that can Arbitrary Written inside the ARM64X image.

## Proposed new obfuscation technique using DVRT ARM64X

Pointed out that technique is resistant to static analysis or dynamic analysis using WinDbg after dump.

# Summary and message

Presented analysis results of compatibility technologies for Windows on ARM

Introduced attack methods using these compatibility technologies.

It has just been announced, and there may be various other attack methods.

- Further research is necessary

The concepts of the attack methods presented today is widely applicable.

- For example, the Caching Mechanism is implemented for speed when implementing binary translation.
  - We believe that same concept will be applicable to the similar compatibility technologies in the future.

**We hope security research regarding compatibility technologies will become more active and as a result sufficient countermeasures will be developed in the future.**

# Links to tools and PoC

## XTA Cache File Related

https://github.com/FFRI/XtaTools (PoC code for XTA Cache Hijacking)

https://github.com/FFRI/radare2 (radare2 for XTA Cache File analysis)

Black Hat EU 2020 Presentation (Details regarding file formats and XTA Cache Hijacking)

## ARM64EC・ARM64X Related

https://github.com/FFRI/ProjectChameleon (PoC Code/tool/analysis document)

https://ffri.github.io/ProjectChameleon/ (Document aggregating analysis results.)

# Thank you!

For questions/comments:

Twitter DM: @FFRI_Research

email: research-feedback@ffri.jp