脆弱性緩和技術の最前線

CPU レベルの技術

FFRI Security, Inc.
株式会社ＦＦＲＩセキュリティ
リサーチ・エンジニア 中川 恒

イントロダクション

脆弱性緩和技術と脆弱性攻略技術は、イタチごっこを繰り返しながら互いに発展を続けてきた。これまで、ASLR、W ^X、stack canary など様々な緩和技術が導入されてきた [1]。しかし、例えば W ^X については Return-Oriented-Programming (以下、ROP) などのいわゆる code-reuse attacks によりバイパス可能であることが示され [2]、その対策として Control Flow Integrity (以下、CFI) を担保する技術が登場している [3]。すなわち、新しい緩和技術の導入が新しい脆弱性攻略技術を生み、さらに脆弱性緩和技術が強化されるという歴史をたどっている。

その歴史の中で、近年 CPU レベルでの脆弱性緩和技術が登場している。例えば、Tiger Lake 以降の Intel CPU には shadow stack による return address の改竄検知を行う技術が導入されている [4]。Arm においても Armv8 では脆弱性緩和のための命令拡張が多く導入されており、v8.3 ではポインタに認証用のコードを挿入し、return address や関数ポインタの改竄を検知する Arm PAC が導入されている。v8.5 においては間接分岐命令の分岐先を制限する Arm BTI などが導入されている [5]。Arm PAC については iOS において導入されており [6]、return address や関数ポインタの改竄検知に利用されている。

本文書はこのような近年導入が進みつつある CPU レベルでの脆弱性緩和技術のうち、Intel や Arm の CPU に実装されている、あるいは今後実装される予定のものに絞って技術的詳細を説明し、こうした技術の導入により具体的にどういった脆弱性攻撃に対して有効な対策となるのかを紹介する。また、近年の OS とコンパイラにおけるサポート状況についても説明する。特に Windows については、 Insider Preview Builds の Windows の独自のリバースエンジニアリング結果から得られた最新の実装状況について説明する。

Table of Contentes

1. 用語の整理

本章では本文書を理解する上で必要な用語を整理する。

1.1 Control flow hijacking

本文書では control flow をプログラムを構成する basic block の実行順序と定義する。ここで、basic block とは内部に分岐を持たず、最初の命令が実行の開始点、末尾の命令が実行の終了点となっている命令列のことを指す。以下に x64 アセンブリでの例をあげる (ソースコード 1.1 を参照)。

```
; basic block 0
func:
        cmp edi, esi
        jg tag1

; basic block 1
        mov eax, edi
        sub eax, esi
        ret

; basic block 2
tag1:
        mov eax, edi
        add eax, esi
        ret
```

ソースコード 1.1　3 つの basic block より構成される x64 アセンブリコード

コメントに示したとおり、上記アセンブリは basic block 0、1、2 の 3 つにより構成されている。この 3 つの basic block は末尾に分岐命令 (ret や jg など) を持ち、末尾以外には分岐命令を持たないことがわかる。

プログラムに脆弱性が存在する場合、攻撃者により return address や間接分岐命令の分岐先が書き換えられ、プログラム作成者の意図しない control flow でコードが実行される。これを control flow hijacking と呼ぶ。control flow hijacking は以下の 2 つに大別される。

- Forward-edge の control flow hijacking: 関数ポインタ・仮想関数テーブルの書き換え等により、間接分岐命令の実行時に control flow を攻撃者の意図したものに変更
- Backward-edge の control flow hijacking: return address の書き換えにより、関数 return 時に control flow を攻撃者の意図したものに変更

1.2 Spatial/Temporal memory safety

Spatial memory safety とはメモリ参照がポインタで指し示されるオブジェクトや配列の領域内にある性質のことを指す。ソースコード 1.2 に spatial memory safety が破られているプログラムの例を示す。

```
void func() {
        // off-by-one error

        char buffer[10];
        for (int i = 0; i <= 10; i++) {
                buffer[i] = i; // spatial memory
                     safety is violated when i
                    == 10.
        }

        // ...
}
```

ソースコード 1.2　spatial memory safety を破る例

ソースコード 1.2 は、stack に確保した buffer が char 型の長さ 10 の配列になっているのに対し、配列の境界外である 11 番目の要素に値を書き込んでいる [*1]。そのため、このコードは spatial memory safety を満たさない。

Temporal memory safety とはオブジェクトへのポインタを介した全てのメモリ参照が、そのオブジェクトが有効なタイミングで行われている性質のことを指す。ここで有効とは解放済みではない領域のことを指す。ソースコード 1.3 に、temporal memory safety が破られている例を示す。

```
void func() {
        // ...

        char* buffer = (char*)malloc(10);

        // do something ...

        free(buffer); // freeing buffer
```

[*1] C 言語において配列のインデックスは 0 始まりであるため、buffer[10] で 11 番目の要素にアクセスすることとなる

```
        for (int i = 0; i < 10; i++) {
                buffer[i] = i; // <--- access
                    although buffer was freed.
        }
        // ...
}
```

ソースコード 1.3　temporal memory safety を破る例

ソースコード 1.3 では malloc で確保された buffer が解放されているにも関わらず、矢印の箇所でアクセスされている。そのため、temporal memory safety が満たされていない。

1.3　Code pointer integrity

Code pointer integrity とは return address や関数ポインタなどプログラムに使われる code pointer の完全性が保たれている状態のことを指す [7]。例えば stack ベースのバッファオーバーフローにより、return address の書き換えが起きる場合は、code pointer integrity が満たされない。

Code pointer integrity の violation への緩和技術として、関数ポインタの暗号化がこれまでに提案されている。関数ポインタの暗号化は C 言語の標準ライブラリとして入ることはなかったものの、MSVC では現在でも EncodePointer 関数と DecodePointer 関数として専用の API が用意されている。以下に示すソースコード 1.4 は参考文献 [8] のサンプルコードにコメントをつけたものである。

```
#include <Windows.h>

// printf への関数ポインタを暗号化し、log_fn に
    代入
void *log_fn = EncodePointer(printf);

/* ... */

// 復号し、fn に代入
int (*fn)(const char*, ...) = (int(*)(const char
    *, ...)) DecodePointer(log_fn);

fn("Hello\n");
```

ソースコード 1.4　関数ポインタの暗号化を行う API の利用例

暗号化に用いたキーの推定が困難という条件付きではあるが、こうした関数ポインタの暗号化により攻撃者による関数ポインタの改竄は困難になる。

一方、間接関数呼び出し毎に関数ポインタの復号が必要となるため、頻繁に呼び出す関数に対して適用する場合、パフォーマンスへの影響に注意する必要がある。また、復号なしに元の関数ポインタの値を特定できないため、デバッグが困難になる点にも注意が必要である。

2. CPU レベルの脆弱性緩和技術

本章より CPU レベルの脆弱性緩和技術の詳細について説明する。これまで提案された緩和技術は大きく以下のカテゴリに分類される。

- Control flow hijacking からの防御による脆弱性緩和技術
  - Intel Control-flow Enforcement Technology (以下、CET)
  - Arm Branch Target Identification (以下、BTI)
- Spatial/Temporal memory safety の violation の検知による脆弱性緩和技術
  - Arm Memory Tagging Extension (以下、MTE)
- Code pointer integrity の violation の検知による脆弱性緩和技術
  - Arm Pointer Authentication Code (以下、PAC)

2.1 Intel CET

Intel CET は Tiger Lake 世代以降の Intel CPU に搭載された脆弱性緩和技術である。以下の 2 つによって構成される。

- Shadow Stack (以下、SHSTK): backward-edge の control flow hijacking から防御する技術
- Indirect Branch Tracking (以下、IBT): forward-edge の control flow hijacking から防御する技術

2.2 Intel CET SHSTK

SHSTK とはスレッド作成時に OS が確保する stack 領域 (以下では data stack と呼称) とは別に用意された、耐タンパ性を有した [*1]メモリ領域であり、この領域を使った脆弱性緩和技術である。以下、緩和技術そのものについて指す場合には SHSTK、領域自体を指す場合には shadow stack と表記する。

SHSTK が有効になっている Intel CPU は call 命令実行時、return address の値を data stack と shadow stack の両方に push する (図 2-1 を参照)。shadow stack に push された値は、次の ret 命令の実行時に pop され、data stack から pop された値との比較が行われる。この比較結果は通常同じになるが、攻撃者によって return address が書き換えられた場合には異なる値となる。その場合、Control Protection 例外が発生し、プログラムは終了する。

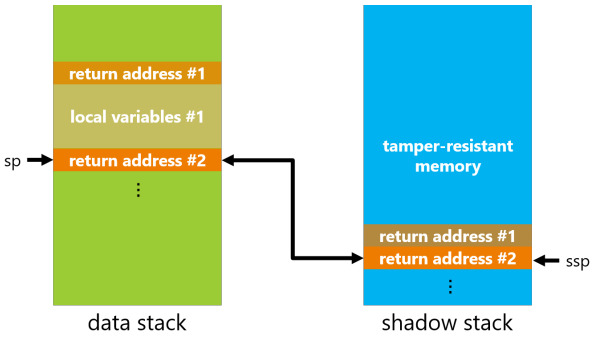

図 2-1 data stack と shadow stack に push される return address の対応関係を示した図 (ここで、sp は data stack のトップを示し、ssp は shadow stack のトップを示す)

先述の通り、shadow stack は耐タンパ性を有しているため、攻撃者が data stack に存在する return address の値の書き換えと同時に shadow stack の該当する値を書き換えることは困難である。そのため、return address の書き換えによる backward-edge の control flow hijacking に対する強力な緩和技術として機能する。

また、SHSTK は return address の改竄検知を追加の命令の実行なしに行えるという利点も持ち合わせている。つまり、call と ret 命令実行時に shadow stack への push と pop が行われ、さらに data stack にある return address と一致することの確認まで自動的に行われる。そのため、パフォーマンスに与える影響を最小限に抑えることが可能である。

ここで、stack canary による緩和技術との比較について述べておく。SHSTK は stack canary と比較してよりバイパスされにくい緩和技術である。stack canary では連続したデータを書き換える関数 (strcpy や memcpy など) でのオーバーフローによる return address の書き換えからしか防御できない。しかし、SHSTK の場合、そ

[*1] ここで「耐タンパ性を有する」とは shadow stack 専用のメモリ保護属性により、通常のロードとストア命令による変更が制限されていることを指す。

れ以外の場合 (例えば、format string bug があり、任意のアドレスが書き換えられる状況など) であっても防御可能である。

一方、SHSTK の運用にあたって誤検知への考慮が必要である。例えば、setjmp や longjmp などによる大域脱出やコンテキストスイッチが発生した場合、shadow stack に push した return address の値と次の ret 命令の際に取り出される値は異なる。そのため、誤検知し例外が発生する。この問題は、スレッドごとにそれぞれ独立した shadow stack 領域を用意し、shadow stack のトップ (ssp) をコンテキストスイッチのタイミングで適切に切り替えることにより回避可能である。

最後に、SHSTK のソフトウェア実装といえる Return Flow Guard (RFG) [9, 10] との比較から、ソフトウェアレベルで実装することの問題点と CPU レベルで実装することの利点について説明する。

RFG は Microsoft によって試験的に導入された脆弱性緩和技術である。原理としては図 2-2 に示すように、

1. 関数のプロローグにおいて mov rax, [rsp] により return address を取り出し、mov fs:[rsp], rax で shadow stack にその値をコピー
2. 関数のエピローグにおいて mov rcx, fs[rsp] により shadow stack から値を取り出し、return address の値と比較

という流れで、return address の改竄を検知し、プログラムを終了させるというものである。

しかしながら、RFG は 検証の結果、不採用となっている。不採用となった経緯と理由については文献 [10] に詳しいが、要因の 1 つとして race condition の問題が挙げられている。ここでいう race condition の問題とは、適切なタイミングで data stack 上の return address の値を書き換えることで、改竄検知をバイパスできることを指す。先述したとおり、RFG は関数 call の直後に、shadow stack に return address の値を push する。ここで関数 call の直後、 shadow stack への push の直前というタイミングで data stack の return address が改竄される場合を考える。この場合、shadow stack に push されるのは改竄後の return address であるため、改竄されているにも関わらず、関数エピローグでの改竄検知をバイパスできる。Intel CET SHSTK の場合、関数 call と同時に shadow stack へ return address の値が push されるため、この race condition の問題は発生し得ない。SHSTK はソフトウェアレベルでの解決が困難な race condition の問題を CPU レベルで解決しているといえる。

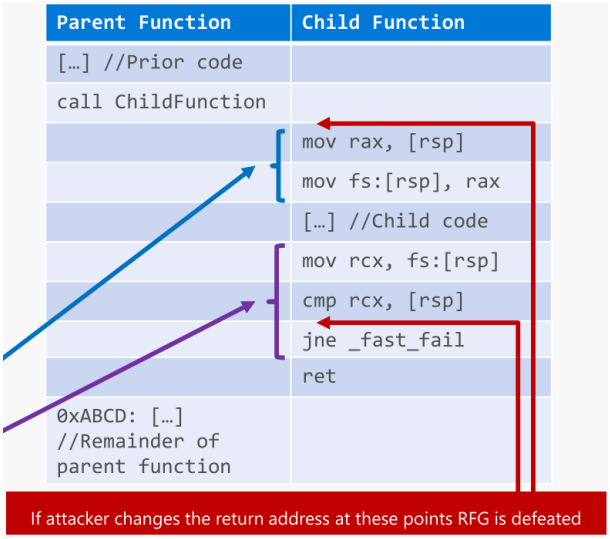

図 2-2 RFG において関数のエピローグとプロローグに挿入される検査コードを示した図 (文献 [10] より転載)

この race condition の問題は call 命令実行と同時に return address を data stack に push する x86/x64 という ISA の仕様により生じている。return address を専用のレジスタに格納する、例えば Arm や RISC-V の場合にはこの問題は発生しない。そのため、他の ISA では race condition の問題が生じない形でソフトウェアでの実装が実現できる。すでに、Android 10 において Shadow Call Stack (SCS) と呼ばれる SHSTK のソフトウェア実装が導入されている [11]。

2.3 Intel CET IBT

IBT は、間接分岐命令の分岐先を制限することで、forward-edge の control flow hijacking から防御する技術である。IBT では endbr という新しい命令が追加されており、これが間接分岐命令の分岐先の “目印” となる。間接分岐命令の実行後、次の命令が endbr の場合、プログラムは正常に実行を継続するが、endbr でない分岐先に移った場合、例外が発生しプログラムは終了する。攻撃者は関数ポインタの書き換えにより、JOP ガジェットやシェルコードなどに制御を移そうとするが、IBT が有

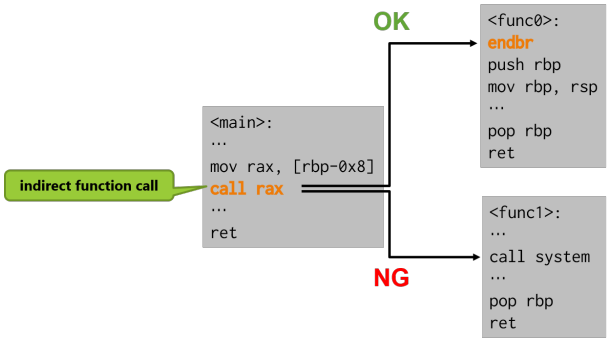

図 2-3 IBT の動作の概要を表した図

効になっている場合、それが困難となる。

上記の振る舞いは CPU の内部的には以下のようになっている。

- 間接分岐命令の実行後、CPU は WAIT_FOR_ENDBRANCH 状態に遷移
- 次の命令として endbr を実行した場合、WAIT_FOR_ENDBRANCH から SUSPEND 状態に遷移
- 次の命令として endbr 以外を実行した場合、Control Protection 例外を送出

IBT をソフトウェア実装したものが CFG といえる [12]。CFG では信頼できる間接分岐先アドレスをビットマップとして管理する。間接分岐命令を実行するたびに、分岐先アドレスがビットマップに含まれる信頼できるアドレスかを検査し、含まれていない場合に例外を出す仕組みとなっている。

IBT は後方互換性を保つよう設計されており、Intel CET をサポートしていないプロセッサでは endbr 命令は nop として扱われる。また、IBT を有効にせずコンパイルした共有ライブラリとリンクし、実行可能とするための機能も有している。これは、IBT による分岐先の検査を無効にする例外リストによって実現されている。この例外リストは legacy code page bitmap と呼ばれ、ページ単位で間接分岐命令の次の命令が endbr でなくともよいアドレスを指定できる。

2.4 Arm Branch Target Identification (BTI)

Arm BTI は Armv8.5-A で利用可能な拡張機能である。BTI は IBT と同様に、br や blr 命令による分岐先を

命令とオペランド	許される分岐元の命令
`bti c`	link 付きの間接分岐命令 (blr など)
`bti j`	link なしの間接分岐命令 (br など)
`bti cj`	全ての間接分岐命令

図 2-4 Arm bti 命令のオペランドとしてとり得る値の一覧とその意味

制限することで、forward-edge の control flow hijacking から防御するための技術である。endbr に相当する命令として bti 命令が追加されている。IBT と異なり、分岐元で実行された命令が br か b もしくはその両方であることを、bti 命令の引数として指定できる (図 2-4)。

この制約により、利用可能な JOP ガジェットの数が IBT より制限される。

2.5 Arm Memory Tagging Extension (MTE)

Arm MTE は tagged memory を実現する Armv8.5-A で利用可能な拡張機能である。tagged memory とはメモリを複数のブロックに分割し、それぞれに tag bit と呼ばれるメタデータを付与したメモリのことである。この tag bit の値を色に見立てるとメモリが色分けされることから memory coloring とも呼ばれる。

tagged memory ではメモリアクセス時のポインタにもメタデータが付与される。ポインタのメタデータが"鍵"に、メモリのメタデータが"錠"に対応する。この 2 つのメタデータが一致する場合にはアクセスが成功する。一方、一致しない場合には例外を送出し、プログラムは終了する。

Arm MTE の場合、tag bit としてポインタの上位 4 bit にメタデータが付与される。仮想アドレスから物理アドレスの変換は、上位 8 bit を除外した上で行われ、正常なアクセスが保証される (これは Top-Byte-Ignore と呼ばれる)。アクセス時にはデータへのアクセスに加え pointer のタグデータと、アクセスするメモリに含まれるタグデータの 2 つが照合され、不一致の場合には例外が送出される。

Arm MTE による heap オーバーフローを検知する例を以下に示す。図 2-5 では buffer という領域が malloc を通じて heap 領域に確保されている。malloc は buffer の領域に 0b1100 というタグデータを付与し、buffer ポインタの上位 4 bit にも 0b1100 という同じタグデータ

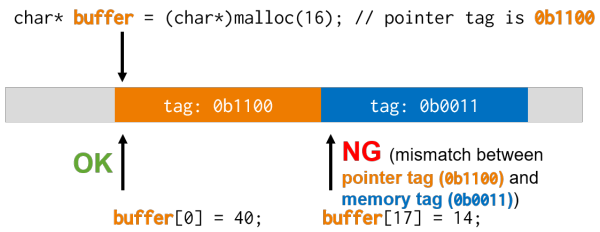

図 2-5 Arm MTE による配列の領域外参照の検知の例

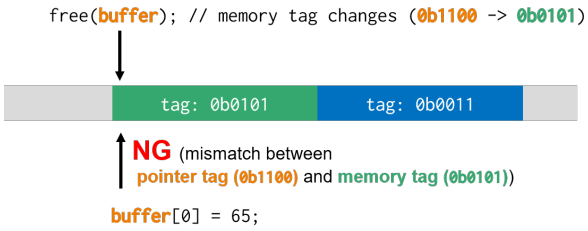

図 2-6 Arm MTE による Use-After-Free 検知の例

を付与する。buffer[0] = 40; とアクセスした場合、buffer ポインタの持つタグデータとメモリに存在するタグデータが一致し、プログラムは通常通り実行を継続する。一方、buffer[17] = 14; とアクセスした場合、buffer ポインタの持つタグデータとメモリに存在するタグデータが一致しない。そのため、アクセス時には例外が送出される。これにより、領域を超えてのアクセスが制限され、heap オーバーフローを検知できる。

また、Arm MTE は heap オーバーフローなどの spatial memory safety の violation の検知に加え、UAF などの temporal memory safety の violation の検知にも利用できる。

UAF を検知例を以下に示す。図 2-6 は malloc により確保された buffer が free を通じて解放されたあと、buffer[0] = 65; により再びアクセスされることを示す。free では渡された領域に別のタグを振り直す処理が行われる。これにより、buffer が指す領域のタグデータは 0b1100 から 0b0101 へと変化する。そのため、使用済みの領域に buffer ポインタを用いてアクセスする場合、buffer ポインタのもつタグデータ 0b1100 と領域のタグデータ 0b0101 が一致しないため、例外が送出される。

Arm MTE は heap 領域のみならず、stack 領域のオーバーフロー検知にも利用できる。これは、stack フレーム生成時、各変数にそれぞれ別のタグを付与することで

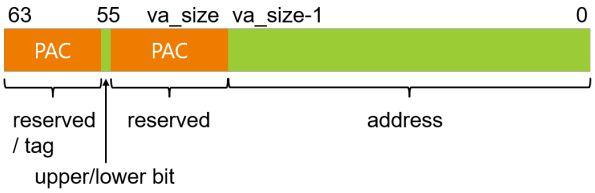

図 2-7 64-bit Arm アーキテクチャのポインタのうち、仮想アドレスとして使っている bit とそれ以外の bit の内訳を示した図 (例えば Linux の場合 va_size の値は 39)

行える。この詳細は第 3 章で説明する。

2.6 Arm Pointer Authentication Code (PAC)

Arm PAC は、ポインタの改竄を検知するためのポインタ認証コード (以下では PAC) をポインタに挿入する技術である。64-bit Arm アーキテクチャでは、ポインタには仮想アドレスとして使っていない bit が存在する (図 2-7)。この使っていない bit にメタデータとして PAC を挿入するのが Arm PAC である。

Arm PAC ではポインタに PAC を挿入する命令と PAC 付きポインタの認証命令がそれぞれ用意されている。

- pac* 命令 (paciasp や pacda など): PAC を対象のポインタ、コンテキスト、キーを元に生成し、ポインタの上位ビットに挿入
- aut* 命令 (autiasp や autda など): PAC 付きポインタの認証を行い、認証に成功した場合には PAC なしのポインタを生成、失敗した場合には無効なポインタを生成

認証に失敗した場合には無効なポインタが取り出されるため、例えばプログラムカウンターが無効な値になり、アクセス違反でプログラムは終了する。これにより、return address や関数ポインタの改竄の検知を行える。

return address の改竄検知を例として挙げる (図 2-8)。図 2-8 で示した例では main の最初にある命令 paciasp により、lr の上位ビットに認証コードが挿入される。lr の値は次の stp 命令により stack に退避され、main 関数の処理が実行された後、ldp 命令により復帰される。復帰された lr の値には認証コードが含まれており、続く autiasp による認証が行われ、生のポインタが取り出さ

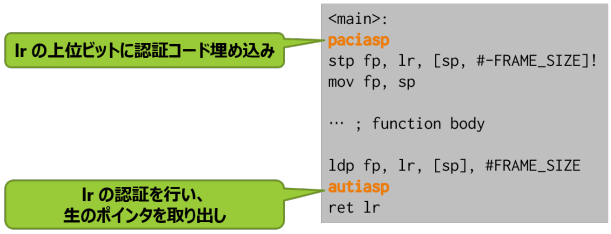

図 2-8 Arm PAC による return address への PAC 挿入をソースコード例とともに示した図

れる。main 関数の処理内で stack の内容が改竄された場合、認証直後の lr の値が無効な値となり、ret lr 命令実行時にアクセス違反によりプログラムが終了する。これにより、backward-edge の control flow hijacking から防御が可能となる。

Arm PAC 相当の機能をソフトウェアで実装する場合、PAC 生成ごとに暗号化する関数の呼び出しが必要になり、パフォーマンスの低下が見込まれる。一方で Arm PAC では暗号化がハードウェア実装されており、高速に行える。return address への PAC 挿入のように、頻繁に暗号化が必要となる場合、暗号化処理がハードウェア実装されていることで、パフォーマンスへ与える影響を最小限に抑えることができる。

Arm PAC は Armv8.3 から optional 扱いの機能として導入されている。iPhone で利用されている Apple A12 Bionic 以降の Arm CPU では PAC はすでに利用可能であり、iOS の一部のプログラムでは、PAC 挿入とその認証処理が行われている [6]。

3. OS とコンパイラにおけるサポート状況

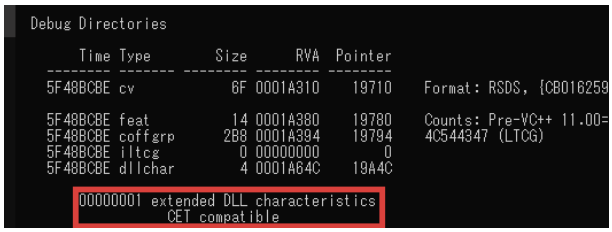

図 3-1 /CETCOMPAT フラグを有効にしてコンパイルしたバイナリの extended DLL characteristics の対応するフラグが有効になることを示した図

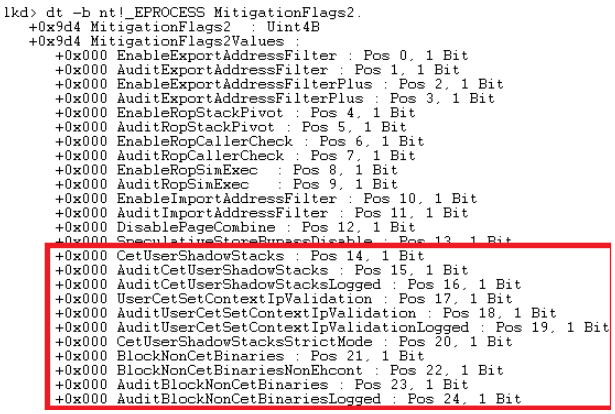

図 3-2 EPROCESS 構造体のメンバー MitigationFlags2 の各ビットのうち、Intel CET による緩和技術に関係するフラグのみを赤枠で囲んで示した図

```
        /* check MitigationFlags2Value.CetUserShadowStacks */
if (((newProcess->field_0x9d4 & 0x4000) != 0) &&
   ((ProcessContext == 0 || ((*(uint *)(ProcessContext + 4) & 0x8000) == 0)))) {
        /* set KTHREAD.CetUserShadowStack */
  Thread->field_0x74 = Thread->field_0x74 | 0x100000;
}
```

図 3-3 スレッド生成時に呼ばれる PspAllocateThread で EPROCESS の CetUserShadowStacks が有効な場合、KTHREAD の CetUserShadowStack の該当するフラグも合わせて設定されることを示した図

3.1 Intel CET SHSTK

SHSTK は Windows と Linux、またそれぞれの OS で使われているコンパイラである GCC と MSVC においてサポート済みである。ここでは Windows のサポート状況についてそのリバースエンジニアリング結果も含めて紹介する。Linux については触れないが、SHSTK サポート用のパッチがすでに提供されている [13]。

以下では Windows 10 (build version 20190.1000) の結果を示す。なお、Insider Preview Builds の Windows の結果となるため、将来的に変更の可能性があることを注記しておく。

また、過去 Black Hat Asia 2019 において Windows における SHSTK サポートに関するリバースエンジニアリングの結果が公表されているが [14]、今回ここに示す結果と一部違う箇所がある。

3.1.1 コンパイラサポート

MSVC (2019 version 16.7 以降) では新しい linker flag として /CETCOMPAT が追加されており、このフラグを有効にしてビルドすると、extended DLL characteristics の対応するフラグが有効となる (図 3-1 を参照)。このフラグが有効になった実行ファイルが実行される際、最新の Windows カーネルでは Intel CET SHSTK の CPU サポートを確認後、SHSTK による防御が有効となる。

3.1.2 Kernel opaque structure の変更

ここから、OS におけるサポート状況について説明する。

まず EPROCESS と KTHREAD 構造体に加えられた、SHSTK に関係する変更点について見ることとする。以下に示すのは EPROCESS 構造体の MitigationFlags2 のメンバーの内容である (図 3-2 を参照)。

図 3-2 では赤枠で SHSTK に関係するビットフラグを示している。このフラグのうち CetUserShadowStacks が有効になっている場合、図 3-3 に示すように、ユーザーモードにおいて SHSTK が有効になり、スレッドの生成時に KTHREAD のメンバーである CetUserShadowStack が 1 に設定される [*1]。

また、図 3-4 で、KTHREAD 構造体には CetUserShadowStack に加え CetKernelShadowStack というフィールドが存在することに注目してほしい。これについては

[*1] ここで、EPROCESS 構造体のメンバーに見られるフラグが CetUserShadowStacks と複数形なのに対し、KTHREAD 構造体に含まれるメンバーでは CetUserShadowStack と単数形となっている。スレッドごとにそれぞれ shadow stack を持つため、1 つのプロセスは複数の shadow stack を管理する。そのため、EPROCESS 構造体では複数形、KTHREAD 構造体では単数形となる。

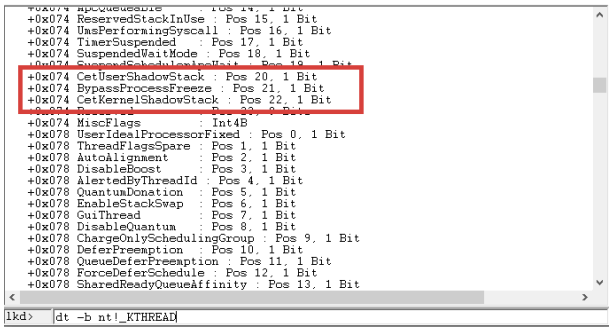

図 3-4 KTHREAD 構造体のメンバーに含まれる CetUserShadowStack と CetKernelShadowStack を示した図

```
JZ              LAB_1800afc81
XOR             EDX,EDX
RDSSPD          EDX                     Read current ssp

MOV             R9,qword ptr [RCX + 0x528] Read next ssp

RSTORSSP        qword ptr [R9]          Validate restore token and restore context

SAVEPREVSSP                             Save previous ssp with restore context
SUB             RDX,0x8
MOV             qword ptr [RAX + 0x528],RDXSave previous ssp in context
```

図 3-5 kernelbase!SwitchToFiber における ssp のコンテキスト退避と復帰処理の逆アセンブル結果

後述するが、Black Hat Asia 2019 の結果 [14] から Windows が更新され、現在ではカーネル側でも SHSTK のサポートが進みつつある。

3.1.3 コンテキストスイッチ処理

OS 側ではコンテキストスイッチが起きた場合の対応も合わせて行われている。コンテキストの退避処理は、ssp を、XState と呼ばれるコンテキストを保持する構造体のメンバーとして退避することで行われる [15]。

図 3-5 に示すのは kernelbase!SwitchToFiberContext という Fiber に実行を移す関数の一部だが、rdssp rdx により ssp の読み出し、rstorssp 及び saveprevssp による切り替え先の ssp のコンテキストの設定、現在の ssp の退避処理が行われている。

その他、syscall 実行時に呼ばれる nt!KiSystemCall64 関数においても、ユーザーモード実行時に保持していた ssp を一度退避する処理が見られる (図 3-6 を参照)。

そして sysret 命令によりカーネルモードからユーザーモードに復帰する過程においてその逆の処理が見られる (図 3-7)。

ここで、GS:[0x9260] と GS:[0x9268] はそれぞれユーザーモードとカーネルモードの ssp の退避と復帰先が保

```
14041ac64 65 48 8b        MOV             RCX,qword ptr GS:[0x9268]
          0c 25 68
          92 00 00
14041ac6d 48 85 c9        TEST            RCX,RCX
14041ac70 74 0c           JZ              LAB_14041ac7e
14041ac72 f3 0f 01 e8     SETSSBSY
14041ac76 f3 0f 01 29     RSTORSSP        qword ptr [RCX]
14041ac7a f3 0f 01 ea     SAVEPREVSSP
```

図 3-6 ユーザーモードからカーネルモード遷移後、ssp を退避する箇所の逆アセンブル結果

```
14041b4db 33 c9           XOR             ECX,ECX
14041b4dd f3 48 0f        RDSSPD          ECX
          1e c9
14041b4e2 65 4c 8b        MOV             R8,qword ptr GS:[0x9268]
          04 25 68
          92 00 00
                    If UserShadowStack enabled, restore its value
14041b4eb 49 83 c0 08     ADD             R8,0x8
14041b4ef 49 3b c8        CMP             RCX,R8
14041b4f2 75 11           JNZ             LAB_14041b505
14041b4f4 65 48 8b        MOV             RCX,qword ptr GS:[0x9260]
          0c 25 60
          92 00 00
14041b4fd f3 0f 01 29     RSTORSSP        qword ptr [RCX]
14041b501 f3 0f 01 ea     SAVEPREVSSP
```

図 3-7 カーネルモードからユーザーモードに復帰する直前、ssp を復帰する箇所の逆アセンブル結果

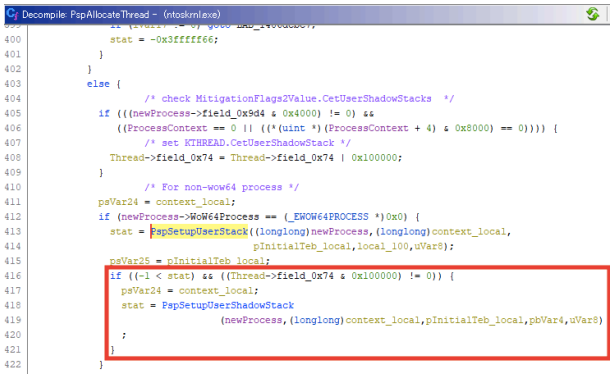

図 3-8 PspAllocateThread 関数で shadow stack 領域の初期化処理を行う箇所のデコンパイル結果

存されていると推察される。

3.1.4 shadow stack 領域の確保処理

Black Hat Asia 2019 の発表時点ではスレッド作成時に shadow stack 領域の確保処理が行われていないと報告されていた [14]。現在の Insider Preview ビルド Windows ではこの確保処理が行われ、ssp に確保した領域の先頭が設定されるよう変更されている。

図 3-8 には、NtCreateUserProcess などから呼ばれる ETHREAD 構造体の初期化処理を行う PspAllocateThread 関数の一部を示す。

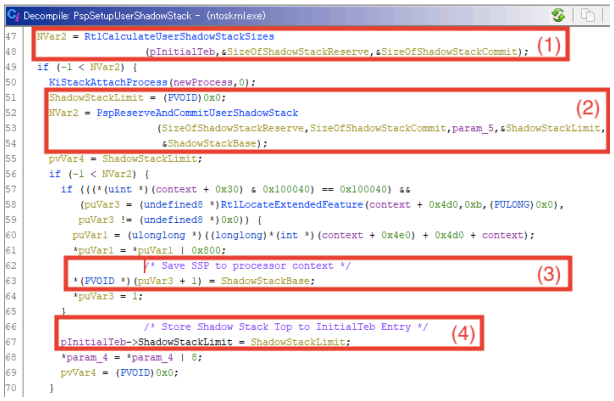

図 3-9 shadow stack 領域の確保および CONTEXT_EX 構造体へのコンテキストの設定処理を行う箇所のデコンパイル結果

ここで、CetUserShadowStacks フラグが有効の場合、PspSetupUserShadowStack 関数が呼び出され、shadow stack 領域の確保と初期化処理が行われる (図 3-9 を参照)。

PspSetupUserShadowStack では以下の処理が行われる。それぞれ、上図の番号の処理内容に対応する説明を記載している。

(1) shadow stack 領域の reserve と commit サイズを RtlCalculateUserShadowStackSizes により計算
(2) (1) で計算したサイズ分を PspReserveAndCommitUserShadowStack により reserve して commit
(3) (2) において確保した領域の stack のトップにあたる値を取り出し、コンテキストに値として設定
(4) Initial TEB に shadow stack の境界にあたるアドレスを代入

RtlCalculateUserShadowStackSizes は第 1 引数の INITIAL_TEB 構造体に含まれる情報から reserve と commit 時のサイズをそれぞれ計算する (図 3-10 を参照)。shadow stack の reserve サイズと data stack のサイズを同じに設定しているため、shadow stack が先に枯渇することはない。また、commit サイズがデータ stack の 1/10 程度と、物理メモリへの割り当てサイズは比較的小さめに設定されている。

PspReserveAndCommitUserShadowStack は以下のような関数シグニチャとなっており (ソースコード 3.1 を参照)、第 4 引数と第 5 引数にそれぞれ shadow stack

図 3-10 RtlCalculateUserShadowStackSizes 関数のデコンパイル結果

の下限とベースアドレスを格納して返す。shadow stack は MmAllocateUserStack 関数により reserve されたのち、ZwAllocateVirtualMemory により commit される (図 3-11 を参照)。

```
NTSTATUS PspReserveAndCommitUserShadowStack(
    _In_ SIZE_T SizeOfShadowStackReserve,
    _In_ SIZE_T SizeOfShadowStackCommit,
    _In_ uint unknown,
    _Out_ PVOID *pShadowStackLimit,
    _Out_ PVOID *pShadowStackBase);
```

ソースコード 3.1 shadow stack 用のメモリを確保する PspReserveAndCommitUserShadowStack の関数シグニチャ

ここで図 3-11 に示すデコンパイル結果において ZwAllocateVirtualMemory 割り当て時のページ保護属性が PAGE_READONLY になっていることに疑問を感じられた方がいるかもしれない。Intel CET の仕様書 [4] によれば、

> The shadow stack is protected from tamper through the page table protections such that regular store instructions cannot modify the contents of the shadow stack. To provide this protection the page table pro-

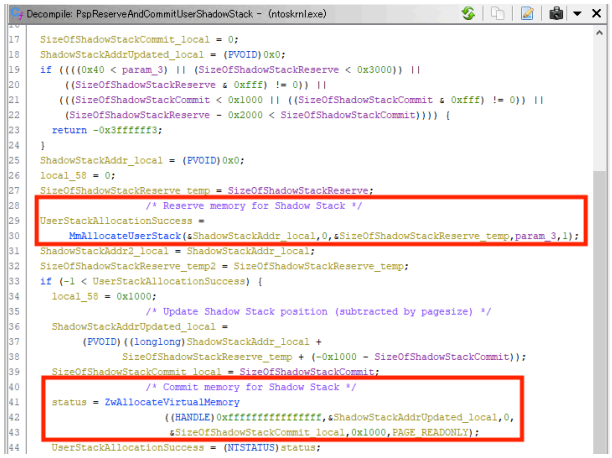

図 3-11 PspReserveAndCommitUserShadowStack 関数内で shadow stack の reserve と commit を行なっている箇所のデコンパイル結果

図 3-12 x86 アプリケーションのプロセスの場合、EPROCESS->WOW64Process は null ではなく、PspSetupUserShadowStack 関数が call されないことを示した箇所

tections are extended to support an additional attribute for pages to mark them as “Shadow Stack” pages.

となっており、shadow stack には専用のページ保護属性が割り当てられるとの記載がある。これについては暫定的に PAGE_READONLY に設定し、今後新規に属性を追加しそれに変更するなどの可能性が考えられるが、詳細は現時点では不明である。

最後に、現時点では x86 アプリケーションに対し shadow stack による脆弱性緩和策が講じられていない点を補足しておく。図 3-12 に示すように、EPROCESS->WOW64Process が null とならない x86 アプリケーションのプロセスの場合、PspSetupUserShadowStack 関数の呼び出しがそもそも行われず、緩和策は無効となっている。x86 アプリケーションは現状まだ広く用いられていること、また Intel CET は 32bit mode でも動作可能であることから、近く実装されることが考えられる。

3.1.5 kernel shadow stack

第 3.1.2 章で述べたとおり、カーネル側での shadow stack 対応も進みつつある。

KiCreateKernelShadowStack という関数が追加されており、KTHREAD 構造体の初期化処理を行う KeInitThread においてこの関数の呼び出しが行わ れている (図 3-13 を参照)。また、KTHREAD 構造体にも KernelShadowStack や KernelShadowStackInitial などカーネルモード実行時に使われる shadow stack の境界やサイズ情報を保持するメンバーが追加されている。さらに、KeInitThread 関数において初期化処理が見られる。

図 3-13 KiCreateKernelShadowStack 関数の呼び出し箇所のデコンパイル結果

残念ながら、KiCreateKernelShadowStack はまだ実装されておらず、常にステータスコードとして 0xc0000002 (STATUS_NOT_IMPLEMENTED) を返す。実際に確保されるサイズやどういった権限で確保されるのかは現在のところ不明であるが、TYPE_OF_MEMORY [*2] にはす

[*2] MEMORY_ALLOCATION_DESCRIPTOR のメンバー Memo-

```
LoaderSkMemory = 0n36
LoaderSkFirmwareReserved = 0n37
LoaderIoSpaceMemoryZeroed = 0n38
LoaderIoSpaceMemoryFree = 0n39
LoaderIoSpaceMemoryKsr = 0n40
LoaderKernelShadowStack = 0n41
LoaderMaximum = 0n42
```

図 3-14 TYPE_OF_MEMORY に追加された LoaderKernelShadowStack 変数

でにカーネル用の shadow stack 領域を示す変数が追加されるなど、実装が現在も進行中と推察される (図 3-14 を参照)。

3.2 Intel CET IBT

Linux ではすでに Intel CET IBT のサポートが行われている。また、コンパイラについても GCC version 8 以降ではすでにサポート済みであり、関数の先頭に endbr 命令が自動的に追加される。以下のソースコード 3.2 に例を示す。

```
<__cxa_finalize@plt>:
    endbr64   ; <-------
    bnd jmp QWORD PTR [rip+0x2fad]          ; 3ff8
        <__cxa_finalize@GLIBC_2.2.5>
    nop     DWORD PTR [rax+rax*1+0x0]

<main>:
    endbr64   ; <-------
    sub     rsp,0x8
    lea     rdi,[rip+0xf95]
    call    1050 <puts@plt>
    xor     eax,eax
    add     rsp,0x8
    ret

<_start>:
    endbr64   ; <-------
    xor     ebp,ebp
    mov     r9,rdx
    pop     rsi
    mov     rdx,rsp
    and     rsp,0xfffffffffffffff0
    push    rax
    push    rsp
    lea     r8,[rip+0x146]
    lea     rcx,[rip+0xcf]
    lea     rdi,[rip+0xfffffffffffffb8] # 1060 <
        main>
    call    QWORD PTR [rip+0x2f32] # 3fe0 <
        __libc_start_main@GLIBC_2.2.5>
    hlt
    nop
```

ソースコード 3.2 endbr 命令が関数の先頭に追加された逆アセンブル結果の例

上記アセンブリに矢印で示しているが、関数の先頭全てに endbr64 という命令が追加されていることがわかる。これにより、関数分岐命令の分岐先が全てこの endbr64 始まりのもののみに制限され、JOP などの forward-edge control flow hijacking に利用可能なガジェットが制限される。

標準ライブラリにおいても Intel CET IBT はサポートされており、関数の先頭全てにおいて endbr 命令が挿入され、分岐先が制限されている。Linux カーネル側でもすでにサポートが行われており、IBT 未サポートの共有ライブラリの関数を呼び出した時の誤検知の対策も実装され、積極的な導入が進んでいる [16]。

一方、Windows では IBT のサポートは発表されていない。おそらく CFG の後継である xFG として

1. 間接分岐命令の分岐先が信頼できるアドレスであること
2. 関数呼び出しの場合、呼び出し先の関数シグニチャが一致すること

の 2 点を確認する緩和技術を実用化しつつあること [17]、実用化できれば xFG が IBT より強い緩和策として機能することが関係していると考えられる。

ちなみに、Arm BTI については Intel CET IBT をベースに Linux でのサポートが進行している [18]。

3.3 Arm PAC

3.3.1 iOS / macOS におけるサポート状況

Arm PAC は iOS / macOS においてすでにサポートされており、カーネル空間とユーザー空間の双方において Arm PAC による防御が有効になっている。去年発表された Apple Silicon 搭載の Mac においても Arm PAC による脆弱性緩和技術の導入が発表されており [19]、Apple 社製品には今後広く普及することが予想される。

コンパイラでもすでに Clang においてサポートされ

ryType において使われる列挙体であり、各メモリの使用用途をローダに示すために用いられる。

```
; void __fastcall sign_thread_state(arm_context *state, u64 pc, u64 cpsr, u64 lr)
sign_thread_state                       ; CODE XREF: fleh_dispatch64+9C↑p
                                        ; Lell_sp0_synchronous_vector_long_impl_S3_4_c15_
                PACGA           X1, X1, X0
                AND             X2, X2, #NOT 0x20000000 ; Carry flag
                PACGA           X1, X2, X1
                PACGA           X1, X3, X1
                STR             X1, [X0,#arm_context.pac_sig]
                RET
; End of function sign_thread_state
```

図 3-15 sign_thread_state 関数の逆アセンブル結果 (文献 [20] から転載)

ており、mbranch-protection=pac-ret オプション付きでコンパイルすると、lr の上位ビットに PAC を挿入する処理が見られる。

iOS ではその他、関数ポインタにも PAC を挿入する処理が導入されており、forward-edge control flow hijacking への対策も施されている。また、スレッドのコンテキストが改竄されていないことを保証する用途でも Arm PAC が利用されている。以下文献 [20] の発表からの引用となるが、machine_thread_create というスレッド作成関数に sign_thread_state という関数の呼び出しが見られる。この関数は pc、cpsr、lr の 3 つのレジスタを使った認証コードを生成し、コンテキストに格納する (図 3-15 を参照)。

そして、復帰時には pc、cpsr、lr の値を用い再度認証コードを生成し、コンテキストに保存されている値と一致することを確認する。これにより、コンテキストが改竄されていないことを保証する。

3.3.2 Windows におけるサポート状況

本文書の執筆現在、Microsoft から公式にアナウンスはされていないが、Windows でも Arm PAC の将来的なサポートを示唆する処理が加えられている。

図 3-16 は、WinDbg 上で EPROCESS 構造体のメンバー MitigationFlags2 を表示したものである。PointerAuthUserIp、AuditPointerAuthUserIp、AuditPointerAuthUserIpLogged の 3 つの新しいフラグが追加されていることがわかる。

その他 ntoskrnl には KePointerAuthEnabled や KePointerAuthSupported など Arm PAC サポートの有無、機能の有効と無効化を管理するフラグが追加されており、ID_AA64ISAR1_EL1 レジスタ (実装されている命令セットを管理するシステムレジスタである。各ビットマップの意味は 図 3-17 を参照) にアクセスし、これらのフラグを更新する処理が追加されている。これは ntoskrnl の entry ポイントから呼ばれる KiInitializeBootStructure を介して KiInitializePointerAuth において行われる。

```
lkd> dt -b nt!_EPROCESS MitigationFlags2
   +0x9d4 MitigationFlags2 : Uint4B
   +0x9d4 MitigationFlags2Values :
      +0x000 EnableExportAddressFilter : Pos 0, 1 Bit
      +0x000 AuditExportAddressFilter : Pos 1, 1 Bit
      +0x000 EnableExportAddressFilterPlus : Pos 2, 1 Bit
      +0x000 AuditExportAddressFilterPlus : Pos 3, 1 Bit
      +0x000 EnableRopStackPivot : Pos 4, 1 Bit
      +0x000 AuditRopStackPivot : Pos 5, 1 Bit
      +0x000 EnableRopCallerCheck : Pos 6, 1 Bit
      +0x000 AuditRopCallerCheck : Pos 7, 1 Bit
      +0x000 EnableRopSimExec : Pos 8, 1 Bit
      +0x000 AuditRopSimExec : Pos 9, 1 Bit
      +0x000 EnableImportAddressFilter : Pos 10, 1 Bit
      +0x000 AuditImportAddressFilter : Pos 11, 1 Bit
      +0x000 DisablePageCombine : Pos 12, 1 Bit
      +0x000 SpeculativeStoreBypassDisable : Pos 13, 1 Bit
      +0x000 CetUserShadowStacks : Pos 14, 1 Bit
      +0x000 AuditCetUserShadowStacks : Pos 15, 1 Bit
      +0x000 AuditCetUserShadowStacksLogged : Pos 16, 1 Bit
      +0x000 UserCetSetContextIpValidation : Pos 17, 1 Bit
      +0x000 AuditUserCetSetContextIpValidation : Pos 18, 1 Bit
      +0x000 AuditUserCetSetContextIpValidationLogged : Pos 19, 1 Bit
      +0x000 CetUserShadowStacksStrictMode : Pos 20, 1 Bit
      +0x000 BlockNonCetBinaries : Pos 21, 1 Bit
      +0x000 BlockNonCetBinariesNonEhcont : Pos 22, 1 Bit
      +0x000 AuditBlockNonCetBinaries : Pos 23, 1 Bit
      +0x000 AuditBlockNonCetBinariesLogged : Pos 24, 1 Bit
      +0x000 XtendedControlFlowGuard : Pos 25, 1 Bit
      +0x000 AuditXtendedControlFlowGuard : Pos 26, 1 Bit
      +0x000 PointerAuthUserIp : Pos 27, 1 Bit
      +0x000 AuditPointerAuthUserIp : Pos 28, 1 Bit
      +0x000 AuditPointerAuthUserIpLogged : Pos 29, 1 Bit
      +0x000 CetDynamicApisOutOfProc : Pos 30, 1 Bit
```

図 3-16 EPROCESS 構造体のメンバー MitigationFlags2 に含まれる Arm PAC に関係すると推察されるフラグ

Field descriptions

The ID_AA64ISAR1_EL1 bit assignments are:

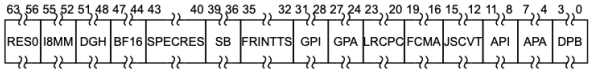

図 3-17 ID_AA64ISAR1_EL1 のビットマップの各フィールド (文献 [21] より転載)

KiInitializePointerAuth の最初の処理を少し紐解いてみる。図 3-18 では ID_AA64ISAR1_EL1 レジスタの APA へのアクセス (図 3-18 における (1) の箇所) と API へのアクセス (図 3-18 における (2) の箇所) が見られる。

KiInitializePointerAuth では、まず ID_AA64ISAR1_EL1 のビットを一通り調べたのち、プロセッサが Arm PAC をサポートしている場合、グローバル変数の KePointerAuthEnabled と KePointerAuthSupported を有効にする (図 3.3.2 を参照)。

現時点では Arm PAC に関する処理はこれ以上見つからない。しかし、今後 OS レベルでサポートされる可能性は高いと考えられ、実装が進むと予想される。

```
void KiInitializePointerAuth(uint param_1,longlong param_2)

{
  ulonglong uVar1;
  ulonglong uVar2;
  byte bVar3;
  uint qarma_implemated;

  uVar1 = cRead_8(id_aa64isar1_el1);
                    /* Indicates whether QARMA or Architected algorithm is implemented in the PE for
                       address authentication */
  qarma_implemated = 0;
  if ((uVar1 >> 4 & 0xf) != 0) {
    qarma_implemated = 3;
  }
  uVar2 = (ulonglong)qarma_implemated;
  uVar1 = cRead_8(id_aa64isar1_el1);
  bVar3 = (byte)qarma_implemated;
                    /* Indicates whether an IMPLEMENTATION DEFINED algorithm is implemented in the
                       PE for address authentication */
  if ((uVar1 >> 8 & 0xf) != 0) {
    if ((qarma_implemated & 1) != 0) {
                    /* WARNING: Subroutine does not return */
      KeBugCheck2(0x5d,9,0,0,0,0);
    }
    bVar3 = bVar3 | 1;
    uVar2 = (ulonglong)bVar3;
  }
```

(1)

(2)

図 3-18 ID_AA64ISAR1_EL1 レジスタの APA と API にアクセスする箇所のデコンパイル結果

```
if (param_1 == 0) {
  KePointerAuthSupported = bVar2;
  wil_details_FeatureReporting_ReportUsageToService
            (&Feature_Pointer_Auth_User__private_reporting,0x1878288,0,0,
             &Feature_XFG_User_logged_traits,0);
  if ((*(uint *)(*(longlong *)(param_2 + 0xf0) + 0x84) >> 0x10 & 1) != 0) {
    KePointerAuthEnabled = KePointerAuthEnabled | 4;
    KePointerAuthKernelIpKey = *(undefined8 *)(*(longlong *)(param_2 + 0xf0) + 0xe58);
    DAT_141201950 = *(undefined8 *)(*(longlong *)(param_2 + 0xf0) + 0xe60);
  }
}
```

図 3-19 KiInitializePointerAuth 関数内で KePointerAuthEnabled などのフラグを有効にする箇所のデコンパイル結果

3.4 Arm MTE

すでに Linux v5.4 ではシステムコール時に渡すポインタとして tagged pointer を取れるよう AArch64 Tagged Address ABI が新しく追加されている。Arm MTE のユーザー空間におけるタグチェック機能の実装は Linux v5.10 でマージされている [22]。

Android では Linux kernel v5.4 でマージされた AArch64 Tagged Address ABI のパッチを 4.14 (Pixel 4) 以降ですでに含んでおり、heap 領域の memory tagging もすでにサポートしている。これは scudo という Android で用いられているメモリアロケータを介して提供されており [23]、malloc 実行時の tagging と free 時の tagging がそれぞれ実装されている。

また、stack 領域についてもサポートが行われており、stack に確保された変数それぞれについてタグを割り当てる処理が関数プロローグに出力される。例えば、以下のソースコード 3.3 を考える。

```
#include <stdio.h>

__attribute__((noinline))
void func(int id) {
        int x = 10;
        char buffer[10] = {};
        for (int i = 0; i < 10; i++) buffer[i] =
            0;
        printf("%p␣%p\n", &x, &buffer[id]);
}

int main(int argc, char* argv[]) {
        func(argc);
}
```

ソースコード 3.3 Arm MTE による stack 領域の memory tagging を確認するためのサンプルコード

func は int 型の変数と char 型の要素数 10 の配列の 2 つを stack 領域に確保し、それぞれのアドレスを標準出力に出力する。

Arm MTE を有効にしてコンパイルすると、x と buffer という領域それぞれに別のタグを割り当てる処理が見られる。実際、-march=armv8+memtag -fsanitize=memtag を有効にし、Clang でコンパイルした結果を示すと、ソースコード 3.4 のように stack 上の変数にタグを付与する処理が見られる。

```
<func>:
    sub     sp, sp, #0x30           ; stack フレームを生成
    stp     x29, x30, [sp, #32]
    irg     x8, sp                  ; x8 に sp の値をコピーし、タグを挿入
    ; C 言語のソースコードにおける変数 x を指すポインタにタグを付与
    addg    x1, x8, #0x10, #0x0
    ; C 言語における変数 buffer を指すポインタにタグを付与
    ; (ただし、変数 x に付与したものとは別のタグになるように設定)
    addg    x8, x8, #0x0, #0x1
    add     x2, x8, w0, sxtw
    adrp    x0, 400000 <_init-0x418>
    mov     w9, #0xa
    add     x0, x0, #0x698
    add     x29, sp, #0x20
    ; x9 xzr の 2 つのデータを x1 にストアし、x1 に付与されたタグも合わせてストア
    ; 格納するアドレスを指すレジスタに存在するタグをタグ付きメモリにストア (この場合 x1)
    stgp    x9, xzr, [x1]
    ; x8 が指すアドレスに 0 をストアし、x8 に付与されたタグも合わせてストア
```

```
stzg      x8, [x8]
bl        400480 <printf@plt>
ldp       x29, x30, [sp, #32]
stg       sp, [sp]
stg       sp, [sp, #16]
add       sp, sp, #0x30
ret
```

ソースコード 3.4　Arm MTE による stack 領域の memory tagging 処理を示すアセンブリコード

アセンブリにコメントとして示しているが、処理の概要は次のようになる。

1. irg 命令により、sp にタグを付与したポインタを生成 (x8 に代入)
2. addg 命令により、stack 上での格納先アドレスを作成した後、タグを付与 (この時、変数 x と buffer に付与されるタグの値はそれぞれ異なるように設定)
3. sgtp 命令と stzg 命令によりタグとデータの両方をストア

タグの付与が必要になるため、実行すべき命令数が若干増加する。上記プログラムの場合には、stack に保持する変数が比較的少ないため、さほど命令数が増えるわけではない。MTE 有効と無効の場合でそれぞれ 16 命令と 18 命令となっている。しかし、stack に保持する変数が多くなる場合にはその限りではなく、命令数増加によるパフォーマンス低下の可能性がある。

4. バイパス手法の研究

Arm PAC と Intel CET を除き、これらの脆弱性緩和技術を実際に搭載した CPU は執筆現在出回っておらず、バイパス手法の研究は十分に行われていない。Arm PAC については、先述したとおり Apple A12 Bionic 以降の CPU に実装され、iOS に Arm PAC を使った緩和策が実装されている。そのため、実機を用いてバイパス手法についての研究が行われている。成果は Black Hat USA 2020 [20] で発表されるなど、研究は活況を呈している。

カーネル側で使われている Arm PAC バイパスについては Google Project Zero チームによりブログ [6] と Black Hat USA 2020 [20] で発表されている。また、ユーザーランドの PAC バイパスについても DEF CON 27 において発表されている [24]。

ここではユーザーランドの PAC バイパスの研究 [24] について簡単に概要を紹介する。

PAC バイパスでは auth に成功する PAC を偽装する必要がある。PAC の入力として与えるのは、対象のポインタ、コンテキスト、キー の 3 つになる。このうちコンテキストとキーの 2 つが推定できれば PAC を偽造でき、auth 系の命令実行時に無効なポインタが生成されることからバイパスに成功する。

まず、コンテキストの値の推定については iOS は関数ポインタを null コンテキストで認証コードを生成するよう実装しているため必要ない。

次にキーについては Arm PAC で使えるキーのうち A-key (関数ポインタの認証コードの生成で使われるもの) が複数のユーザープロセスで使いまわされることが報告されている。そのため、キーの値は推定できないが、差し替え対象となる関数ポインタの認証コードを生成し、PAC 付きポインタの偽造は可能である。

PAC の偽造を防ぐ意味では各プロセスで異なる PAC キーを使うべきだが、COW による最適化を優先しこのような処置が取られている。仮に A-key において各プロセスで異なるキーを用いた場合、共有ライブラリに含まれる関数ポインタを認証コードがプロセスごとに異なる。そのため、COW による最適化が行えず、プロセスごとに共有ライブラリのコピーを作る必要がある。このコピー生成のコストを削減する目的で A-key は複数プロセスで同じ値が共有されている。

5. まとめと今後の展望

Intel と Arm において近年採用が進みつつある CPU レベルの脆弱性緩和技術の紹介と各 OS 及びコンパイラにおけるサポート状況について紹介した。実機として出回っているものはまだ少ないものの、すでに積極的な導入が始まっている。オーバーヘッドが大きすぎるがために、コンパイラやカーネルといった低レイヤーのソフトウェアレベルでの実装が困難だった強力な緩和策を、CPU レベルというさらに低レイヤーで実装し実用化していると言える。

一方、冒頭でも述べたとおり、脆弱性緩和技術と攻略技術は常にイタチごっこの関係にあり、強力な緩和策であってもバイパスされうることを念頭におく必要がある。実際、第 4 章で述べたとおり、CPU レベルの脆弱性緩和技術も、OS 側の実装の問題でバイパスされる可能性がある。OS を常に最新の状態に保っておくことが今後も重要である。

参考文献

[1] Robert C Seacord. *Secure Coding in C and C++ (2nd Edition)*. Pearson Education, 2013.

[2] Erik Buchanan, Ryan Roemer, Stefan Savage, and Hovav Shacham. Return-oriented Programming: Exploitation without Code Injection. In *Black Hat USA*, 2008.

[3] Control Flow Guard - Win32 apps | Microsoft Docs. `https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard`. (Accessed on 12/25/2020).

[4] Intel Corporation. *Control-flow Enforcement Technology Specification*, 2019.

[5] Arm Limited or its affiliates. *Providing protection for complex software*, 2020.

[6] Project Zero: Examining Pointer Authentication on the iPhone XS. `https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html`. (Accessed on 08/30/2020).

[7] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pp. 81–116. 2018.

[8] WIN04-C. Consider encrypting function pointers - SEI CERT C Coding Standard - Confluence. `https://wiki.sei.cmu.edu/confluence/display/c/WIN04-C.+Consider+encrypting+function+pointers`. (Accessed on 09/01/2020).

[9] Tencent Security Xuanwu Lab. Return Flow Guard - Tencent Security Xuanwu Lab. `https://xlab.tencent.com/en/2016/11/02/return-flow-guard/`, 2016. (Accessed on 12/25/2020).

[10] Joe Bialek. The Evolution of CFI Attacks and Defenses. In *OffensiveCon*, 2018.

[11] Sami Tolvanen. Google online security blog: Protecting against code reuse in the linux kernel with shadow call stack. `https://security.googleblog.com/2019/10/protecting-against-code-reuse-in-linux_30.html`. (Accessed on 12/25/2020).

[12] Trend Micro. *Exploring control flow guard in windows 10*, 2015.

[13] LKML: Yu-cheng Yu: [PATCH v10 00/26] Control-flow Enforcement: Shadow Stack. `https://lkml.org/lkml/2020/4/29/1364`. (Accessed on 09/27/2020).

[14] Bing Sun and Jin Liu. How to Survive the Hardware-assisted Controlflow Integrity Enforcement. In *Black Hat Asia*, 2019.

[15] Yarden Shafir and Alex Ionescu. R.I.P ROP: CET Internals in Windows 20H1. `https://windows-internals.com/cet-on-windows/`, 2020. (Accessed on 08/28/2020).

[16] Control Flow Enforcement - Part (4) [LWN.net]. `https://lwn.net/Articles/758252/`. (Accessed on 08/30/2020).

[17] David Weston. Advancing Windows Security. In *Bluehat Shanghai*, 2019.

[18] arm64: ARMv8.5-A: Branch Target Identification support [LWN.net]. `https://lwn.net/Articles/789370/`. (Accessed on 08/30/2020).

[19] Explore the new system architecture of Apple Silicon Macs - WWDC 2020. `https://developer.apple.com/`

videos/play/wwdc2020/10686/. (Accessed on 08/30/2020).

[20] Brandon Azad. iOS Kernel PAC, One Year Later. In *Black Hat USA*, 2020.

[21] Arm Limited or its affiliates. *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*, 2020.

[22] The Arm64 memory tagging extension in Linux [LWN.net]. https://lwn.net/Articles/834289/. (Accessed on 01/13/2021).

[23] scudo: Add initial memory tagging support. https://reviews.llvm.org/D70762. (Accessed on 08/30/2020).

[24] Xiaolong Bai. Xiaolong Bai - HackPac Hacking Pointer Authentication in iOS User Space. In *DEF CON 27*, 2019.

株式会社ＦＦＲＩセキュリティ

株式会社ＦＦＲＩセキュリティは、日本発のサイバーセキュリティをリードする専門家集団です。国際的なセキュリティカンファレンスでの研究発表実績もある世界トップレベルのサイバーセキュリティ専門家集団が、 先進的な調査研究により、今後予想される脅威を先読みし、一歩先行くコンセプトで製品・サービスを展開しています。

株式会社ＦＦＲＩセキュリティ
〒100-0005 東京都千代田区丸の内３丁目３番１号 新東京ビル２階
E-mail: research-feedback[at]ffri.jp URL:https://www.ffri.jp/