Demystifying the Secure Enclave Processor

Tarjei Mandt, Mathew Solnik, and David Wang

Azimuth Security, OffCell Research
{tm,dw}@azimuthsecurity.com,ms@offcellresearch.com

Abstract. The secure enclave processor (SEP) was introduced by Apple as part of the A7 SOC with the release of the iPhone 5S, most notably to support their fingerprint technology, Touch ID. SEP is designed as a security circuit configured to perform secure services for the rest of the SOC, with no direct access from the main processor. In fact, the secure enclave processor runs its own fully functional operating system – dubbed SEPOS – with its own kernel, drivers, services, and applications. This isolated hardware design prevents an attacker from easily recovering sensitive data (such as fingerprint information and cryptographic keys) from an otherwise fully compromised device. In this paper, we aim to shed some light the secure enclave processor and delve into the internals of SEPOS. In particular, we look at the SEPOS architecture itself and detail how the iOS kernel and the SEP exchange data using an elaborate mailbox mechanism. Furthermore, we show how this data is handled by SEPOS and relayed to its services and applications in order to allow iOS to interact with endpoints exposed by the SEP. Finally, we evaluate the SEP attack surface, its attack robustness, and highlight potential attack vectors.

Keywords: iOS, SEPOS, Secure Enclave Processor, exploitation

1 Introduction

Since its introduction, the Secure Enclave Processor (SEP) has become an important component in providing security to Apple devices. Notably, SEP is used to support services that process highly sensitive data such as Touch ID and Apple Pay. The Secure Enclave Processor runs its own operating system – dubbed SEPOS – and fully operates in its own protected memory space in physical memory. Thus, an attacker who has gained full control of iOS cannot easily gain access to SEPOS and compromise its data.

Although designed to assist iOS and perform operations on its behalf, SEPOS is a very different operating system. In fact, SEPOS is built around the L4 microkernel and features an array of custom written drivers, services and applications. Since iOS has no direct access to SEPOS, it communicates through a mechanism known as the secure mailbox. Essentially, the secure mailbox is implemented as a shared memory region between the application processor and the secure enclave processor, where messages are passed using an interrupt based delivery system. Thus, by monitoring certain interrupts, either side can be notified about message requests and replies.

While SEP has been around since the iPhone 5S, little information exists on its inner workings, such as how SEPOS handles requests made from iOS. In fact, no part of SEPOS is documented by Apple nor by any third party, leaving the public with many unanswered questions. What services are exposed by the SEP? How are these services accessed? How resilient is SEP against attacks?

In this paper we attempt to shed some light on the secure enclave processor and answer many of these unanswered questions. In particular, we show how iOS interacts with SEPOS and how drivers leverage built-in frameworks in order to talk to specific SEP services and applications. We also provide an overview of the SEPOS architecture and detail many of its core components, such as the bootstrap server. Given a fundamental understanding of how SEPOS works, we then provide an assessment of its attack surface and demonstrate how an attacker may potentially compromise SEP, given a vulnerability in one of its exposed services. Notably, we show that SEPOS lacks many exploit mitigations found in most contemporary operating systems such as address space layout randomization (ASLR), randomized stack cookies, and heap hardening.

The rest of this paper is organized as follows. In Section 2, we examine the interface used by iOS to communicate with SEPOS, and show how drivers leverage this to communicate with SEP applications. In Section 3, we look at the SEPOS architecture and detail its many components, including the operating system kernel, drivers, services, and applications. Furthermore, in Section 4, we evaluate the attack the attack surface of SEPOS and present an assessment of its attack robustness (exploitability). Finally, in Section 5 we provide a conclusion of the paper.

2 Communication

In order to communicate with the SEP, an elaborate mailbox mechanism is used to send messages between the Application Processor (AP) and SEP processor. In this Section, we describe how the secure mailbox is implemented and show how drivers in iOS use it to communicate with services provided by the SEP.

2.1 Secure Mailbox

Whenever a driver in iOS wants to communicate with a service in SEPOS, it sends a message to the secure mailbox. The secure mailbox is essentially a region of shared memory visible to both the AP and the SEP processor, where both message requests and responses are written. The mailbox itself is implemented by the `AppleA7IOP` kernel extension, and is initialized in its `start()` function. Specifically, this function attempts to map the I/O registers of the SEP device by calling `mapDeviceMemoryWithIndex(0,0)` as shown in Listing 1.

Notably, `mapDeviceMemoryWithIndex()` is used to map a physical range of a device, given an index into the array of ranges assigned to the device. The function then returns an instance of `IOMemoryMap` that describes the mapping. The ranges assigned to the SEP device on a given Apple device can be observed

```
bool
AppleA7IOP::start( IOService *provider )
{
    bool success = IOService::start( provider );

    if ( success )
    {
        _akfProvider = OSDynamicCast( AppleARMIODevice, provider );

        _akfRegisterMap = _akfProvider->mapDeviceMemoryWithIndex( 0, 0 );

        ...
    }

    return success;
}
```

Listing 1: AppleA7IOP mapping SEP I/O registers

by leveraging the `ioreg` utility. For instance, the output in Listing 2 is from an iPhone 6S, running a flavor of iOS 9.

```
iPhone:~ root# ioreg -w 70 -d 1 -rxin sep
+-o sep@DA00000  <class IORegistryEntry:IOService:AppleARMIODevice, i$
    {
      "IODeviceMemory" = (({"address"=0x20da00000,"length"=0x10000}))
    }
```

Listing 2: `ioreg` output of device memory assigned to SEP

As can be seen in Listing 2, a single physical memory range is assigned to the SEP device of an iPhone 6S under the `IODeviceMemory` key. This is the SEP device I/O registers, also known as the SEP configuration space. At the time of writing (iOS 9) there are two different versions of the SEP device I/O registers, abstracted by the classes `AppleA7IOPV1` (iPhone 6 and earlier) and `AppleA7IOPV2` (iPhone 6S). These classes, in addition to the `AppleA7IOP` base class, provide all the functions necessary to operate on the secure mailbox. In particular, both version classes provide functions for posting messages to the inbox and reading messages from the outbox. These operations simply write

or read bytes at specific offsets in the I/O register mapping, depending on the version used. For instance, the function for posting a message to the SEP inbox for `AppleA7IOPV2` is as shown in Listing 3.

```
AppleA7IOPV2::_inbox(unsigned long long)
    LDR             X8, [X0,#0x98] ; virtual address of SEP registers
    STR             X1, [X8,#0x4010] ; post message to inbox
    RET
```

Listing 3: Function to post a message to the SEP inbox

The function for reading messages from the SEP outbox looks very similar, except reads from a different offset into the register mapping (Listing 4).

```
AppleA7IOPV2::_outbox(void)
    LDR             X8, [X0,#0x98] ; virtual address of SEP registers
    LDR             X0, [X8,#0x4038] ; read message from outbox
    RET
```

Listing 4: Function to read a message from the SEP outbox

The previously listed functions both read and write from specific offsets in the SEP I/O register mapping. The I/O registers are also used to perform other operations such as firmware mapping, set status flags, generate non-maskable interrupts, etc. An outline of which offsets into the I/O register mapping are used to perform what mailbox operations for `AppleA7IOPV2` devices is shown in Table 1, with associated status flags listed in Table 2.

The functions that operate on the I/O registers are not meant to be used directly. Instead, the `AppleA7IOP` class exports a set of general functions that internally leverage the functionality provided by `AppleA7IOPV1` and `AppleA7IOPV2`. In particular, `AppleA7IOP::getMailbox()` and `AppleA7IOP::postMailbox()` provide a more general way for reading and writing messages to a mailbox. These functions also implement additional mechanisms to check if the mailbox is empty or full, depending on whether a read or a write is attempted. `AppleA7IOP` also provides functions for waking up the SEP, enabling mailbox interrupts, dumping the mailbox log, and registering doorbell (callback) handlers. The latter allows a driver to register a callback for whenever a message is successfully sent or received.

Offset	Type	Description	Notes
0x4000	UInt32	Disable mailbox interrupt	0x1: inbox; 0x1000: outbox
0x4004	UInt32	Enable mailbox interrupt	0x1: inbox; 0x1000: outbox
0x4008	UInt32	Inbox status bits	See Table 2
0x4010	UInt64	Inbox value	Request message
0x4020	UInt32	Outbox status bits	See Table 2
0x4038	UInt64	Outbox value	Reply message

Table 1. AppleA7IOPV2 mailbox I/O register format (iPhone 6s)

Flag	Value	Description
MBOX_CTRL_enable	0x00001	Enable mailbox
MBOX_CTRL_full	0x10000	The mailbox is full
MBOX_CTRL_empty	0x20000	The mailbox is empty
MBOX_CTRL_ovfl	0x40000	Overflow
MBOX_CTRL_udfl	0x80000	Underflow

Table 2. Mailbox status flags

2.2 Message Interrupts

In order for the SEP to be informed of any new messages, an interrupt is triggered whenever the AP attempts to write to the SEP inbox. This interrupt is not generated by the AP itself as that could possibly open up to denial-of-service attacks if the SEP could be interrupted for long periods of time. Instead, a hardware filter is used to filter any incoming read or write attempts and generate an interrupt for the SEP if necessary. This filter, described as the "SEP Filter" in Patent US8832465 [2], is only used to filter incoming read or writes (hence may also be used to protect areas of physical memory such as that belonging to the Secure Enclave). Thus, the components of the SEP may have full access to the other components of the SOC including the AP kernel memory (except for the portion of memory assigned as the AP trust zone).

Once the SEP has received and processed a message, it writes a response back to the outbox. In order to let the AP know that a response is ready, it signals an interrupt to the CPU processor by transmitting an interrupt message to the interrupt controller. This interrupt is caught by a handler registered by AppleA7IOP upon initialization. In its start() routine, after the SEP I/O registers have been mapped, AppleA7IOP calls filterInterruptEventSource() to create an interrupt event source filter to receive interrupt callbacks. Specifically, handlers for both the inbox (AppleA7IOP::_inboxHandler) and the outbox (AppleA7IOP::_outboxHandler) are registered. If the SEP writes a message to the outbox and signals an interrupt, AppleA7IOP::_outboxHandler() is invoked. This function looks as shown in Listing 5.

Note from the Listing 5 that the outbox handler attempts to call the registered doorbell handler (akfAction). This handler is set by the class function

```
kern_return_t
AppleA7IOP::_outboxHandler(IOInterruptEventSource * src)
{
    A7MessageLog::logMsg( 0, 1 ); // log interrupt

    if ( src->event )
    {
        thread_wakeup_prim( src->event, false, THREAD_AWAKENED );
    }

    src->akfAction( src->akfObject, src->akfArg, 0 );

    return KERN_SUCCESS;
}
```

Listing 5: AppleA7IOP outbox handler

`setDoorbellAction()` and allows a different driver to handle the response. In the next Section, we look at how the SEP Manager leverages this mechanism to handle messages, and how this kernel extension implements support for interacting with SEP endpoints from iOS.

2.3 SEP Manager

The SEP Manager (implemented by the `AppleSEPManager` kernel extension) is responsible for managing all SEP endpoints in iOS and provides an interface for drivers to register new endpoints and communicate with SEP services. It leverages the functionality provided by the previously discussed `AppleA7IOP` extension in order to send and receive messages through the secure mailbox, and also provides a mechanism for buffering and passing those messages onward to their end destination.

In its `start()` routine, `AppleSEPManager` initializes several endpoints that are used by various facilities in iOS. An endpoint is represented by an object that inherits the `AppleSEPEndpoint` class and provides functions for both sending and receiving messages (wrappers for the mechanisms implemented by `AppleA7IOP`). It is commonly created by calling the function `AppleSEPEndpoint::withOptions()`. The prototype of this function is shown in Listing 6.

Notably, `withOptions()` of the `AppleSEPEndpoint` class takes a pointer to a callback routine to invoke whenever a response is waiting, as well as an integer value that represents the endpoint index (used to uniquely identify the endpoint). The callback function takes three arguments, the object of the specific endpoint, an optional parameter, and a pointer to the received message. Although the message itself is always 8 bytes long, the format may be different depending

```
AppleSEPEndpoint *
AppleSEPEndpoint::withOptions
(
    AppleSEPManager *provider,          // endpoint manager
    OSObject *owner,                    // endpoint owner (receives events)
    IOSlaveEndpoint::Action action,     // action invoked when work is available
    void *refcon,                       // parameter to action routine
    UInt32 handle                       // endpoint handle/index
)
```

Listing 6: `AppleSEPEndpoint::withOptions()` prototype

on the endpoint. However, the message format will in many cases look similar to that shown in Listing 7. Note that the first byte of a message is always the destination endpoint index.

```
struct
{
    uint8_t    endpoint;
    uint8_t    tag;
    uint8_t    opcode;
    uint8_t    param;
    uint32_t   data;
} sep_msg;
```

Listing 7: AP/SEP message format

Once an endpoint instance has been created, `AppleSEPManager` stores it internally in a table of registered endpoints. An endpoint of a specified index can then be retrieved by calling `AppleSEPEndpoint::endpointWithIndex()`. There are many endpoints registered in a given system, both internally by `AppleSEPManager` and externally by specific drivers such as `AppleSEPKeyStore`. An outline of these endpoints is shown in Appendix A.

2.4 Message Handling

In order to handle and buffer received mailbox messages, `AppleSEPManager` also registers a doorbell handler by invoking `AppleA7IOP::setDoorbellAction()`.

As previously mentioned in Section 2.2, the doorbell handler is invoked whenever SEP triggers an interrupt to notify the AP that a message was placed in the outbox. Specifically, `AppleSEPManager` registers the callback function `AppleSEPManager::_doorbellAction()`, shown in Listing 8.

```
void
AppleSEPManager::_doorbellAction( void *, int )
{
    AppleSEPMessage msg;
    AppleSEPEndpoint *ep;
    AppleA7IOP *iop;

    iop = OSDynamicCast( AppleA7IOP, this->iop );

    for ( status = iop->getMailbox( 0, &msg, false );
          status == kIOReturnSuccess;
          status = iop->getMailbox( 0, &msg, false ) )
    {
        ep = NULL;

        if ( msg.endpoint < 32 )
        {
            ep = AppleSEPEndpoint::endpointWithIndex( msg.endpoint );
        }
        else if ( msg.endpoint == 255 )
        {
            ep = this->boot_ep;
        }

        if ( ep )
        {
            status = ep->queueReceivedMessage( &msg );
        }
    }
}
```

Listing 8: Doorbell handler

This function first attempts to call `AppleA7IOP::getMailbox()` to retrieve a message from the outbox. If successful, it then extracts the endpoint number from the message and calls `AppleSEPEndpoint::endpointWithIndex(uint)` to retrieve the associated endpoint object. Finally, it calls `queueReceivedMessage()` in order to queue the message onto the endpoint's message stack and then signals the endpoint by invoking the `IOSlaveEndpoint::signalDoorbell()` function.

This in turn causes the endpoint's `AppleSEPEndpoint::checkForWork()` workloop function to be triggered. Notably, the `checkForWork()` function first calls `AppleSEPEndpoint::receiveMessage()` in order to pull the message from the endpoint's message queue. It then calls the `action` callback routine registered upon endpoint initialization in order to handle the message.

2.5 Control Endpoint

The control endpoint is used to issue control requests to the SEP and is also used to support endpoint management. In particular, the control endpoint is used to assign out-of-line buffers to an endpoint (described in Section 2.6), generate, read, and/or invalidate nonces, and send raw SEP console commands (not available in production devices). The control endpoint handles several commands, as shown in Table 3.

Opcode	Name	Description
0	`NOP`	Used to wake up the SEP
1	`ACK`	Acknowledgment from SEP
2	`SET_OOL_IN_ADDR`	Request out-of-line buffer address
3	`SET_OOL_OUT_ADDR`	Reply out-of-line buffer address
4	`SET_OOL_IN_SIZE`	Size of request buffer
5	`SET_OOL_OUT_SIZE`	Size of reply buffer
10	`TTYIN`	Write to SEP console
12	`SLEEP`	Sleep the SEP
18	`SLEEP_WAKE`	Sleep the SEP and enable interrupt
19	`NOTIFY`	Notify console driver thread
20	`SECMODE_REQUEST`	Request the security mode
21	`NONCE_GENERATE`	Generate a nonce
22	`NONCE_READ`	Read the nonce value
23	`NONCE_INVALIDATE`	Invalidate the nonce
24	`SELF_TEST`	Counter test

Table 3. Control endpoint commands

iOS provides a SEP management utility (`/usr/libexec/seputil`) that primarily allows a user to interact with the control endpoint. This is made possible by implementing several methods for control endpoint requests within `AppleSEPUserClient`, provided by the SEP Manager. The user client also implements a method to enable a user to dump the AP/SEP message log (as populated by `AppleSEPManager`). An excerpt of its output is shown in Listing 9.

Notably, the log provides information on all recently sent (`TX`) and received (`RX`) messages, including their data. The log also provides a time stamp for when the message was observed.

```
iPhone:~ root# /usr/libexec/seputil --log
Kernel message log has 128 entries
530705645112: TX message ept 0, tag 8, opcode 4, param c, data 4000
530705646396: RX interrupt
530705646464: RX message ept 0, tag 8, opcode 1, param 0, data 4000
530705647256: TX message ept 0, tag 8, opcode 2, param c, data 81cf5c
530705652516: RX interrupt
530705652600: RX message ept 0, tag 8, opcode 1, param 0, data 81cf5c
530705654852: TX message ept 0, tag 8, opcode 5, param c, data 4000
530705655952: RX interrupt
530705656004: RX message ept 0, tag 8, opcode 1, param 0, data 4000
530705656760: TX message ept 0, tag 8, opcode 3, param c, data 81f360
530705658376: RX interrupt
530705658460: RX message ept 0, tag 8, opcode 1, param 0, data 81f360
530705659248: TX message ept c, tag 1, opcode 8, param 0, data 0
530705669540: RX interrupt
530705669624: RX message ept c, tag 1, opcode 8, param 0, data 0
...
```

Listing 9: AP/SEP message log

2.6 Out-of-Line Data Transfers

In Section 2.2, we described how the application processor communicates with the secure enclave processor using an interrupt-driven mailbox mechanism. The use of interrupts can also be observed in Listing 9 where interrupts are received whenever a response is waiting in the mailbox. Since the mailbox only can pass at most 7 bytes of data at a time (excluding a byte for the destination endpoint), transferring large amounts of data can be slow and degrade overall system performance. As such, out-of-line buffers may optionally be assigned to endpoints and used instead for transferring data between the AP and the SEP.

Since the SEP can access all parts of physical memory, the AP sets up a shared memory buffer by simply allocating a portion of physical memory that also is visible to the SEP. In order to let the SEP know about the shared memory buffer and assign it to an endpoint, two messages are sent to the control endpoint. In the first message, a `SET_OOL_IN_SIZE` (or `SET_OOL_OUT_SIZE`) request is sent with param of the message header set to the target endpoint and value indicating the byte size of the buffer. In the second message, a `SET_OOL_IN_ADDR` (or `SET_OOL_OUT_ADDR`) request is sent, again with param specifying the target endpoint, but value defining the physical address of the shared buffer, right shifted by the page size. The alert reader may already have noticed that Listing 9 actually shows an out-of-line buffer registration process for an endpoint. In this particular case, both the request and reply buffers are 0x4000 bytes in size,

while the request buffer is located at 0x81cf5c000 and the reply buffer is located at 0x81f360000.

An endpoint does not manually need to send messages to the control endpoint in order to assign an out-of-line request and reply buffers to an endpoint. Instead, the wrapper functions `AppleSEPEndpoint::setSendOOLBuffer(IOSlaveMemory *)` and `AppleSEPEndpoint::setReceiveOOLBuffer(IOSlaveMemory *)` are commonly used. Note that both these functions take the `IOSlaveMemory` object returned by `allocateVisibleMemory()` as their argument.

2.7 Drivers Using SEP

Since SEP was introduced with iPhone 5S, several iOS drivers have had a large part of their functionality moved into the SEP. One notable example of a driver whose core functionality was moved into the SEP is the `AppleKeyStore` kernel extension (renamed to `AppleSEPKeyStore`). Notably, `AppleKeyStore` is responsible for managing the user keybag where the wrapped class keys used in normal operation of the device are stored. If a service or daemon (typically `keybagd`) requests a specific operation to be performed such as loading a new keybag, `AppleKeyStore` will forward the request to the SEP. An example of this is shown in Listing 10, where a `deliver_msg` callback is provided to the `__load_keybag` function.

```
IOReturn
AppleKeyStore::load_keybag( int tag, void *data, int size, int *handle )
{
    unsigned int     res;

    res = _load_keybag(
              AppleKeyStore::deliver_msg,     // SEP message handler
              tag,                            // session tag
              data,                           // keybag data
              size,                           // keybag length
              handle);                        // returned handle

    ...
}
```

Listing 10: Messages sent between `AppleSEPKeyStore` and SEP

The callback invoked by `__load_keybag` is a generic function used by the `AppleSEPKeyStore` kernel extension to send messages to the SEP. It is executed as a gated function in order to ensure to be executed in a single-threaded

manner. As endpoints in `AppleSEPManager` may be registered on demand (typically whenever the first driver request is made), the SEP message function first attempts to resolve the endpoint used to communicate with the corresponding application running in SEP. This is performed by first calling the `AppleSEPKeyStore::sep_endpoint()` function. Essentially, this function checks if it already has resolved the pointer to an `AppleSEPEndpoint` instance, and if not, requests `AppleSEPManager` to create a new endpoint for the provided handle value using `AppleSEPManager::endpointForHandle()`. The prototype of this function is shown in Listing 11. Note that `endpointForHandle()` is essentially a wrapper for `AppleSEPEndpoint::withOptions()` (described in Section 2.3).

```
AppleSEPEndpoint *
AppleSEPManager::endpointForHandle
(
    AppleSEPManager *provider          // endpoint provider
    OSObject *owner,                   // endpoint owner (receives events)
    IOSlaveEndpoint::Action action,    // action invoked when work is available
    UInt32 handle,                     // endpoint handle (index)
    void *refcon                       // parameter to action routine
)
```

Listing 11: `AppleSEPManager::endpointForHandle()` prototype

Once `AppleSEPKeyStore` has successfully registered a new endpoint with `AppleSEPManager`, it proceeds to assign out-of-line request and reply buffers for the data to be exchanged between AP and SEP. This is performed by leveraging the method described in Section 2.6. In particular, `AppleSEPManager` first allocates 0x4000 bytes of visible memory using `allocateVisibleMemory()` in order to obtain an `IOSlaveMemory` object. This object is then passed to the endpoint's `setSendOOLBuffer()` method in order to assign the send buffer parameters to the endpoint, on both the AP and SEP side (via the control endpoint). Similarly, the receive buffer is assigned by leveraging the same method, but instead using `setReceiveOOLBuffer()`.

Once `AppleSEPKeyStore::sep_endpoint()` returns with an endpoint object, the SEP message function retrieves a command gate from the command pool assigned to the endpoint and then starts the process of marshaling the request data into the shared memory buffer. Notably, `AppleSEPKeyStore` maintains a table of function pointers that describes how data should be copied into the shared memory region. Once the data has been copied, the message function then invokes the endpoint's `sendMessage()` function in order to notify SEP that data has been placed in the request buffer. In particular, the message specifies

the request number (opcode) as well as an integer value defining the location into the shared memory buffer where the data is to be read from (stored as part of the message data). An example of this is shown in the log output of Listing 12.

```
1057653057604: TX message ept 7, tag 19, opcode b4, param 0, data 140000
1057653062208: RX interrupt
1057653062296: RX message ept 7, tag 99, opcode b4, param 0, data 300000
1057653296668: TX message ept 7, tag 19, opcode b5, param 0, data 140000
1057653298080: RX interrupt
1057653298164: RX message ept 7, tag 99, opcode b5, param 0, data 300000
```

Listing 12: Messages sent between `AppleSEPKeyStore` and SEP

Once data has been sent and written to the SEP mailbox, the message handler enters a sleep with `IOCommandGate::commandSleep()` on the command gate it previously obtained. Since `AppleSEPKeyStore` leverages a command gate mechanism when sending messages, the action handler doesn't itself handle the responses. Instead the handler will check the tag field of the message to see if the highest bit is set (0x80). If this is the case, then the message is considered to be a response to a previously sent message and the handler invokes `IOCommandGate::commandWakeup()` to wake up the thread currently holding the command gate (the thread that invoked the SEP message handler).

Once the message handler thread has been woken up as a result of a message being received, it copies out the data from the shared memory reply buffer by leveraging the information held in the response message. Again, the data in the reply message provides an integer value that defines the location of the actual reply data in the shared buffer.

Note that we in this Section have focused on how data is communicated to SEP from iOS (AP). In Section 3 we provide a more in-depth look of the SEPOS architecture, how SEP handles mailbox messages, and how they ultimately reach their final destination.

3 SEPOS

At the heart of the secure enclave processor runs SEPOS, an L4 microkernel based operating system. In this Section, we look at the architecture of SEPOS and detail its various components.

3.1 L4 (Kernel)

SEPOS is an entirely self-contained operating system with its own kernel, drivers, services, and applications. Although completely different from iOS, SEPOS was not written entirely from scratch. It is based on a microkernel that has gained significant popularity in embedded systems in recent years, namely L4. Although the L4 microkernel has matured and seen notable improvements over the last 20 years [3], it only provides the bare bones of a fully functional operating system. As such, any vendor who wishes to adopt the L4 microkernel is required to make several implementation choices on their own.

The L4 microkernel was first introduced in 1993 by Jochen Liedtke [4]. It was developed to address the poor performance of earlier microkernels, and improved IPC performance over its predecessor L3 by a factor 10-20 faster. This was considered a milestone as applications running on microkernel based platforms greatly rely on IPC mechanisms for performing most everyday tasks. L4 has since become platform independent, and seen numerous implementations and variants. Some of these variants (considered predecessors to SEPOS) are shown in Table 4.

Id	Subid	Kernel	Supplier
0	1	L4/486	GMD
0	2	L4/Pentium	IBM
0	3	L4/x86	UKa
1	1	L4/MIPS	UNSW
2	1	L4/ALPHA	TUD, UNSW
3	1	Fiasco	TUD
4	1	L4Ka::Hazelnut	UKa
4	2	L4Ka::Pistachio	UKa, UNSW, NICTA
4	3	L4Ka::Strawberry	UKa
5	1	NICTA::Pistachio-embedded	NICTA
6	1	SEPOS	APPL

Table 4. L4 variants

The L4 microkernel was designed to be both minimal and general. However, it wasn't until *L4Ka::Pistachio* that it first became platform independent. This paved way for more widespread deployment, and soon afterwards a modified version specifically designed for resource constrained embedded system was

released. *NICTA::Pistachio-embedded* both reduced kernel complexity and memory footprint, and was deployed at wide-scale when Qualcomm adopted it for their CDMA chipsets. NICTA::Pistachio-embedded was also later used in Darbat, an L4 experimental port of Darwin. Darbat was developed to study the characteristics of a large-scale based microkernel system, and included a port of IOKit, a modified libc to communicate with the Darbat server, and XNU.

Although it may seem likely that SEPOS is based on Darbat, the two implementations share very little in common. It was never a goal for SEPOS to emulate XNU or support any of its subsystems such as Mach or IOKit. In fact, most SEPOS components are written entirely from scratch, such as the bootstrap server, driver and services host, and many applications. An outline of the SEPOS architecture and its components is shown in Figure 1.

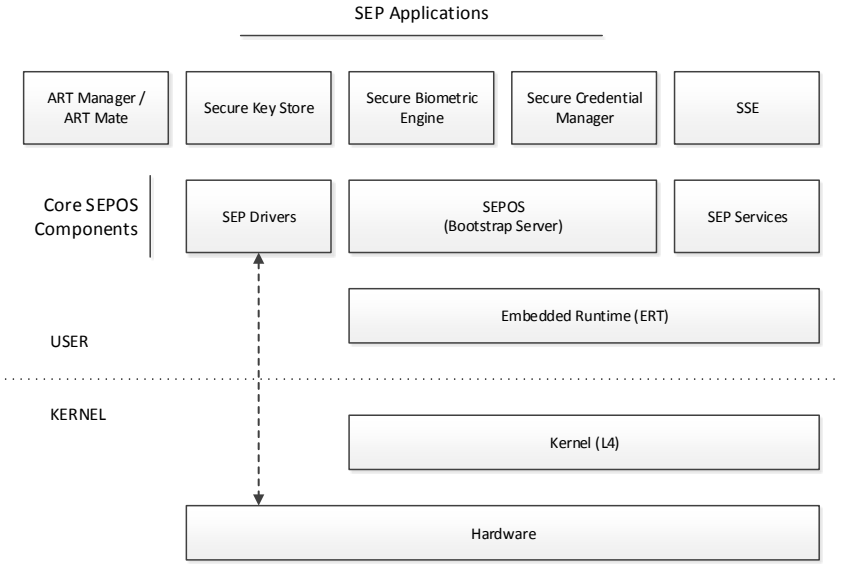

Fig. 1. SEPOS architecture

The L4 kernel in SEPOS is like Darbat based on Pistachio-embedded. It features only a minimal set of interfaces, as the major part of the operating system is implemented in user-mode. As such, only around 20 system calls exist in the SEPOS kernel, some of which have been specifically introduced by Apple. A list of these system calls is shown in Table 5.

In SEPOS, the L4 kernel is responsible for initializing the machine state to a point where it becomes usable. This includes steps such as initializing the kernel page table, setting up the kernel interface page (KIP), configuring interrupts on

Num	Name	Description	Privileged
0x00	L4_Ipc	Set up ipc between two threads	
0x04	L4_ThreadSwitch	Yield execution to thread	
0x08	L4_ThreadControl	Create or delete threads	✓
0x0c	L4_ExchangeRegisters	Exchange registers	
0x10	L4_Schedule	Set thread scheduling information	
0x14	L4_MapControl	Map or free virtual memory	✓
0x18	L4_SpaceControl	Create a new address space	✓
0x1c	L4_ProcessorControl	Sets processor attributes	
0x20	L4_CacheControl	Cache flushing	
0x24	L4_IpcControl	Adjust ipc access	✓
0x28	L4_InterruptControl	Enable or disable an interrupt	✓
0x2c	L4_GetTimebase	Gets the system time (?)	
0x30	L4_SetTimeout	Set timeout for ipc sessions	
0x34	L4_SharedMappingControl	Set up a shared mapping	✓
0x38	L4_SleepKernel	?	
0x3c	L4_PowerControl	?	
0x40	L4_KernelInterface	Get information about kernel	

Table 5. SEPOS L4 system calls

the hardware, starting the timer, initializing the mapping database, and starting the kernel scheduler. The kernel scheduler starts all interrupt and kernel threads, as well as the initial process known as the root task. In L4-based operating systems, the root task is the most privileged process and implements many of the core subsystems needed by a fully functional operating system. We provide a more thorough description of the root task and the subsystems it implements in Section 3.2.

Since many of the L4 system calls are considered to be quite powerful (e.g `L4_MapControl` and `L4_SpaceControl`), they are only made available to the most privileged task(s), i.e. the root task. These system calls specifically check the caller's space address and compares it to those that are considered privileged. In some L4 distributions such as Darbat, more than one task may be considered privileged. In SEPOS, only the root task is allowed to invoke privileged system calls. An example of a privileged system call is shown in Listing 13. Note here that `is_privileged_space()` is basically a wrapper for `is_roottask_space()`, which compares the provided space address to that of the root task.

The system calls provided by the L4 kernel are never called directly by user space applications. Instead, a low-level library (ERT) is provided to interface with the kernel and implement basic mechanisms. The ERT (believed to be an acronym for Embedded Runtime) divides its functions into public and private functions, and prefixes these functions `ert` and `ertp` respectively. Examples of private functions include `ertp_map_page()` and `ertp_unmap_page()` (both wrappers for `L4_MapControl`), whereas public functions include more commonly used routines such as `ert_rpc_call()` (leverages `L4_Ipc` internally).

```
SYS_SPACE_CONTROL ( threadid_t space_tid, word_t control,
                    fpage_t kip_area, fpage_t utcb_area )
{
    TRACEPOINT (SYSCALL_SPACE_CONTROL,
        printf("SYS_SPACE_CONTROL: space=%t, control=%p, kip_area=%p, "
        "utcb_area=%p\n",  TID (space_tid),
        control, kip_area.raw, utcb_area.raw));

    // Check privilege
    if (EXPECT_FALSE (! is_privileged_space(get_current_space())))
    {
        get_current_tcb ()->set_error_code (ENO_PRIVILEGE);
        return_space_control(0, 0);
    }

    ...
}
```

Listing 13: Privilege check in `L4_SpaceControl` system call

3.2 SEPOS (Root Task)

In SEPOS, the root task is simply called SEPOS. It is responsible for starting all remaining applications that run in SEPOS, as well as maintaining contextual information about every running task. This includes information about a task's virtual address space, privilege level, running threads, and so on. The root task also invokes the bootstrap server, an IPC-based service that provides applications with the services they need in order to perform privileged tasks such as mapping virtual memory or starting new threads.

Application Startup When the root task first starts up, it attempts to find two tables in memory - the root arguments and the application list. These tables are found by first retrieving the pointer to the image macho header and then subtracting 0x1000 and 0xEC8 bytes respectively. This references an area immediately preceding the root task in the SEP firmware image, mapped in physical memory. The root arguments are the boot arguments loaded by the SEP ROM and also contain a 32-bit CRC value (at offset 0x2c) for all the binaries embedded by the firmware. An output of the root arguments as embedded by the SEP firmware on an iPhone 6S (running iOS 9) is shown in Listing 14. Note that some fields are populated at runtime (such as the pointer to the SEP ROM arguments), hence will appear as zero in the on-disk firmware image.

```
8:4000h: 01 00 11 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
8:4010h: 89 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
8:4020h: 00 00 00 00 00 00 00 00 00 00 00 00 F9 8C EA 0E  ................
8:4030h: 00 00 00 00 00 00 00 00 46 69 72 6D 77 61 72 65  ........Firmware
8:4040h: 20 6D 61 67 69 63 20 73 74 72 69 6E 67 0A 57 69   magic string.Wi
8:4050h: 74 68 6F 75 74 20 77 68 69 63 68 2C 20 77 68 61  thout which, wha
8:4060h: 74 20 61 72 65 20 74 68 65 73 65 20 62 69 74 73  t are these bits
8:4070h: 3F 0A 53 45 50 20 64 65 6E 69 65 64 2E 20 20 20  ?.SEP denied.
```

Listing 14: Root arguments

The application list contains information about all applications embedded by the SEP firmware and is leveraged by the root task to initialize and start all remaining applications. Notably, the application list contains an array of `app_info` structures, where each entry holds the information necessary to load the app such as its physical address (firmware offset), preferred virtual base address, size, name, and entry point. The format of the `app_info` structure is shown in Listing 15.

```
struct app_info
{
    uint64_t     physical_addr;
    uint32_t     virtual_base;
    uint32_t     size;
    uint32_t     entry;
    uint8_t      name[12];
    uint8_t      hash[16];
}
```

Listing 15: Application information format

The application list typically contains several applications, some of which may be specific to a particular device or group of devices. The application list embedded by the SEP firmware of an iPhone 6S (running iOS 9) is shown in the output of Listing 16. Note that the high 32-bits of the physical addresses shown are zero as these are set at runtime (typically to 0x88 - the SEP physical region).

```
8:4130h: 00 00 00 00 00 00 00 00 00 40 08 00 00 00 00 00  .........@......
8:4140h: 00 70 00 00 00 A0 01 00 24 AB 00 00 53 45 50 4F  .p......$...SEPO
8:4150h: 53 20 20 20 20 20 20 20 31 F0 D3 DE EF 0E 38 07  S       1.....8.
8:4160h: AB 21 DB DA DA 8D 08 EF 00 E0 09 00 00 00 00 00  .!..............
8:4170h: 00 80 00 00 00 80 02 00 2C CD 00 00 53 45 50 44  ........,...SEPD
8:4180h: 72 69 76 65 72 73 20 20 62 91 03 23 3A 6B 3C 90  rivers  b..#:k<.
8:4190h: B5 49 97 F4 E0 85 DF 9D 00 60 0C 00 00 00 00 00  .I.......`......
8:41A0h: 00 80 00 00 00 80 01 00 D0 3C 01 00 73 65 70 53  .........<..sepS
8:41B0h: 65 72 76 69 63 65 73 20 3E C2 98 F0 9E 24 37 A5  ervices >....$7.
8:41C0h: A8 5D 55 C5 39 37 F0 E7 00 E0 0D 00 00 00 00 00  .]U.97..........
8:41D0h: 00 80 00 00 00 C0 00 00 30 EE 00 00 41 52 54 4D  ........0...ARTM
8:41E0h: 61 74 65 20 20 20 20 20 E4 EE 3A 22 62 84 3B B1  ate     ..:"b.;.
8:41F0h: B5 E8 71 3A 62 7D F0 99 00 A0 0E 00 00 00 00 00  ..q:b}..........
8:4200h: 00 10 00 00 00 70 07 00 D8 43 01 00 73 6B 73 20  .....p...C..sks
8:4210h: 20 20 20 20 20 20 20 20 28 9A D0 5C 62 82 3C 33          (..\b.<3
8:4220h: AB 91 B2 D3 C8 64 8B 61 00 10 16 00 00 00 00 00  .....d.a........
8:4230h: 00 10 00 00 00 90 02 00 50 0F 01 00 73 63 72 64  ........P...scrd
8:4240h: 20 20 20 20 20 20 20 20 31 FB CB F0 F9 DD 39 8A          1.....9.
8:4250h: AF 95 52 5C A1 F0 78 F2 00 A0 18 00 00 00 00 00  ..R\..x.........
8:4260h: 00 10 00 00 00 E0 01 00 88 73 00 00 73 73 65 20  .........s..sse
8:4270h: 20 20 20 20 20 20 20 20 99 5F E2 24 92 28 3C E4          ._.$.(<.
8:4280h: 90 43 99 78 39 3D 1F C6 00 80 1A 00 00 00 00 00  .C.x9=..........
8:4290h: 00 10 00 00 00 10 2E 00 40 66 06 00 73 62 69 6F  ........@f..sbio
8:42A0h: 2D 73 64 20 20 20 20 20 35 DA 9B 6D 04 28 39 5B  -sd     5..m.(9[
8:42B0h: 89 90 E2 A5 0A 9C 4B 42 00 00 00 00 00 00 00 00  ......KB........
```

Listing 16: Application list

For each application the root task needs to load, it calls `proc_create()` in order to initialize a process context structure and create a new address space (using `L4_SpaceControl`) for the new task. The root task maintains a list of all running processes using a fixed sized array of 16 entries, hence currently allows at most 16 applications to be running simultaneously. Once an address space has been created, the root task then maps the header segment of the embedded Mach-O binary and subsequently calls `macho2vm()` to map each segment into the task address space. Finally, the root task calls `thread_create()` in order to create a new thread and begins execution at the entry point provided in the application list (Figure 2).

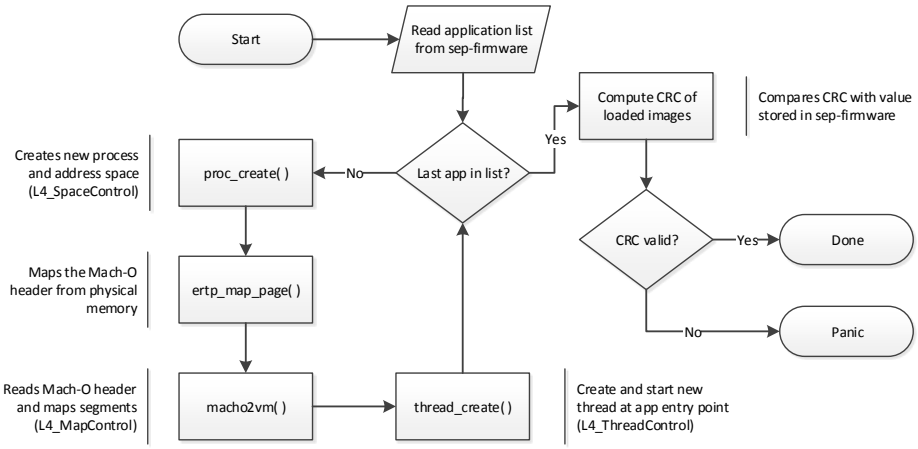

Fig. 2. Application startup

Bootstrap Server Once all applications have been loaded, the root task starts the bootstrap server. The bootstrap server provides an interface to a set of functions that applications may call in order to perform operations that typically would require use of privileged system calls such as `L4_MapControl` or `L4_ThreadControl`. Essentially, these functions make up the SEPOS API, and enable applications to interact with SEPOS as a whole.

The methods exported by the bootstrap server are organized into separate tables called interfaces. An overview of both interfaces and their supported methods is provided in Appendix B. In particular, whenever an IPC request is made to the bootstrap server by an application, both an interface and a method id are provided. The bootstrap server then iterates the table of exported functions until it finds a method that matches both the interface and method id. Note that in the event the bootstrap server cannot match against either the interface or method id, a catch all routine (method id: -1) is invoked instead. This routine simply panics the system with a message "`unhandled system message %u from %s`", where `%u` is the supplied message id and `%s` is the name of the sender process/thread.

One of the most commonly used interfaces provided by the bootstrap server is the object interface. Objects enable SEP applications to create virtual memory mappings, by either specifying the physical memory range to map (using `sepos_object_create_phys()`) or by using a reserved region for anonymous physical mappings (`sepos_object_create()`). Once created, objects may subsequently be mapped into a process address space with the desired memory protection by leveraging the `sepos_object_map()` method. A list of the supported object interface methods is shown in Appendix B.2

In order to restrict object access, each object has an associated access control list. The access control list is a linked list that specifies the maximum protection allowed when mapped by a process, and enables objects to be shared with different protection levels (`sepos_object_share()`) between processes (Figure 3). Access control lists may also be used to specify default permissions for privileged (10001) and non-privileged (10000) processes. We describe application privileges and entitlements and how they are used by bootstrap server methods in the next section.

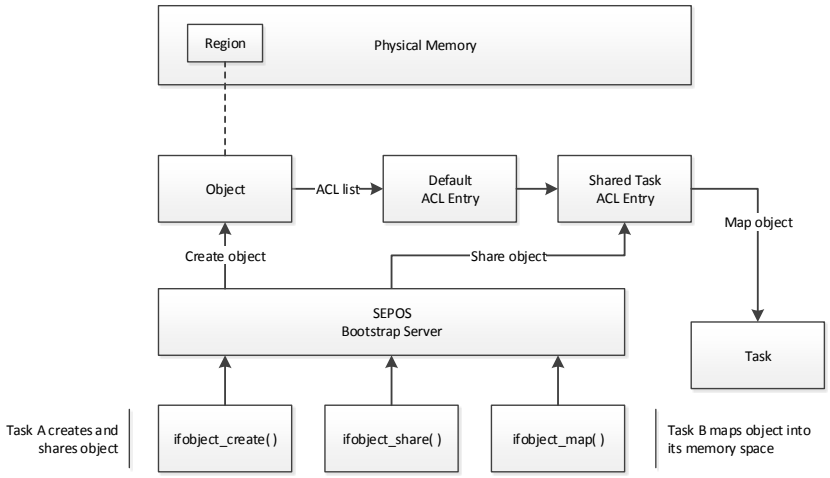

Fig. 3. Objects and access control lists

Privileged Methods Although most methods implemented by the bootstrap server is made available to all applications, some methods require special privileges. Notably, APIs that enable applications to query information about other processes, threads, or objects require the caller to be a privileged SEPOS application. On launch (in `proc_create()`), SEPOS marks an application as privileged if the first four letters of the application name is in the range of 'A ' (0x41202020) and 'ZZZZ' (0x5A5A5A5A). Methods then check if a process is privileged by invoking the function `proc_has_privilege( int pid )`, where `pid` specifies the process identifier of the application that made the request. An example of a privileged method is shown in Listing 17.

```
int sepos_object_acl_info(int *args)
{
    int result;
    int prot;
    int pid;

    args[18] = 1;
    *((_BYTE *)args + 104) = 1;
    result = proc_has_privilege( args[1] ); // check sender pid
    if ( result == 1 )
    {
        result = acl_get( args[5], args[6], &pid, &prot);
        if ( !result )
        {
            args[18] = 0;
            args[19] = prot;
            args[20] = pid;
            result = 1;
            *((_BYTE *)args + 104) = 1;
        }
    }
    return result;
}
```

Listing 17: Privileged bootstrap server method

Some methods provided by the bootstrap server are also considered very powerful and thus require special entitlements to be used. Examples of such APIs include `sepos_object_create_phys()` and `sepos_object_remap()`, both of which allow applications to map arbitrary physical memory. Applications are assigned entitlements on startup (see Listing 18) by SEPOS if they are listed in a table of entitled tasks.

The `task_entitlements` variable references a table embedded by the DATA segment of the SEPOS binary, shown in Listing 19. Notably, each entry in this table holds the first four letters of the application name (the application identifier), followed by the entitlements to be assigned on startup.

In particular, the privileged tasks table is used to assign entitlements to three applications: SEPDrivers, ARTManager/ARTMate and Debug. The latter is an application used by Apple internally and is not available in production builds of SEPOS. The entitlements assigned by this table indicate if an application is allowed to map a specific physical region (2) and if it also can map a physical region belonging to SEPOS (4). For instance, SEPDrivers is allowed to map a physical region as it needs to do this in order to both map I/O registers and initialize the shared buffers used for out-of-line endpoint data transfers.

```
int proc_create( int name )
{
    ...

    proctab[ pid ].privileged = ( name >= 'A   ' && name <= 'ZZZZ' );
    proctab[ pid ].entitlements = 0;

    while ( task_entitlements[ 2 * i ] != name )
        if ( ++i == 3 )
            return pid;

    proctab[ pid ].entitlements  = task_entitlements[ 2 * i + 1 ];

    return pid;
}
```

Listing 18: Process entitlement assignment

```
__const:0000DE50 ; _DWORD task_entitlements[10]
__const:0000DE50 task_entitlements  DCD 'SEPD'
__const:0000DE54                    DCD 2
__const:0000DE58                    DCD 'ARTM'
__const:0000DE5C                    DCD 6
__const:0000DE60                    DCD 'Debu'
__const:0000DE64                    DCD 6
```

Listing 19: Task entitlements table

Although the applications listed previously are allowed to map physical memory, both `sepos_object_create_phys()` and `sepos_object_remap()` also enforce additional validation on the provided physical address. In particular, these functions call `phys_addr_validate()` (shown in Listing 20) to make sure that the physical address provided only references I/O registers or AP/SEP physical memory.

Applications do not directly interact with the bootstrap server. Instead, high level APIs internally make use of it when needed. Most of these APIs are implemented in libSEPOS, a library primarily designed to interface with SEPOS and provide methods to simplify tasks such as driver and service interaction. libSEPOS also implement more familiar mechanics such as the workloop.

```
#define NS( x ) ( x & 0xffffff7fffffffff ) // non-secure addr

int phys_addr_validate( uint64_t addr, unsigned int size )
{
    uint64_t end = addr + size;

    if ( end <= addr )
        return 0;

    // check io regs
    if ( ( NS( addr ) >= 0x200000000 ) && ( NS( end ) <= 0x300000000 ) )
        return 1;

    // check good phys region
    if ( ( NS( addr ) >= 0x800000000 ) && ( NS( end ) <= 0x1000000000 ) )
        return 1;

    return 0;
}
```

Listing 20: Physical address validation

3.3 Drivers

SEPOS includes several drivers that are designed to support services and applications such as the True Random Number Generator (TRNG) and the AP/SEP endpoint driver (AKF). These drivers are hosted by a single application named SEPDrivers and runs entirely in user-mode. In order to support low-level operations, each driver also maps a copy of any associated I/O registers into the address space of the driver application. This allows drivers in SEPOS to operate on the I/O registers directly without the need to resort to kernel provided services. The list of drivers currently implemented in SEPOS is shown in Listing 6.

Each driver loaded by SEPDrivers is represented by an object that inherits a class named `Driver`. The `Driver` class provides a set of default methods used to handle various driver related events, such as received interrupts, control requests, and read/write operations. A driver may overload any of these methods whenever it wants to do any custom handling. For instance, the TRNG driver overloads the read method in order to provide random data back to the caller.

When the driver application starts up, the class object of each driver is initialized by invoking their constructor methods. Internally, these methods call the `Driver::Driver()` constructor method in order to initialize the driver object and map the associated I/O registers using the provided physical address. This function also calls `Driver::interrupt_register()` in order to reg-

Name	Description	I/O Registers
AES_SEP	AES SEP	0x20d300000
AES_HDCP	AES HDCP	0x20d380000
AES_AP_CMC	AES AP (CMC)	0x20d001000
AKF	Endpoint management (mailbox)	0x20da04000
Console	Console output	n/a
Expert	Platform expert	0x2102bc000
GPIO	General-purpose input/output	0x20df00000
I2C	Inter-integrated circuit (I2C)	0x20d700000
KEY	Key management	0x20d680000
PKA		0x20d600000
PMGR	Power management	0x20d200000
TRNG	True random number generator	0x20d500000
Timers	Clock timer	n/a

Table 6. Drivers in SEPOS (iOS 9; iPhone 6S)

ister any interrupts that it wishes to respond to. Finally, the constructor calls `Driver::spawn()` in order to register the driver thread name with SEPOS and start the driver workloop. This workloop responds to any requests made from other threads and optionally invokes any of the `Driver` class methods for the specific driver depending on the operation requested.

Function	Description
`driver_lookup(name, int)`	Lookup handle to driver
`driver_control(handle, opcode, buf, count)`	Perform a driver control request
`driver_read(handle, endpoint, buf, len)`	Read bytes from a driver
`driver_write(handle, endpoint, buf, len)`	Write bytes to a driver

Table 7. Driver API

Once all drivers have been initialized and registered with SEPOS, SEPDrivers starts a woorkloop of its own and registers the workloop thread as the service "SEPD". The SEPD service is used to translate driver names into handles (thread ids) which subsequently may be used to talk to a specific driver over RPC. This mechanism is specifically leveraged by the driver API (Table 7) provided to applications when attempting to perform a driver lookup (Figure 4).

Once a handle to a specific driver has been obtained via `driver_lookup()`, applications then use `driver_control()` to invoke a specific driver method or `driver_read()`/`driver_write()` in order to perform a read or write operation.

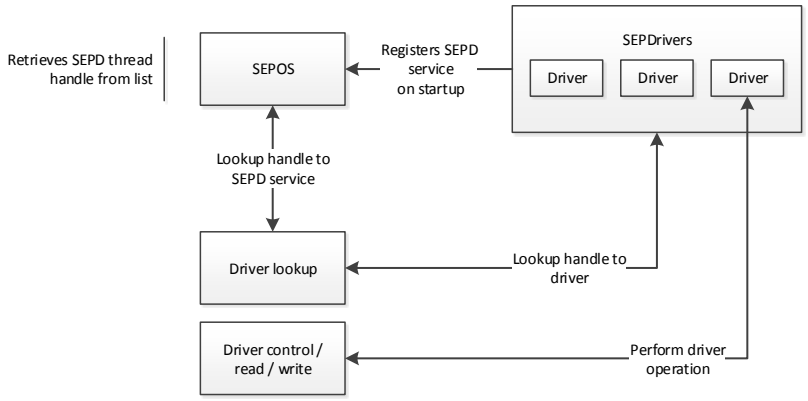

Fig. 4. Driver interaction

AKF The AKF driver is responsible for managing endpoints and also interacts with the secure mailbox from the SEP. It enables SEP applications to register new endpoints and communicate data with endpoints registered in the application processor. When initializing the AKF driver, the physical address of the secure mailbox is provided as the associated device I/O registers to map upon load. These registers are essentially the same I/O registers mapped by `AppleA7IOP` in iOS. The AKF driver is also configured to respond to the interrupts generated on mailbox updates in order to allow incoming messages to be handled.

Whenever a SEP application wants to receive messages sent by the AP, it first needs to register a new endpoint with AKF. This is typically done by first looking up the handle to the AKF driver using `driver_lookup()`. The application then issues an endpoint registration request (0x412C) using `driver_control()` and passes a pointer to the requested endpoint number as the argument. On success, an interest notification (0x4082) may optionally be registered for the endpoint in order for the application to be notified about any incoming messages. Futhermore, if the application needs to map out-of-line buffers, it may request buffer objects by issuing the request for incoming (0x412D) or outgoing (0x412E) buffers. These buffer objects may then subsequently be mapped into the application address space by invoking the `sepos_object_map()` routine exported by the bootstrap server.

In order to receive an endpoint message, a SEP application reads 8 bytes from the AKF driver using `driver_read()`. This API also takes an endpoint number and allows AKF to return the message for a specific endpoint. Once a message has been processed by the application, a response may be written back to the endpoint using `driver_write()`. Again, this API also takes an argument allowing the caller to specify which endpoint the message is to be written to.

Besides managing endpoints and relaying messages, AKF also handles some endpoint messages on its own. Notably, AKF creates threads to handle both con-

trol endpoint (EP0) and log endpoint (EP1) messages. As mentioned in Section 2.6, the control endpoint is used to perform certain management related SEP tasks, such as setting up the out-of-line buffers associated with endpoints.

3.4 Services

The services provided in SEPOS are like drivers also hosted by their own application. However, unlike drivers, services are implemented in a much simpler way. Notably, the services application (named sepServices) defines each service it provides as a server. These servers then provide a set of methods in order to support the operations they implement. The list of available servers in SEPOS is shown in Table 8.

Server	Name
skgs	Key Generation Service
test	Test Service
sars	Anti Replay Service
enti	Entitlement Service

Table 8. Services in SEPOS (iOS 9)

On startup, sepServices registers itself with the bootstrap server as the service "sepS". It also creates a thread with an associated workloop for each server it implements. In order to simplify service interaction, a separate service API is provided in libSEPOS, shown in Table 9.

Function	Description
`service_lookup( int name )`	Lookup handle to service
`service_call( int handle, ... )`	Invoke service routine

Table 9. Service API

An application that wants to request a specific service may do so by first calling `service_lookup()` and by specifying a 4-byte service name. Internally, this function requests the "sepS" service from the bootstrap server and invokes the lookup method exported by sepServices in order to find the requested service. This causes sepServices to iterate its list of servers and return a handle back to the application if a match is found. The application may then use the returned handle (actually a thread id, used to establish an IPC connection) and call `service_call()` in order to request a specific operation to be performed by the service (Figure 5).

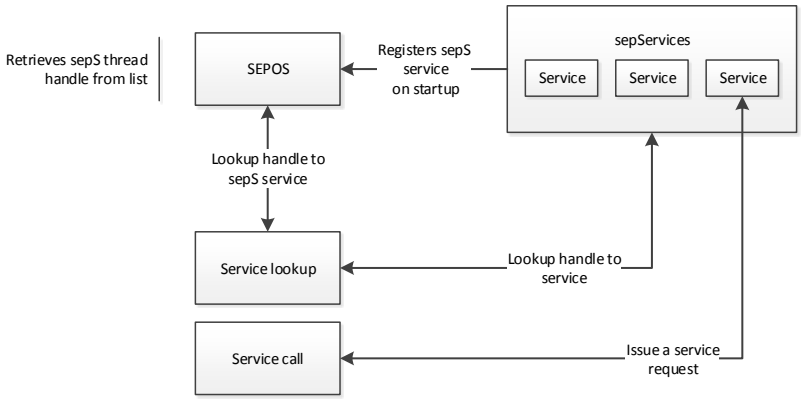

Fig. 5. Service interaction

3.5 Applications

Alongside drivers and services, SEPOS also runs several applications designed to support various applications, services, and frameworks implemented in the application processor (iOS). These applications are listed below.

ART Manager/Mate The anti-replay token manager/mate is an application that handles anti-replay tokens. The application was originally called ART Manager, but was renamed to ART Mate after the Counter Service was replaced with the Anti-Replay Service for the iPhone 6S. Note that the original ART Manager is still present on older devices (iPhone 6 and prior).

Secure Biometric Engine The secure biometric engine (sbio) application is responsible for handling biometric information and listens on endpoint 8 for request sent by the AppleMesaSEPDriver kernel extension.

Secure Credential Manager The secure credential manager (scrd) application implements the user client methods that were previously found in AppleCredentialManager.kext. This way, none of the internal data structures are exposed to iOS (AP), as only only wrapper functions remain in the AppleSEPCredentialManager extension.

Secure Key Store The secure key store (sks) application implements the user client methods that were previously found in the AppleKeyStore iOS kernel extension. This way, none of the internal data structures are exposed to iOS (AP), as only only wrapper functions remain in the AppleSEPKeyStore extension.

SSE The SEP secure element (sse) is an application that handles requests for the Secure Element. It receives and responds to requests from the AppleSSE.kext when required. Note that the SSE application is only present on devices that implement support for the Secure Element (aka Apple Pay) such as the iPhone 6S.

4 Analysis

In this Section, we first evaluate the attack surface exposed by SEPOS from the application processor (iOS). We then evaluate the robustness of SEPOS by assessing its mitigations and ability to withstand various forms of attacks.

4.1 Attack Surface

From a software point of view, evaluating the SEP attack surface mostly comprises the methods in which data is communicated between the AP and SEP. Assuming that an attacker already has obtained AP kernel level privileges (i.e. has the ability to execute arbitrary code under EL1), this essentially boils down to looking at how mailbox messages and the shared out-of-line buffers are handled.

As noted in Section 3.3, a SEP application needs to register an endpoint with AKF in order to handle messages sent from the AP. As such, every endpoint registered with AKF contributes to the SEP attack surface and is a potential target. This also includes endpoints that don't have a corresponding endpoint registered in the AP. A list of endpoints registered by applications in SEP and their use of out-of-line request and reply buffers is shown in Table 10. Note that some applications/endpoints are only present on some devices.

Endpoint	Owner	OOL In	OOL Out	Notes
0	SEPD/ep0			Control endpoint
1	SEPD/ep1		✓	
2	ARTM	✓	✓	iPhone 6 and prior
3	ARTM	✓	✓	iPhone 6 and prior
7	sks/sks	✓	✓	
8	sbio/sbio	✓	✓	
10	scrd/scrd	✓	✓	
12	sse/sse	✓	✓	iPhone 6 and later

Table 10. AKF registered endpoints (on devices running iOS 9)

As also noted previously, applications receive mailbox messages by reading 8 bytes from the AKF driver upon receiving an interest notification. These messages are then typically handled by a dedicated function specific to the application, often containing a switch-case statement for all the supported opcodes. The complexity of the parsing performed by these functions varies between applications, but may in some cases be quite involved as shown by the function call graph in Figure 6.

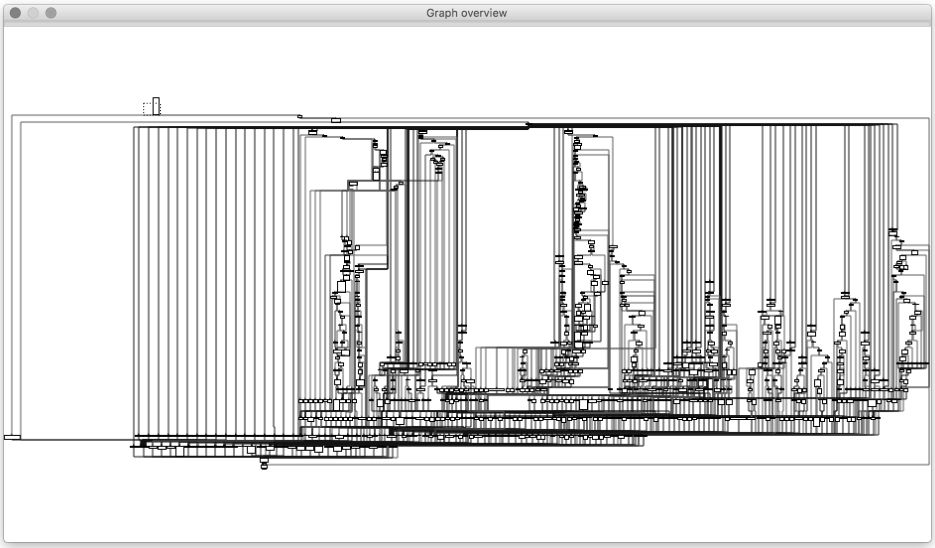

Fig. 6. Secure Biometric Engine message handler

4.2 Robustness

In order to determine robustness, we analyze the effort required to exploit typical vulnerabilities such as stack and heap based buffer overflows. Specifically, this includes looking at the hurdles typically encountered during exploit development such as address space layout randomization, allocator hardening and exploit mitigations.

Address Space Layout Randomization Most modern operating systems include some form of process space randomization in order to make it more difficult for attackers to leverage known locations in memory when exploiting vulnerabilities. This typically involves randomizing the base address of the loaded image(s), as well as stack and heap addresses.

In SEPOS, applications are always loaded at their preferred base address. In fact, most applications are either loaded at virtual base address 0x8000 or 0x10000 depending on whether the binary was compiled with a PAGEZERO segment or not. Recall from Section 3.2 that the image segments of a Mach-O binary are mapped using `macho2vm()` during process initialization. A simplified version of this function is shown in Listing 21.

Notably, `macho2vm()` iterates through all the vm segments present in the provided macho image, and creates a memory object using `object_create_phys()` for each segment. The `object_create_phys()` function creates an object with the address and size of the physical region that holds the segment data. This region is subsequently mapped into the process address space using the speci-

```
int
macho2vm( int pid, void * macho, uint64_t physaddr )
{
    if ( macho_first_vm_segment( macho, &seg ) )
    {
        while ( 1 )
        {
            if ( seg.initprot && seg.vmsize )
            {
                // create memory object
                obj = object_create_phys( ... );

                if (!obj)
                    return ENOMEM;

                // map memory object
                if ( !object_map_fixed( pid, obj, seg.vmaddr, seg.initprot ) )
                    return 2;

                if ( !macho_next_vm_segment( &seg ) )
                    return 0;
            }
        }
    }

    return 0;
}
```

Listing 21: Function for mapping Mach-O segments

fied memory protection (`seg.initprot`). As can be seen, segments are always mapped at the virtual address provided in the segment header (`seg.vmaddr`). Note also that `macho2vm()` requires the initial protection mask of a segment to be non-zero in order to be processed. This consequently results in segments such as PAGEZERO (where the initial and maximum protection masks are both set to zero) to be ignored.

Another important factor to consider when looking at the address space layout of a process is how virtual memory is allocated at runtime. Since most allocators rely on a mechanism to allocate additional memory if needed, being able to predict these addresses may assist greatly in exploitation. In SEPOS, additional virtual memory is generally requested by leveraging the object APIs provided by the bootstrap server. Notably, an application cannot choose a specific memory region to allocate in its address space as this is entirely managed by the SEPOS task. In particular, whenever an application invokes the `sepos_object_map()`

method to map an object into its process space, SEPOS determines if it first has access to the object by inspecting its access control list. It then retrieves the next available address in its virtual address space, starting from the lowest address. Once a location is found, it adds 0x4000 bytes to the allocation address. Although this results in a fairly predictable layout, the mappings become non-contiguous, consequently making it harder to exploit out-of-bounds accesses.

Although most services in SEPOS are implemented in user-mode, the kernel itself may also be of interest to attackers in order to gain elevated privileges. As kernel vulnerabilities typically require the attacker to learn some information about the kernel address space in order to be practically exploitable, knowing the kernel base address or predicting the addresses of its allocations may be very valuable. Unfortunately, the SEPOS kernel does not make an effort to complicate exploitation by randomizing the kernel address space. In particular, the kernel is always mapped at 0xF0001000, with its page tables and memory allocations directly following the kernel image.

Stack Hardening Stack corruptions are one of the most basic and widely understood vulnerability classes. As such, numerous mitigations such as stack randomization, stack frame reorganizing, and stack cookies have been introduced in order to make stack corruption vulnerabilities more difficult to exploit. The stack also plays an important part in return-oriented programming (ROP), and has received much attention for that reason in both exploitation and mitigation research in the last 5-10 years.

Randomization of an application's thread stacks is typically important in order to prevent ret-to-stack attacks as well as preventing an attacker from targeting specific values or pointers on the stack. When SEPOS starts a new application, it does not allocate a stack for the main thread. Instead, the main thread uses an image embedded stack (`__sys_stack`) located in the DATA segment of the application image. This is potentially bad in two ways: It allows an attacker to easily predict the stack base address as the address of the DATA segment is already known. It may also allow an attacker to corrupt adjacent data in the DATA segment if a stack corruption is sufficiently large.

While the main thread stack is always located in the DATA segment, additional threads have their stacks allocated at runtime using objects. In particular, the `sepos_thread_create()` method exported by the bootstrap server creates an anonymous thread stack object (of default size 0x2000) and maps it into the address space of the target process. Again, since this mechanism leverages object mappings, they are not randomized, but allocated in a deterministic and sequential pattern (as explained in Section 4.2). On the other hand, as object mappings have gaps between them, they provide a guard-page like protection (Figure 7).

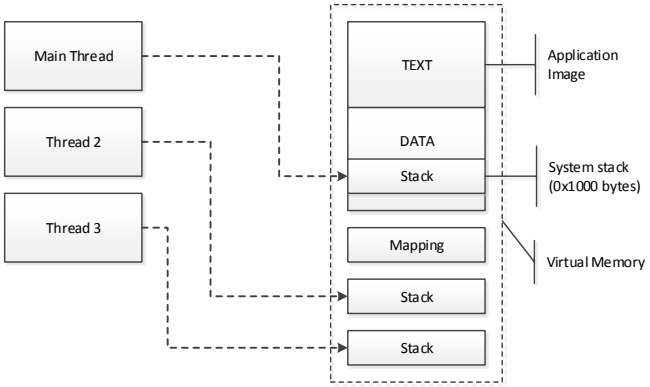

Fig. 7. Application thread stacks

Although SEP applications do not randomize thread stacks, they do try to mitigate stack corruptions by introducing stack cookies during compilation. Consider a very simple function such as `driver_register_interest()`, shown in Listing 22.

```
__text:0000BBF4                PUSH            {R4,R7,LR}
__text:0000BBF6                ADD             R7, SP, #4
__text:0000BBF8                SUB             SP, SP, #0xC
__text:0000BBFA                MOVW            R4, #(___stack_chk_guard)
__text:0000BBFE                MOV             R3, SP
__text:0000BC04                LDR             R4, [R4]
__text:0000BC06                STR             R4, [SP,#0x10+cookie]
__text:0000BC08                STMEA.W         SP, {R1,R2}
__text:0000BC0C                MOVW            R1, #0x4082
__text:0000BC10                MOVS            R2, #2
__text:0000BC12                BL              _driver_control
__text:0000BC16                LDR             R1, [SP,#0x10+cookie]
__text:0000BC18                SUBS            R1, R4, R1
__text:0000BC1A                ITT EQ
__text:0000BC1C                ADDEQ           SP, SP, #0xC
__text:0000BC1E                POPEQ           {R4,R7,PC}
__text:0000BC20                BL              ___stack_chk_fail
```

Listing 22: Function compiled with stack cookie check

This function invokes `driver_control()` in order to perform a control request using an 8-byte buffer. Because data may be written back to this buffer, a stack cookie check is compiled in. This cookie is loaded from a global variable in the function prologue and later on checked again in the epilogue.

Unfortunately, the stack cookie is not randomized at all and always set to the value `GARD`. Thus, even a simple string bounded by a null terminator can be used to bypass the cookie check when attempting to exploit a stack corruption vulnerability. Ideally, the cookie value should be sufficiently randomized such that it cannot be easily guessed by an attacker.

Heap Hardening Memory corruption vulnerabilities affecting dynamically allocated memory, i.e. heap corruptions, is another common vulnerability class prevalent on many platforms. Although SEP applications sometimes leverage objects to allocate virtual memory, they primarily use one of the `malloc()` functions provided by the C runtime.

In SEPOS, `malloc()` leverages the K&R implementation. The name K&R originates from the original `malloc()` implementation found in the book on C by Kernighan and Ritchie. This implementation of `malloc()` has also been used in other L4 distributions such as Darbat. In particular, K&R leverages a singly linked free list where each block is ordered by size. In order to keep track of the size, each block is also preceded by a `Header` structure, shown in Listing 23. This structure both contains a pointer to the next free block (if on the free list) and the size of the block. Whenever a block is freed, K&R also looks at adjacent blocks and, if free, coalesces them into a larger block.

```
typedef long long Align; /* for alignment to long long boundary */

union header { /* block header */
    struct {
        union header   *ptr; /* next block if on free list */
        unsigned       size; /* size of this block */
    } s;
    Align              x;    /* force alignment of blocks */
};

typedef union header Header;
```

Listing 23: `malloc()` `Header` type definition

Interestingly, `malloc()` does not allocate any new pages on its own. Instead, each application registers a fixed size memory region it wants to reserve for `malloc()` during startup (typically in its `main()` function) by calling `__malloc_init()`. In SEPOS, applications commonly leverage a malloc area embedded by the DATA segment of the application image. Again, similar to the main thread stack, this makes the address of `malloc()` blocks very predictable. Additionally, as new `malloc()` blocks are allocated from the reserved region in

sequential order, an attacker can easily determine and manipulate the process heap layout.

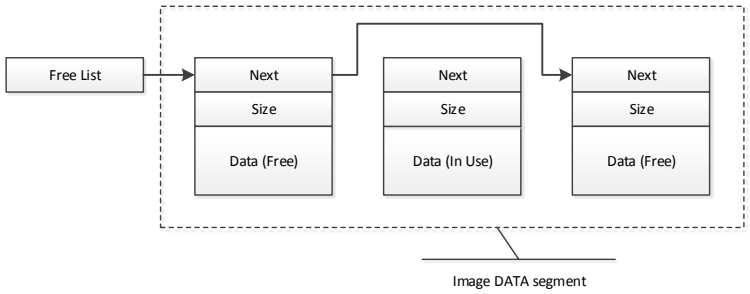

Fig. 8. `malloc()` heap in SEP applications

When exploiting heap corruption vulnerabilities, attackers usually have a choice between attacking heap metadata or application data. Targeting heap metadata is commonly considered to be the better option of the two as it can be used more generically. However, because of this, heap metadata has also received more hardening over the years such as header XOR encoding and safety checks of linked list operations. SEPOS (and presumably many other L4 distributions), on the other hand, does not introduce any hardening of the heap metadata. Thus, an attacker may easily overwrite free list pointers of adjacent blocks in order to corrupt arbitrary memory or corrupt the block size of a block in order to overwrite adjacent data.

No-Execute Protection In the process of exploiting a vulnerability, an attacker typically needs to inject the set of instructions that he or she wants the system to finally execute. Many years ago, this was seen as a trivial task as operating systems would happily execute code residing on the stack or in the process heap. However, since processors have introduced support for non-executable memory, this is no longer considered to be an easy task.

Although many L4 distributions historically have lacked support for non-executable memory, SEPOS does implement support for both XN (Execute-Never) and PXN (Privileged Execute Never). In fact, all pages mapped into a process space have both the XN and PXN bits set in their backing page table entries unless explicitly marked as executable (in which user-mode pages will be marked with PXN). This can be observed by inspecting `space_t::map_fpage()` in the SEPOS kernel (excerpt shown in Listing 24). Note here that SEPOS operates on ARMv8 (64-bit) page tables.

```
__text:F0005504                 MOV             R0, R6
__text:F0005506                 MOV             R1, R4
__text:F0005508                 BL              is_nonsecure_addr
__text:F000550C                 UXTH            R1, R4
__text:F000550E                 AND.W           R2, R11, #0x600000
__text:F0005512                 CMP             R0, #0
__text:F0005514                 ORR.W           R4, R5, #0x20
__text:F0005518                 ORR.W           R1, R1, R2 ; set XN/PXN
__text:F000551C                 MOVW            R2, #0xFDF
__text:F0005520                 IT EQ
__text:F0005522                 ANDEQ.W         R4, R5, R2 ; mask NS bit
__text:F0005526                 LSRS            R0, R6, #0xC
__text:F0005528                 BFI.W           R4, R0, #0xC, #0x14
__text:F000552C                 LDR             R0, [SP,#0x20+pte]
__text:F000552E                 STR             R4, [R0] ; modify pte
__text:F0005530                 STR             R1, [R0,#4]
__text:F0005532                 MOVS            R0, #0
```

Listing 24: Page table entry update

Summary In Table 11, we summarize the exploit mitigations present in SEPOS.

Mitigation	Present	Notes
Stack Cookie Protection	Yes (...)	Fixed cookie value
Address Space Layout Randomization	No	
Stack Guard Pages	Yes/No	Not for main thread
Object Map Guard Pages	Yes	Gaps between object mappings
Heap Metadata Protection	No	
Null-Page Protection	No	Must be root task to map page
No-Execute Protection	Yes	Both XN and PXN

Table 11. Mitigations summary

5 Conclusion

In this paper, we have detailed the components of SEPOS and shown how iOS and SEP communicate using an interrupt-based mailbox mechanism. In particular, we have shown that SEPOS exposes several endpoints to iOS, in order to support services such as the Apple Key Store, Credential Manager, and Touch ID. These endpoints collectively make up the SEP attack surface, some of which implement very complex message handlers. In order to determine the exploitability of common issues such as stack and heap based buffer overruns, we also assessed the attack robustness of SEPOS. In particular, we found that SEPOS lacks many of the exploit mitigations found in contemporary operating systems. Notably, SEPOS lacks address space layout randomization, randomized stack cookies, and hardening against heap metadata attacks.

References

[1] Apple Inc.: iOS Security - White Paper May 2016. `http://www.apple.com/business/docs/iOS_Security_Guide.pdf`

[2] Apple Inc.: Patent US8832465 Security enclave processor for a system on a chip. September 2014. `http://www.google.com/patents/US8832465`

[3] Gernot Heiser, Kevin Elphinstone. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. ACM Transactions on Computer Systems. `https://www.nicta.com.au/publications/research-publications/?pid=8988`

[4] Jochen Liedtke. On -kernel construction. Symposium on Operating Systems, pages 237250, Copper Mountain, CO, USA, December 1995.

A AP/SEP Endpoints

Index	Name (class)	Driver (kext)	Description
0	`AppleSEPControl`	`AppleSEPManager`	Control
1	`AppleSEPLogger`	`AppleSEPManager`	Logging
2	`AppleSEPARTStorage`	`AppleSEPManager`	ART Storage
3	`AppleSEPARTRequests`	`AppleSEPManager`	ART Requests
4	`AppleSEPTracer`	`AppleSEPManager`	Tracer
5	`AppleSEPDebug`	`AppleSEPManager`	Debug
7	`AppleSEPKeyStore`	`AppleSEPKeyStore`	Keystore
8	`AppleMesaSEPDriver`	`AppleMesaSEPDriver`	MESA
9	`AppleSPIBiometricSensor`	`AppleBiometricSensor`	Biometrics
10	`AppleSEPCredentialManager`	`AppleSEPCredentialManager`	Credentials
11	`AppleSEPPairing`	`AppleSEPManager`	Pairing
12	`AppleSSE`	`AppleSSE`	Secure Element
254	`L4Info`	?	
255	`SEP Bootrom`	?	

B SEPOS Methods

B.1 System Methods

Class	Id	Function	Description
0	0	`sepos_proc_getpid()`	Get the process pid
0	1	`sepos_proc_find_service()`	Find a registered service by name
0	1001	`sepos_proc_limits()`	Query process limits
0	1002	`sepos_proc_info()`	Query process info
0	1003	`sepos_thread_info()`	Query thread info
0	1004	`sepos_thread_info_by_tid()`	Query info for thread id
0	1100	`sepos_grant_capability()`	
0	2000	`sepos_panic()`	Panics the operating system

B.2 Object Methods

Class	Id	Function	Description
1	0	`sepos_object_create()`	Create an anonymous object
1	1	`sepos_object_create_phys()`	Create an object from a physical region
1	2	`sepos_object_map()`	Map an object in a task's address space
1	3	`sepos_object_unmap()`	Unmap an object (not implemented)
1	4	`sepos_object_share()`	Share an object with a task
1	5	`sepos_object_access()`	Query the access control list of an object
1	6	`sepos_object_remap()`	Remap the physical region of an object
1	1001	`sepos_object_object_info()`	Query object information
1	1002	`sepos_object_mapping_info()`	Query mapping information
1	1003	`sepos_object_proc_info()`	Query process information
1	1004	`sepos_object_acl_info()`	Query access control list information

B.3 Thread Methods

Class	Id	Function	Description
2	0	`sepos_thread_create()`	Create a new thread
2	1	`sepos_thread_kill()`	Kill a thread (not implemented)
2	2	`sepos_thread_set_name()`	Set a service name for a thread
2	3	`sepos_thread_get_info()`	Get thread information

B.4 Miscellanous Methods

Class	Id	Function	Description
3	-1	`sepos_irq_interface()`	
105	100	`sepos_power_setmin()`	
106	100	`sepos_system_information()`	Query system information