Audit tokens explained

Scott Knight

The recent Objective by the Sea v3.0 conference had a lot of great talks. Two that stood out to me were *Abusing and Securing XPC in macOS Apps* by Wojciech Reguła and *Job(s) Bless Us! Privileged Operations on macOS* by Julia Vashchenko. Both talks discussed different aspects of XPC services and the types of security bugs that can occur in them. There were some great best practice recommendations that both speakers shared for securing your own XPC services. One of those recommendations was to use the audit token rather than PID when checking the connecting process. Since the audit token APIs aren't public I thought it would be interesting to take a closer look at what audit tokens actually are and where they come from.

PID vs. audit token

I think it's worth brielfy covering the XPC best practice around PID vs. audit token before diving into the history and internals of the audit token itself. A XPC service listener has the chance to accept or deny an incoming connections. The most common way to validate the incoming connection is to make use of the Security framework provided by Apple. This framework provides a way to access the code signing

information of the remote process and you can then check to make sure the calling process is validly signed by your own company. A common way to get access to the code signing information is to do something like the following:

```
SecCodeRef code = NULL;
NSDictionary *attributes = @{
    kSecGuestAttributePid: @(connection.processIdentifier)
};
SecCodeCopyGuestWithAttributes(0, attributes, kSecCSDefaultFlags
```

The issue with this is since it only uses PID to locate the process, and PIDs wrap around and can be reused, it's possible to race the code signing verification and trick it into checking a different binary. Ian Beer and Samuel Groß both have some good descriptions of this issue:

- https://bugs.chromium.org/p/project-zero/issues/detail?id=1223
- https://saelo.github.io/presentations/warcon18_dont_trust_the_pid.pdf

The fix for the code above is to do the following:

```
SecCodeRef code = NULL;
NSDictionary *attributes = @{
    kSecGuestAttributeAudit: @(connection.auditToken)
};
```

```
SecCodeCopyGuestWithAttributes(0, attributes, kSecCSDefaultFlags
```

There's one catch with this updated code. The `auditToken` property is not a public property of `NSXPCConnection`. Wojciech Reguła has a great sample XPC application that shows an easy way to access this private property. As we'll see shortly, making use of the `auditToken` allows the Security framework to use the internal fields to ensure the connecting process is the one the XPC service thinks it is.

The history of audit tokens

The audit token mentioned above is actually part of the OpenBSM subsytem in macOS. OpenBSM is an open source implementation of Sun's Basic Security Module (BSM) security audit API and file format. Back in 2003 Apple decided to submit macOS to the National Information Assurance Partnership (NIAP) for Common Criteria certification. As part of preparing for this certification Apple contracted McAfee to port the Solaris BSM implementation to macOS.

The BSM code can be seen in the XNU kernel as far back as xnu-517 which was released in October of 2003 as part of macOS 10.3.0. It wasn't until more than a year later in November of 2004 when macOS 10.3.6 was released that end users could actually optionaly install the ported BSM

code to enable auditing on their macOS systems. Shortly after the macOS 10.3.6 release Apple did officially receive Common Criteria certification. This certification validated macOS for high security uses such as by the U.S. Government.

Apple eventualy relicensed their BSM port under the BSD license to allow integration into FreeBSD and other systems. This code base is what eventually became the OpenBSM project.

Kernel implementation

Now that we know some of the history behind the audit token lets actually take a look at what it is and how the kernel implements and stores it. We start be taking a look at the definition of `audit_token_t` itself.

```
/*
 * The audit token is an opaque token which identifies
 * Mach tasks and senders of Mach messages as subjects
 * to the BSM audit system.  Only the appropriate BSM
 * library routines should be used to interpret the
 * contents of the audit token as the representation
 * of the subject identity within the token may change
 * over time.
 */
typedef struct{
    unsigned int                    val[8];
```

```
} audit_token_t;
```

We can already start to see why the `auditToken` property of `NSXPCConnection` is marked as private. The comment explicitly mentions that `audit_token_t` structure is meant to be opaque and only the appropriatee BSM library routines should be used to interpret its contents.

task structure

Continuing on in the kernel code if we search for other structures that make use of `audit_token_t` we find the following:

```
struct task {
    /* Synchronization/destruction information */
    decl_lck_mtx_data(, lock);               /* Task's lock */
    os_refcnt_t     ref_count;      /* Number of references to m
    boolean_t       active;         /* Task has not been termina
    boolean_t       halting;        /* Task is being halted */
    ...

    /* Task security and audit tokens */
    security_token_t sec_token;
    audit_token_t   audit_token;

    ...
};
```

A `task` is one of the fundamental structures of the Mach portion of the XNU kernel. It represents a collection of resources, virtual address space, and a port name space. It contains a queue of `thread` structures which are what are actually responsible for running code. Searching further through the code we can find the `int set_security_token_task_internal(proc_t p, void *t)` function which actually sets the fields of the audit token. Here's an excerpt from that function:

```
my_cred = kauth_cred_proc_ref(p);
my_pcred = posix_cred_get(my_cred);

...

/*
 * The current layout of the Mach audit token explicitly
 * adds these fields.  But nobody should rely on such
 * a literal representation.  Instead, the BSM library
 * provides a function to convert an audit token into
 * a BSM subject.  Use of that mechanism will isolate
 * the user of the trailer from future representation
 * changes.
 */
audit_token.val[0] = my_cred->cr_audit.as_aia_p->ai_auid;
audit_token.val[1] = my_pcred->cr_uid;
audit_token.val[2] = my_pcred->cr_gid;
audit_token.val[3] = my_pcred->cr_ruid;
audit_token.val[4] = my_pcred->cr_rgid;
```

```
audit_token.val[5] = p->p_pid;
audit_token.val[6] = my_cred->cr_audit.as_aia_p->ai_asid;
audit_token.val[7] = p->p_idversion;
```

Again, we see mention in the comments that only the appropriate BSM library functions should be used to interpret an `audit_token_t` structure. If we look in the OpenBSM code, we can see that the function audit_token_to_au32 is the function that should be used to extract the contents out of an `audit_token_t` structure.

Now that we know a `tasks`'s `audit_token` field is set from `set_security_token_task_internal` lets continue further to see where this code is actually called from. There are three functions that call `set_security_token_task_internal`:

- `exec_handle_sugid`
- `audit_session_join_internal`
- `set_security_token`

We'll ignore the first two and instead focus on `set_security_token` for now. Searching for `set_security_token` we see the following call sites:

- `load_init_program_at_path`
- `fork1`
- `setuid`

- `seteuid`
- `setreuid`
- `setgid`
- `setegid`
- `setregid`
- `setgroups_internal`
- `persona_proc_adopt`
- `audit_session_setaia`

The first function `load_init_program_at_path` is responsible for starting `launchd`. The call to `set_security_token` here ensures that even the `launchd` process has an `audit_token` set on its `task` structure. The `fork1` function is eventually called from either the `fork` or `vfork` system calls. Since `fork` and `vfork` are responsible for creating new processes in the system it ensures all new processes also have an `audit_token` set on their `task` structure. All of the `setXXX` functions are responsible for changing a field that is also stored as part of the `audit_token_t` structure. It makes sense that if any of those fields change then the `audit_token_t` values should also be updated. The `persona_proc_adopt` is part of the persona subsystem. The persona subsystem provides a mechanism for a process or thread to assume a uid, gid and group memberships all at once. So again this is similar to the `setXXX` functions. Since the `audit_token_t` structure stores some of the audit session related identifiers

we also have a call to `set_security_token` from `audit_session_setaia` which is responsible for updating credentials of a process based on new audit info.

All of these calls just go to ensure that a `task` always has it's `audit_token` field populated and updated appropriately throughout it's lifetime. It is actually possible to retrieve a `task`'s `audit_token` using the Mach `task_info` API. Here's a short example:

```
audit_token_t token;
mach_msg_type_number_t size = TASK_AUDIT_TOKEN_COUNT;

kr = task_info(mach_task_self(), TASK_AUDIT_TOKEN, (task_info_t)
if (kr != KERN_SUCCESS) {
    printf("Error getting task audit_token\n");
    exit(1);
}
```

There is actually another Mach API call to set a `task`'s `audit_token` called `host_security_set_task_token`. The first parameter of the function is the host security port which is not accessible from user space at all. So in practice only the kernal can change a `task`'s `audit_token`.

mach message trailers

If we continue searching through the XNU source for usage

of `audit_token_t` we come across the following structure:

```
typedef struct{
    mach_msg_trailer_type_t       msgh_trailer_type;
    mach_msg_trailer_size_t       msgh_trailer_size;
    mach_port_seqno_t             msgh_seqno;
    security_token_t              msgh_sender;
    audit_token_t                 msgh_audit;
} mach_msg_audit_trailer_t;
```

Mach messages can optionally have message trailers appended to them. Trailers are not settable from user space and can only be set by the kernel. The contents of the trailer are assumed to be trusted since only the kernel can set them. If we further search the code for `msgh_audit` we can find the following call sites:

- `ipc_kmsg_get`
- `ipc_kmsg_get_from_kernel`
- `mach_msg_send`
- `mach_msg_overwrite`

These are the core mach messaging functions of the kernel and each one has a section of code similar to the following:

```
/*
 * reserve for the trailer the largest space (MAX_TRAILER_SIZE)
 * However, the internal size field of the trailer (msgh_trailer
```

```
 * is initialized to the minimum (sizeof(mach_msg_trailer_t)), t
 * the cases where no implicit data is requested.
 */
trailer = (mach_msg_max_trailer_t *) ((vm_offset_t)kmsg->ikm_hea
trailer->msgh_sender = current_thread()->task->sec_token;
trailer->msgh_audit = current_thread()->task->audit_token;
trailer->msgh_trailer_type = MACH_MSG_TRAILER_FORMAT_0;
trailer->msgh_trailer_size = MACH_MSG_TRAILER_MINIMUM_SIZE;
```

As part of the mach message routines the kernel will allocate a trailer structure of the maximum size possible. When the message is copied back to user space of the receiving process only the portion of the trailer requested will be delivered.

This is a really interesting feature because it means as part of every Mach message sent it's possible to have the sender's `audit_token` value sent along with it. Here's a short example of how this can be done.

```
struct message
{
    mach_msg_header_t header;
    char data[256];
    mach_msg_audit_trailer_t trailer;
};

mach_port_t server_port;
struct message msg;
```

```
kern_return_t kr;

kr = mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEI\
if (kr != KERN_SUCCESS) {
    printf("Error allocating port\n");
    exit(1);
}

const mach_msg_option_t options = MACH_RCV_MSG |
    MACH_RCV_TRAILER_TYPE(MACH_RCV_TRAILER_AUDIT) |
    MACH_RCV_TRAILER_ELEMENTS(MACH_RCV_TRAILER_AUDIT);

kr = mach_msg(&msg.header,
              options,
              0,
              sizeof(msg),
              server_port,
              MACH_MSG_TIMEOUT_NONE,
              MACH_PORT_NULL);

if (kr != KERN_SUCCESS) {
    printf("Error receiving mach message 0x%x\n", kr);
    exit(1);
}

printf("pid  %d\n", msg.trailer.msgh_audit.val[5]);
```

csops_audittoken system call

Continuing our search through the XNU source code there's

one more place worth mentioning making use of the `audit_token_t` structure. Suprisingly it's a portion of the code that doesn't seem to have anything to do with the OpenBSM subsystem. It's part of the code signing system calls. There's two system calls implemented in XNU to support code signing:

- `int csops(pid_t pid, unsigned int ops, void * useraddr, size_t usersize);`
- `int csops_audittoken(pid_t pid, unsigned int ops, void * useraddr, size_t usersize, audit_token_t * token);`

These system calls get used by the Security framework to support checking code signing attributes of processes. Both of them will call into the `csops_internal` function. If we look into that code we can see how the `audit_token_t` gets used:

```
pt = proc_find(pid);
if (pt == PROC_NULL) {
    return ESRCH;
}

upid = pt->p_pid;
uidversion = pt->p_idversion;
if (uauditoken != USER_ADDR_NULL) {
    error = copyin(uauditoken, &token, sizeof(audit_token_t));
    if (error != 0) {
```

```
        goto out;
    }
    /* verify the audit token pid/idversion matches with proc */
    if ((token.val[5] != upid) || (token.val[7] != uidversion))
        error = ESRCH;
        goto out;
    }
}
```

When the `csops_audittoken` version of the system call is used the code will perform an additional check to make sure the process looked up by PID actually matches the values on the `audit_token_t`. The key is to make use of the `p_idversion` field in addition to the `p_pid`. This prevents the PID reuse vulnerability that was mentioned at the start of this write up.

XPC and audit tokens

Now that we know how the kernel implements and stores the `audit_token_t` we can take a closer look at XPC and how it makes use of the functionality. Unfortunately `libxpc.dylib` is not open source so we'll have to do some reverse engineering to understand what's going on. If you're using the XPC C API instead of the higher level Objective-C API the way to get a connection's `audit_token_t` is to use `void xpc_connection_get_audit_token(xpc_connection_t,`

`audit_token_t *);`. If we disassemble `xpc_connection_get_audit_token` we see the following:

```
__int64 __fastcall xpc_connection_get_audit_token(_QWORD *connec
{
    __int64 v2; // rax

    os_unfair_lock_lock_with_options((char *)connection + 84, 0x
    token[3] = connection[15];
    token[2] = connection[14];
    v2 = connection[12];
    token[1] = connection[13];
    *token = v2;
    return os_unfair_lock_unlock((char *)connection + 84);
}
```

Since we don't have any source for `libxpc.dylib` we'll do some guess work here but the function is pretty straight forward. It simply accesses internal fields of the `xpc_connection_t` structure passed in and copies it to the `audit_token_t` pointer. I would guess that the `xpc_connection_t` structure actually has an `audit_token_t` field within it. Based on this knowledge we can continue our search looking for where this `xpc_connection_t` field gets set. With a short amount of digging we can find the `_xpc_connection_set_creds` function. Disassembled it looks like the mirror opposite of `xpc_connection_get_audit_token`:

```
__int64 __fastcall _xpc_connection_set_creds(_QWORD *a1, __int64
{
    __int64 *v2; // r14
    __int64 v3; // rax

    v2 = (__int64 *)_xpc_mach_msg_get_audit_token(a2);
    os_unfair_lock_lock_with_options((char *)a1 + 84, 0x10000LL)
    a1[15] = v2[3];
    a1[14] = v2[2];
    v3 = *v2;
    a1[13] = v2[1];
    a1[12] = v3;
    return os_unfair_lock_unlock((char *)a1 + 84);
}

unsigned __int64 __fastcall _xpc_mach_msg_get_audit_token(__int6
{
    return a1 + ((*(unsigned int *)(a1 + 4) + 3LL) & 0xFFFFFFFF
}
```

From here we can search for all the callers of `_xpc_connection_set_creds` and find the following functions:

- `_xpc_connection_mach_event`
- `_xpc_connection_unpack_message`

The `_xpc_connection_mach_event` function is called from two other functions: `_xpc_connection_create` and `_xpc_connection_set_event_handler2`. In both cases the

functions that use `_xpc_connection_mach_event` pass it into a `libdispatch` event handler for handling Mach messages. `_xpc_connection_unpack_message` itself is called from `_xpc_connection_mach_event` as well as two other XPC message sending functions.

Based on what we know about the kernel implementation and the disassembly and function names above it's safe to say that XPC is retrieving the `audit_token_t` from the mach message trailer for each message that it receives. It will then call `_xpc_connection_set_creds` passing in the `xpc_connection_t` structure as well as the mach message to ensure that the `audit_token_t` field of the `xpc_connection_t` is up to date.

We can verify this through an XPC test application by calling `xpc_connection_get_audit_token` in our server process when the client first connects, then have the client call `setuid` (which will force the kernel to update the `task`'s `audit_token`) and then send a message to the server. The server can then call `xpc_connection_get_audit_token` again and see that the `audit_token_t` values have been updated.

Conclusion

After diving into all the details of what an `audit_token_t` is, and how it gets used, I think it's important to take a step

back. We started this journey by discussing an XPC service best practice of using the `audit_token_t` rather than the PID to verify the code signing of an incoming connection. With the knowledge of all the internals we can now more clearly say why this is more secure. The `audit_token_t` contains not just the `p_pid` of the incoming process but also the `p_idversion`. The `p_idversion` is what allows the `csops_audittoken` system call to make sure that the process we're checking code signing on actually matches what we expect. It prevents the race condition of the `p_pid` looping back around and the server blindly trusting that it's the same process.