Search

Don't play with fire, as well as race condition

int80 January 1, 2021 No Comments

This is my first blog in 2021. I wish all of you have a wonderful new year.

Background

At Black hat USA 2019, we introduced a socket Use-After-Free (UAF) vulnerability caused by bad locking in the UNIX socket `bind` function on iOS. Briefly speaking, the function `unp_bind` temporarily unlocks the socket while binding the socket to a vnode, leading to a race condition. As a result, we can bind one socket to two vnodes. When the socket is closed and freed, one of the two vnodes still keeps a dangling pointer pointing to the freed socket object. By manipulating the vnodes again, we can trigger the socket UAF in the kernel. For more details, please refer to 1.

`bind` and `connect` are two basic interfaces for a socket, and have the same parameters.

```
int
     connect(int socket, const struct sockaddr *address, socklen_t address_len);
int
     bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

If the `bind` function is buggy, how about the `connect` function?

Socket Programming 101

A UNIX domain socket can be either connection-oriented (type `SOCK_STREAM`) or connectionless (type `SOCK_DGRAM`). In the case of connection-oriented scenario, we can setup a server socket, bind it to a local file address, and then listen and accept new connections. Here, `bind()` assigns a unique name to an unnamed socket.

According to the local file name, we can connect a client socket to the server socket. If it succeeds, a new socket is inserted into the server socket's connection queue. The server can then call `accept()` to extract the first connection request on the queue of pending connections, create a new socket with the same properties of socket, and allocate a new file descriptor for the new socket.

The Vulnerability

A client socket is not supposed to connect to different servers at the same time. However, let's take a look at the function `unp_connect`. I copied and pasted the source code of `unp_connect` as follows.

```
static int
unp_connect(struct socket *so, struct sockaddr *nam, __unused proc_t p)
{
        ...
        socket_unlock(so, 0);  <<--- a. temporary unlock so

        NDINIT(&nd, LOOKUP, OP_LOOKUP, FOLLOW | LOCKLEAF, UIO_SYSSPACE,
            CAST_USER_ADDR_T(buf), ctx);
        error = namei(&nd);     <<--- b. lookup the address
        if (error) {
                socket_lock(so, 0);
                return error;
        }
        nameidone(&nd);
        vp = nd.ni_vp;
        if (vp->v_type != VSOCK) {
                error = ENOTSOCK;
                socket_lock(so, 0);
                goto out;
        }
        ...
        socket_lock(vp->v_socket, 1); /* Get a reference on the listening socket
        so2 = vp->v_socket;
        ...
        if (so < so2) {
                socket_unlock(so2, 0);
                socket_lock(so, 0);  <<--- c. relock the sockets
```

```
                socket_lock(so2, 0);
        } else if (so > so2) {

                socket_lock(so, 0);
        }
        /*
         * Check if socket was connected while we were trying to
         * get the socket locks in order.
         * XXX - probably shouldn't return an error for SOCK_DGRAM
         */
        if ((so->so_state & SS_ISCONNECTED) != 0) {
                error = EISCONN;
                goto decref_out;
        }


        ...
                socket_unlock(so, 0); <<--- d. temporary unlock so

                if ((so2->so_options & SO_ACCEPTCONN) == 0 ||
                    (so3 = sonewconn(so2, 0, nam)) == 0) {   <<---  e. make a ne
                        error = ECONNREFUSED;
                        if (so != so2) {
                                socket_unlock(so2, 1);
                                socket_lock(so, 0);
                        } else {
                                socket_lock(so, 0);
                                /* Release the reference held for
                                 * listen socket.
                                 */
                                VERIFY(so2->so_usecount > 0);
                                so2->so_usecount--;
                        }
                        goto out;
                }
                ...
                        if (so < so2) {
                        socket_unlock(so2, 0);
                        socket_lock(so, 0); <<--- f. relock
                        socket_lock(so2, 0);
                } else {
                        socket_lock(so, 0);
                }

                /* Check again if the socket state changed when its lock was rel
```

```
                if ((so->so_state & SS_ISCONNECTED) != 0) {
                        error = EISCONN;

                        socket_unlock(so2, 1);
                        socket_lock(so3, 0);
                        sofreelastref(so3, 1);  <<-- g. free the new conn
                        goto out;
                }
...

        error = unp_connect2(so, so2);
        ...
}
```

Sorry for the long code snippet. At the first glance, you may have noticed that `unp_connect` performs socket locks and unlocks for multiple times. This is a strong indicator for a race condition. However, if you read the comments in the function, you will find that the developers have realized the potential race conditions. Every time the socket is re-locked, `unp_connect` performs checks on any change of the socket state. For example, after the lock at **(c)**, we can find the following comments:

```
         * Check if socket was connected while we were trying to
         * get the socket locks in order.
         * XXX - probably shouldn't return an error for SOCK_DGRAM
```

Another example: after the lock at **(f)**, we can find the comments:

```
 /* Check again if the socket state changed when its lock was released */
```

Playing with race condition is dangerous. In the case of `unp_connect` , the vulnerability occurs after a race condition is detected. Since I don’t want to write a long blog in holiday, let’s go to the vulnerability directly.

```
                if ((so2->so_options & SO_ACCEPTCONN) == 0 ||
                    (so3 = sonewconn(so2, 0, nam)) == 0) {   <<---  e. make a ne
                ...
                /* Check again if the socket state changed when its lock was rel
                if ((so->so_state & SS_ISCONNECTED) != 0) {
                        error = EISCONN;
                        socket_unlock(so2, 1);
                        socket_lock(so3, 0);
```

```
                              sofreelastref(so3, 1);  <<-- g. free the new conn
```

For the code above, `so3` is a newly created socket from the server socket through the function `sonewconn`. `unp_connect` temporarily unlocks the client socket while performing `sonewconn`, and relocks the client socket. If the client socket's state is changed to `SS_ISCONNECTED`, which implies that the client socket is connected to somewhere else during the temporary unlock, `unp_connect` just returns `EISCONN` and frees `so3`.

Let's focus on the following two lines:

```
                              socket_lock(so3, 0);
                              sofreelastref(so3, 1);
```

Clearly, `so3` is locked through the function `socket_lock`, and passed into function `sofreelastref`. Would `sofreelastref` really and directly free `so3`? No!

```
 void
sofreelastref(struct socket *so, int dealloc)
{
        struct socket *head = so->so_head;

        /* Assume socket is locked */

        if (!(so->so_flags & SOF_PCBCLEARING) || !(so->so_state & SS_NOFDREF)) {
                selthreadclear(&so->so_snd.sb_sel);
                selthreadclear(&so->so_rcv.sb_sel);
                so->so_rcv.sb_flags &= ~(SB_SEL | SB_UPCALL);
                so->so_snd.sb_flags &= ~(SB_SEL | SB_UPCALL);
                so->so_event = sonullevent;
                return;
        }
        ...
```

If a socket's `so_flags` has no `SOF_PCBCLEARING` or `SS_NOFDREF` set, `sofreelastref` does not deallocate the socket. For a newly created socket so3, does it have `SOF_PCBCLEARING` or `SS_NOFDREF` set? Still, no!

Now what we have is that, `so3` is locked, but not freed. The question is where `so3` is? In fact, `so3` is inserted into the server socket `so_incomp` list. Let's go back to the function `sonewconn`.

```
/*
 * When an attempt at a new connection is noted on a socket
 * which accepts connections, sonewconn is called.  If the
 * connection is possible (subject to space constraints, etc.)
 * then we allocate a new structure, propoerly linked into the
 * data structure of the original socket, and return this.
 * Connstatus may be 0, or SO_ISCONFIRMING, or SO_ISCONNECTED.
 */
static struct socket *
sonewconn_internal(struct socket *head, int connstatus)
{
...
  so = soalloc(1, SOCK_DOM(head), head->so_type);
        if (so == NULL) {
                return (struct socket *)0;
        }
...
        /* Insert in head appropriate lists */
        so_acquire_accept_list(head, NULL);

        so->so_head = head;


...
        so->so_flags |= SOF_INCOMP_INPROGRESS;
...
                TAILQ_INSERT_TAIL(&head->so_incomp, so, so_list);
                so->so_state |= SS_INCOMP;
                head->so_incqlen++;
...
        so_release_accept_list(head);
...
```

It's clear now: `so3` is on the server socket's `so_incomp` list. When the server socket is closed, it is responsible to clean up the `so_incomp` list. The following code shows the function `soclose_locked`.

```
int
soclose_locked(struct socket *so)
{
...
                TAILQ_FOREACH_SAFE(sp, &so->so_incomp, so_list, sonext) {
```

```
                                ...
                                if (persocklock != 0) {
                                        socket_lock(sp, 1);
                                }
...
```

Have you pinpointed the issue? As we checked, `so3` is locked and inserted into to `so_incomp` list. However, when the function `soclose_locked` processes the `so_incomp` list, it would lock `sp` again. Actually, `sp` is our locked `so3` !

So far it sounds like a lock issue. Yes, the race condition in `unp_connect` now turns into a double lock issue. A socket object is passed into `socket_lock` twice. Does it cause any memory safety problem?

The Lock

The implementation of locks on iOS is very complicated, as least to me. The readers could check XNU source code for more details. In our case, `socket_lock` calls `unp_lock` , and eventually calls `lck_mtx_lock` and calls `lck_mtx_lock_contended` .

```
 unp_lock(struct socket *so, int refcount, void * lr)
{
...
        if (so->so_pcb) {
                lck_mtx_lock(&((struct unpcb *)so->so_pcb)->unp_mtx);
...

void
lck_mtx_lock(lck_mtx_t *lock)
{
        thread_t        thread;
...
        thread = current_thread();
...
        lck_mtx_lock_contended(lock, thread, FALSE);
}
```

If a lock is unlocked, `lck_mtx_lock_contended` will directly acquire the lock and set the ownership. The ownership is the current thread pointer. Otherwise, if the lock is locked, `lck_mtx_lock_contended` would try to loop waiting until the lock is released. During this process, `lck_mtx_lock_contended` will use the owner thread pointer for many reasons.

How to trigger thread_t UAF

Now we try to turn the double-lock issue into a thread_t UAF. The idea is as follows.

Now we try to turn the double-lock issue into a thread_t UAF. The idea is as follows.

We create two threads that try to connect the same client socket to two different server sockets. If `unp_connect` catches the race condition, it may create a new `so3`, insert it to the corresponding server socket's `so_incomp` list, and then lock `so3` that stores the `thread_t` pointer of the corresponding thread.

And then, we terminate the two threads, as a result, the two `thread_t` objects are deallocated in the kernel. However, the `so3` still keeps a dangling `thread_t` pointer in its lock object.

Now we close all the server sockets, which will trigger the cleanup of the `so_incomp` list of the server sockets. As a result, the kernel will run `socket_lock(so3)` again. Accessing to the owner thread of `so3`'s lock will trigger the `thread_t` UAF problem.

For a complete POC, please check 2.

Conclusion

We shared a thread_t UAF problem in the XNU kernel. We analyzed how a failed race condition turns into a double-lock issue, and then turns into a UAF issue. Hope you enjoy the blog. Thank you for reading.

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment