sock_port – 11.0-12.2 tfp0

Back in May iOS 12.3 got released, and along with it, many security bugs were patched. One of them was from Ned Willamson of Google Project Zero, CVE-2019-8605, a UAF bug in socket handling, reachable from the sandbox. A few months later, and this bug gets developed into a full tfp0 exploit, SockPuppet. The exploit was good and reliable, but not all devices worked properly with it. Specifically, the exploit was broken on the ones with a 4K page size. It was also broken on iOS 11.

Due to this reason and a few other facts (that the code was kinda bloated and hard to read) I decided to give my own shot at an exploit for this bug, and that’s how sock_port came to live, an 11.0-12.2 exploit supporting non-SMAP (A7-A9) devices (later SMAP support was added).

First let’s look at the bug. Williamson gives this code snippet:

```
void in6_pcbdetach(struct inpcb *inp) {
  // ...
  if (!(so->so_flags & SOF_PCBCLEARING)) {
      struct ip_moptions *imo;
      struct ip6_moptions *im6o;

      inp->inp_vflag = 0;
      if (inp->in6p_options != NULL) {
          m_freem(inp->in6p_options);
          inp->in6p_options = NULL; // <- good
      }
      ip6_freepcbopts(inp->in6p_outputopts); // <- bad
      ROUTE_RELEASE(&inp->in6p_route);
      // free IPv4 related resources in case of mapped addr
      if (inp->inp_options != NULL) {
          (void) m_free(inp->inp_options); // <- good
          inp->inp_options = NULL;
      }
      ...
```

The problem here is that `inp->in6p_outputopts` gets freed but not nulled, which allows the pointer to be reused. Williamson further explains that we can reach this condition if we disconnect from a socket without destroying it. From within the sandbox, the following PoC applies:

```
while (1) {
    int s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);

    // Permit setsockopt after disconnecting (and freeing socket options)
    struct so_np_extensions sonpx = {.npx_flags = SONPX_SETOPTSHUT, .npx_mask =
SONPX_SETOPTSHUT};
    setsockopt(s, SOL_SOCKET, SO_NP_EXTENSIONS, &sonpx, sizeof(sonpx));

    int minmtu = -1;
    setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu)); //
allocate in6p_outputopts
    disconnectx(s, 0, 0); // free in6p_outputops
    setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu)); //
UAF

    close(s);
}
```

Now that we know the bug and how to trigger it, let's move on to exploitation. This is a UAF bug, so the technique known as "heap spraying" takes place. An UAFed object has been freed, and what's the difference between freed and non-freed memory? Freed memory can be reused, because well, it's free. There is a chance that the next allocation will end up at the same address where our freed object once was, this way we can control the data of that object. The questions that now pop up are: How do you tell the kernel to allocate memory? How do you make sure it ends up in the same address? What data do you put? "Heap spraying" is the answer to the second question, since we can't control where data goes, we just make a lot of allocations and wait for one of them to get where we want, how we check if it did and whether we can really depends on what object we have UAFed, but in this case, yes you can and it's pretty straightforward. Now, how do you get the data in there? There are multiple methods to do this, such as IOSurface or mach messages. (note that kernel memory is organized in zones depending on allocation size, so the allocation we make must be the same size as the UAFed object for it to end up on the same address. More on this later). IOSurface is a kernel extension used in graphics, it is reachable from the sandbox and offers methods which we can call to get data of a custom size into the kernel. Mach messages instead are used for communication between processes, but since the kernel manages all processes, they send data into the kernel first, so we can use this method for sending data to the kernel as well.

Now that we know what methods we can use for heap-spraying, let's see what can we achieve by controlling `inp->in6p_outputopts`, the problematic object in our case. First let's first find out what `inp->in6p_outputopts` actually is. By searching backwards through XNU code (try to do that yourself! Use this tool by Jonathan Levin), we can come to the conlusion that it's of type "struct ip6_pktopts". And if you're interested in doing some debugging, here's how to get there from your proc struct (info on this on my rootlessJB write-up):

our proc struct -> struct filedesc (named p_fd, offset = 0x100 or 0x108 on iOS 11) -> struct fileproc array (named fd_ofiles, offset = 0) -> element with your socket file descriptor as an index -> struct fileglob (named f_fglob, offset = 8) -> struct socket (named fg_data, offset = 56) -> struct inpcb (named so_pcb, offset = 16) -> struct ip6_pktopts (named inp6_outputopts, offset = 304).

Now, let's take a look at that "struct ip6_pktopts". Here's its definition:

```
struct ip6_pktopts {
    struct mbuf *ip6po_m;
    int ip6po_hlim;
    struct in6_pktinfo *ip6po_pktinfo;
    struct ip6po_nhinfo ip6po_nhinfo;
    struct ip6_hbh *ip6po_hbh;
    struct ip6_dest *ip6po_dest1;
    struct ip6po_rhinfo ip6po_rhinfo;
    struct ip6_dest *ip6po_dest2;
    int ip6po_tclass;
    int ip6po_minmtu;
    int ip6po_prefer_tempaddr;
    int ip6po_flags;
};
```

From here, we can control options using get/setsockopt, options such as `ip6po_pktinfo`, `ip6po_tclass`, `ip6po_minmtu`, `ip6po_prefer_tempaddr`. Remember, you can free this struct whenever you want using the vulnerability, and allocate it with whatever contents you want using heap-spraying. One thing we might do first is leaking data.

Firstly we can read integers like minmtu or prefer_tempaddr. By doing this we can leak the kernel address of our task port, as to why this is useful, that will be explained later in the write-up. How do we do that? async_wake contained a useful technique, a function called fill_kalloc_with_port_pointer(). This function uses mach messages to do an allocation on the kernel, filled with the kernel address of a port a specified number of times. This is done using ool messages (out-of-line, the opposite of in-line. What this means is that we don't send data, but the address of the data) sending send rights to an array with a specified number of ports. The size of the struct we want to attack is 192 bytes (if you do the calculation or sizeof()), 192 bytes divided by the size of one pointer (8 bytes), is 24, so we use fill_kalloc_with_port_pointer() to do a 192 byte allocation filled with the address of our port 24 times.

In the find_port_via_uaf() function, we first get a new socket whose options structure has been freed (thus ready for reallocation), then we call fill_kalloc_with_port_pointer() a lot of times, until our structure gets reallocated by the mach message. In order to know if it worked, we simply read two consecutive integers, merge them into one 64bit integer, and check if result is a kernel pointer. After reallocation, the structure will look like this:

```
XX XX XX XX F0 FF FF FF  XX XX XX XX F0 FF FF FF  |  ................
XX XX XX XX F0 FF FF FF  XX XX XX XX F0 FF FF FF  |  ................
... and so on, with 24 pointers ...
XX XX XX XX F0 FF FF FF  XX XX XX XX F0 FF FF FF  |  ................
```

In this case, I read minmtu & prefer_tempaddr (just like SockPuppet did). This was our first primitive!

Now by looking at the structure further, we see we can control pointers, and one of them we can read back from! That is `ip6po_pktinfo`. By reallocating the structure with this pointer faked, we can read back from it 20 bytes (sizeof(struct in6_pktinfo)), thus achieving a kernel read primitive.

In the read_20_via_uaf() function first we get a bunch of sockets with freed options (128 in this case), then we construct a fake struct in6_pktinfo, set its minmtu field to something we can use as an identifier of this fake struct, then make the reallocation. This time for the reallocation, we use IOSurface as this is an easier way to get an allocation with any size and data we want. After reallocation, we iterate over our sockets until we find the one with our specified minmtu value, and read back the fake struct.
Here goes our second primitive, done! The first two primitives are also used in SockPuppet but with slightly different mechanics, and that's where I got the ideas from. From now on, exploitation is different from SockPuppet.

Now, I thought I could use the read primitive as a write one as well, you can't just read the in6_pktinfo struct, you can also write to it. Unfortunately it appeared that the kernel had input checks, so you cannot specify an arbitrary structure in setsockopt(). But, we can still write something right? Turned out to be so! Looking at the code in XNU that does this, it appears you can write a bunch of 0s. But how is this any useful? Reminding ourselves what tfp0 exploits target, that is getting controlled data over a mach port we have a send right to, so that we can turn that mach port into the kernel task port. And recently a bunch of exploits have done this via mach port UAF bugs. So, how is this related? Mach ports use reference counting. The second member of an ipc_port structure is it's reference count number. If we can leak port addresses, and if we can write a bunch of 0s into a known address, how about making the reference count zero? That will cause the port to get freed, turning this UAF into mach port UAF! Once we UAF a port, we can continue with older techniques implemented into the recent exploits. One thing to note however is that the structure can't be completely filled with 0s, instead the last field of it needs to be set to 1. Otherwise, the kernel will free the address instead of writing 0s to it (while this still works for our purpose, for me it caused a panic because, thinking I just wrote 0s, I called mach_port_insert_right() so the kernel would see the updated references, but in fact it panicked because the port already was freed. We could use this for a better exploitation method, but I didn't realize that so I just chose to continue. The other method, also used in SockPuppet will be discussed later)

So, we turned this struct ip6_pktopts UAF into a mach port UAF. Now we can proceed with reallocating the port with controlled data. Now, since ports have their own zone, we cannot use any technique to get controlled data in there, neither IOSurface nor mach messages. We don't just have to free the port, we need to free the whole page (a page is just a block of memory, on A7-A8 it's 4 KB, on A8X+ it's 16 KB) it stays on (what's called garbage collection), and in order to do this, all elements on that page have to be freed as well. Since if we just allocate a port, the chances are it will land in the middle of a page with other ports in it, we start allocating a lot of mach ports, hopefully landing sooner or later in a fresh page, then we allocate our target port, then a bunch of other ports as well to ensure that no other port gets allocated in that page. If this goes well, the entire fresh page will be filled with ports we control. We can now free the 'before' and 'after' ports normally and do our UAF on the target port. This will free the whole page, making it ready for garbage collection, so that the page can be reallocated in another zone. To trigger garbage collection, we just make a lot of kernel allocations either with IOSurface or mach messages. When allocations take longer than 1 millisecond, we can stop,

assume garbage collection is happening and sleep for one second to ensure it ends.
After this, we can now use IOSurface again to send controlled data over a port. In order for this port to behave like tfp0, we need a few kernel addresses, firstly the kernel's vm_map, secondly ipc_space_kernel. Since the first takes a lot of kernel reads to get found and our kernel read primitive is slow and not 100% reliable, we find only ipc_space_kernel using our first primitive (we do this by reading from our task port address which we have leaked! All task ports have this address). For vm_map we can build another primitive using our fake port and pid_for_task().
Here's the plan:

- Allocate a fake port (in userland)
- Set its io_bits as `IO_BITS_ACTIVE | IKOT_TASK` to mark this port as a task port
- Send our fake port to the kernel via IOSurface
- Create a fake task struct (again, in userland)
- Set the fake task pointer in the port's ip_kobject (this is why the exploit does not work on A10+! The fake task is a userland pointer, which the kernel cannot normally dereference in newer devices because of SMAP!)

Since we haven't set vm_map in our fake task, this port is not a valid tfp0 port, however there's a function which can help us with kernel reading, pid_for_task(). Pass a task port to this function and it will read from fake_task->bsd_info->p_pid, we control fake_task from userland, so we control the bsd_info pointer, we can point fake_task->bsd_info to "target_address – offsetof_p_pid", that way, when pid_for_task reads from "bsd_info + offsetof_p_pid", it actually reads from "target_address – offsetof_p_pid + offsetof_p_pid", cancel offsets out and you get "target_address"! Here's a better read primitive! Since a pid is 32 bits, this will also read 32 bits, to read 64 bits simply make two 32 bit reads and merge them.

Now we can find the vm_map of the kernel, this is done through process iteration, for more technical info look at the code. After we do this, we update our fake task (remember, it's in userland so we can control it directly) by doing:

```
fake_task->map = kernel_vm_map;
```

That same moment, our fake port is now a valid tfp0 port! Now since our port is a result of an IOSurface kernel allocation, we will want to do a clean up. So we create another port, find its address (this time using our tfp0 port, and not our original primitive), and copy over the fake port & fake task. We can now destroy our previous port & deinitialize IOSurface. (After some time, I changed the code that did the port reallocation with some code fro, async_wake, which made the exploit more

reliable, go through the commits to see the old code)
Exploit done! Now, how about SMAP?

In order to support SMAP devices, we need to avoid using userland pointers. SMAP is a hardware-based security measure present on A10 chips and newer, which prevents the kernel from reading userland memory unless special code paths are used (i.e. the copyin() function which reads data from userland). In this exploit, fake_task is located in userland, it could easily stay in kernel memory but we need to dynamically update it after we get kernel read. As stated previously using the initial kernel read primitive is not viable as we need to do a lot of kernel reads. We could as well just do the UAF of the port + spraying, once again after finding the kernel's vm_map, but this would make the reliability of the exploit worse. We need a way to update fake_task while staying on kernel memory. What's a good way to do that? Pipe buffers! Pipes are used as a one-way communication method to send data using file descriptors. A pipe is a pair of file descriptors, one for reading and one for writing. When we write into a pipe, the kernel allocates a buffer on the reading descriptor, writes there and sets the buffer position at the end of it. When we read from the reading descriptor, we get data back from that buffer, and the kernel sets the position back at the beginning, since data has been read. Here's what happens:

pipe() -> create two file descriptors, a reading one and a writing one
read() -> look up the read descriptor, read data from the buffer starting from the beginning of the buffer until we reach the current buffer position **or** until we have read 'n' specified bytes. If we reach the buffer position move it back at the beginning
write() -> look up the corrensponding read descriptor, write data on the buffer starting from the current position until we have written 'n' specified bytes, and move the position right as we write. If the buffer is not big enough to hold the data, reallocate it

With basic understanding of pipes, we now can use this to have a controlled buffer on the kernel. We could use the pipe buffer as our fake task!

We start with this code:

```
	uint8_t buf[0x600]; // create a buffer
	memset(buf, 0, 0x600); // fill with 0s initially
	write(fds[1], buf, 0x600); // since we need 0x600 bytes for our fake task,
write that much so kernel allocates a buffer capable of containing the fake task
	read(fds[0], buf, 0x600); // reset the buffer position by reading the data
back
```

Next we need to find where the pipe buffer is located on the kernel, this is similar to how we can find the socket structures:

```
      uint64_t task = rk64_check(self_port_addr + 
koffset(KSTRUCT_OFFSET_IPC_PORT_IP_KOBJECT));
      uint64_t proc = rk64_check(task + koffset(KSTRUCT_OFFSET_TASK_BSD_INFO));
      uint64_t p_fd = rk64_check(proc + koffset(KSTRUCT_OFFSET_PROC_P_FD));
      uint64_t fd_ofiles = rk64_check(p_fd);
      uint64_t fproc = rk64_check(fd_ofiles + fds[0] * 8);
      uint64_t f_fglob = rk64_check(fproc + 8);
      uint64_t fg_data = rk64_check(f_fglob + 56);
      pipe_buffer = rk64_check(fg_data + 16);
```

Next instead of using the userland fake_task pointer, we do this:

```
      fakeport->ip_kobject = pipe_buffer;
      write(fds[1], fake_task, 0x600); // write fake task, fds[1] is the writing 
descriptor
```

And in order to kernel read with our fake port, we must also update the buffer on the kernel:

```
      read(fds[0], fake_task, 0x600);\ // reset buffer position
      *read_addr_ptr = addr - koffset(KSTRUCT_OFFSET_PROC_PID);\ // update the 
fake task here in userland
      write(fds[1], fake_task, 0x600);\ // write it to the pipe buffer
      value = 0x0;\
      ret = pid_for_task(first_port, (int *)&value); // continue with 
pid_for_task() trick
```

Now that we can control fake_task from userland while it still stays on the kernel, exploit will work on SMAP devices!

Now, what about the better method I mentioned above?

Remember when I said “*the kernel will free the address instead of writing 0s to it (while this still works for our purpose, for me it caused a panic because, thinking I just wrote 0s, I called mach_port_insert_right() so the kernel saw the updated references, but it fact it panicked because port already was freed. We could use this for a better exploitation method, but I didn’t realize that so I just chose to continue…*”

I didn’t realize how useful this was until I read the source of SockPuppet once again. Setting a completely nulled pktinfo struct, would cause it to be freed. And this can be verified if we look at the code in XNU:

```
if (optname == IPV6_PKTINFO && opt->ip6po_pktinfo &&
     pktinfo->ipi6_ifindex == 0 &&
     IN6_IS_ADDR_UNSPECIFIED(&pktinfo->ipi6_addr)) {
    ip6_clearpktopts(opt, optname);
    break;
}
```

`IN6_IS_ADDR_UNSPECIFIED` here just checks if everything is zero, while `ip6_clearpktopts()` does this (along with some other checks, but we’re not interested):

```
if (optname == -1 || optname == IPV6_PKTINFO) {
      if (pktopt->ip6po_pktinfo)
           FREE(pktopt->ip6po_pktinfo, M_IP6OPT);
      pktopt->ip6po_pktinfo = NULL;
}
```

It frees the pktinfo struct, and if you try it, you’ll notice it automatically detects the freeing size (you cannot only free objects with the same size as pktinfo, you can free any object. In the kernel normally you need to provide the size when freeing)

Now why is this useful? What could we do with this that we couldn’t before? Williamson uses some very smart tricks, all of these were discussed in this write-up already, but not given much importance. Remember fill_kalloc_with_port_pointers? What did it do again? Send *send rights* of *any port*, any number of times, via an ool mach message. Also remember how pipes worked? They had a buffer which you completely controlled via read() and write(). The idea is as following:

1. We create a new pipe
2. We UAF its pipe buffer by setting pktinfo as detailed above
3. We use fill_kalloc_with_port_pointers to fill this area with addresses of a

dummy port

4. We use write() to swap one of those pointers with another one pointing to a fake port (in this case, we have to use pipes, this can't be on userland or else we get a panic)
5. We receive the mach message back, getting a send right to our fake port!

Note, how this does not involve UAFing a port, thus we don't need to trigger garbage collection, making this exploit way faster. And since we can use read() to check if the pipe buffer got patched, we don't rely on luck, we stop exactly when what we wanted happened, this also makes the exploit faster.

From here on, exploitation is the same, the fake task can either be in userland or we can use another pipe if we want this to work on SMAP devices.

For any questions, feel free to contact me at @Jakeashacks on Twitter.
Credits go to Ned Williamson for finding the bug and his tricks, Ian Beer for a few techniques, Brandon Azad for iosurface spraying functions, @IBSparkes for machswap which was used in the older implementations of sock_port.

sock_port: https://github.com/jakeajames/sock_port
sock_port 2 (with Ned Williamson's trick): https://github.com/jakeajames/sock_port/tree/sock_port_2