# Debug for Bug: Crack and Hack Apple Core by Itself

Technical Brief

by Lilang Wu and Moony Li

Almost every operating system (OS) now features different built-in tools and techniques for managing security vulnerabilities. Notable examples of these include control flow integrity (CFI) on Android 9 or pointer authentication codes (PAC) on iOS 12 hardware. Industry standard fuzzers like American fuzzy lop (AFL) and syzkaller are also being widely used.

Because of these developments, the bug hunting space left for security researchers seems to be much smaller. Code reviewing based on expert threat knowledge seems to be a path that researchers can take, but it is time consuming and takes much effort.

How do we break the deadlock? We developed a tool called LLDBFuzzer, a debug fuzzer for bug hunting, to help security researchers. This method is based on a next-generation debugger called Low Level Debugger (more popularly known as LLDB), from the LLVM Project. Based on our tests, it has proven to be an effective way to find and expand new attack interfaces, but it is also flexible, scalable, and scriptable for vulnerability research utilities. Moreover, we can demonstrate how to implement an LLDB debugger client within network extensions, which can help us fuzz within virtual machines to significantly improve efficiency.

We tested the LLDBFuzzer on a Mac Pro running the latest OS at the time of experimentation, and our target was Apple Graphic Drivers. Our fuzzing methodology found dozens of vulnerabilities, including double free and out-of-bounds (OOB) read/write bugs that we will cover in the vulnerability analysis portion below. We discuss six vulnerabilities, but these are only a part of what we found. The others will be analyzed later and submitted to Apple.

# 1. A look into LLDBFuzzer

## 1.1. Comparing different bug hunting methods to LLDBFuzzer

There are different methods used in bug hunting, and each has specific pros and cons. Some are only suitable for large-scale deployments, some hit the code coverage ceiling, and others cannot find new attack interfaces. We review the different methods, and compare them with LLDBFuzzer.

**Bug hunt method comparison**

| | Key method | Wait Time | Find new attack interface | Deep coverage |
|---|---|---|---|---|
| **Syzkaller/AFL** | Code coverage feedback | Long | No | No or unknown |
| **Code Review** | Personal knowledge | Unknown | Yes | Yes |
| **LLDBFuzzer** | Debug and taint | Short | Yes | Yes |

*Table 1. Typical Bug Hunt method comparison*

**Code review -** Code reviewing is usually a good way to find new attack interfaces and vulnerabilities hidden in deep locations, especially for logical vulnerabilities. However, this method is time consuming and its results are unpredictable.

**AFL & Syzkaller -** AFL is an open source fuzz-testing tool developed by Michał Zalewski, while syzkaller is a kernel fuzzer. They are based on code coverage feedback that mutate strategy and target modules accordingly. Typically, an AFL-like fuzzer would mutate the input file on the bit level or reassemble the grammar elements according to some syntax for user mod targets. Syzkaller would mutate the system calls according to function prototype towards kernel mode code.

AFL and Syzkaller are suitable for large-scale deployment. However, bug hunters will usually touch the code coverage ceiling — deep code location is difficult to reach for data dependency or code execution sequence dependency. They also can't help find new attack interfaces because fuzzing interfaces are typically configured by experts.

**LLDBFuzzer -** LLDBFuzzer is based on the built-in debug mechanism of operating systems that intercept and break the execution of key API or the instruction at key points (selected according your system and security knowledge), and fuzzes corresponding data or code in an execution context. Since most data or code dependencies are kept during fuzzing, the fuzz activity can touch a deeper code branch compared to the sykaller/AFL-like methods. And since we do not designate the execution channel of the fuzzing, hidden attack interfaces would be exposed because of deep interception.

**Interception method comparison**

| | System mode support | Scriptable | Control Grain | Execution control | Cross platform |
|---|---|---|---|---|---|
| **DTrace** | Kernel | Yes | API | No/View only | Easy |
| **Frida** | User | Yes | Instruction | Yes | Easy |
| **Inline hook** | Both | No | Instruction | Yes | Middle |
| **LLDBFuzzer** | Both | Yes | Instruction | Yes | Easy |

*Table 2. Typical interception method compare*

Here is a brief comparison of the interception method (for Apple systems, in this example), which explains why we choose the debug path:

DTrace and Frida are script based program execution tracing tools with well-documented interface APIs and good tracing capabilities at the API or instruction level. They are also good for cross platform development. However, we can disregard DTrace for its inability to modify the execution code and data at runtime. Frida is likely the best at user mode interception but not at kernel mode.

While inline hook is good for instruction level control, the obvious drawback is that it is too “raw” and will take too much development effort for utility infrastructure and cross platform reconstruction.

## 1.2. Kernel debugging and the LLDBFuzzer

**Kernel debugger overview**

MacOS supports two-machine kernel debugging using LLDB over an Ethernet or FireWire connection. The remote debugger protocol is called the Kernel Debugging Protocol (KDP).

**KDP protocol initialization process in XNU**

The KDP protocol is initialized during system bootstrap, as shown in Figure 1 below. During startup, the system creates a *kdp init* thread and implements a debugger trap. The *kdp init* thread is used to wait for Ethernet drivers registering send and receive handlers, while the debugger loop within the trap is responsible for polling, processing, and replying to the incoming debug command with those two handlers. What's more, XNU implements all the debugger command functions in the kdp.c file and registers them in a dispatch table; for example, `breakpoint set` command refers to the *kdp_breakpoint_set* function. These functions make up the debugger world.

**Kernel debugger mechanism within the Ethernet driver**

The debugger functions implemented within XNU are not enough. If the target machine supports a remote debugger, its Ethernet driver should implement the *IOKernelDebugger* service and its object interfaces with the kernel debugger protocol (KDP) module and dispatches KDP requests to its target (provider).

Figure 2 shows the support for remote debugging. The target, designated as the debugger device, must implement a pair of handler functions that are called to handle KDP transmit and receive requests during a debugging session. Only a single *IOKernelDebugger* in the system can be active at a given time. The active *IOKernelDebugger* is the one that has an IOKDP object attached as a client.

The debugger device is usually a subclass of *IOEthernetController*. However, any IOService can service an *IOKernelDebugger* client, implement the two polled mode handlers, and transport the KDP packets through a data channel. However, KDP assumes that the debugger device is an Ethernet interface and therefore it will always send, and expect to receive, an Ethernet frame.

Figure 3 shows the architecture of KDP debugger implementation in Drivers. From the figure, we can see that the subclass of *IOEthernetController* implements the receive and send handlers, and *IOKernelDebugger* registers these two handlers into XNU. Therefore, remote devices can operate the debugger command on the target machine.

For FireWire debugging, KDP is used over a FireWire cable courtesy of a kernel extension (AppleFireWireKDP.kext) on the target machine and a translator program (FireWireKDPProxy) on the debugger machine. The translator routes data between the FireWire connection and UDP port 41139 on the debugger system, and it acts as a local proxy for the target machine. LLDB still performs network-based debugging, except that it communicates with localhost instead of directly communicating with the shim on the target machine.

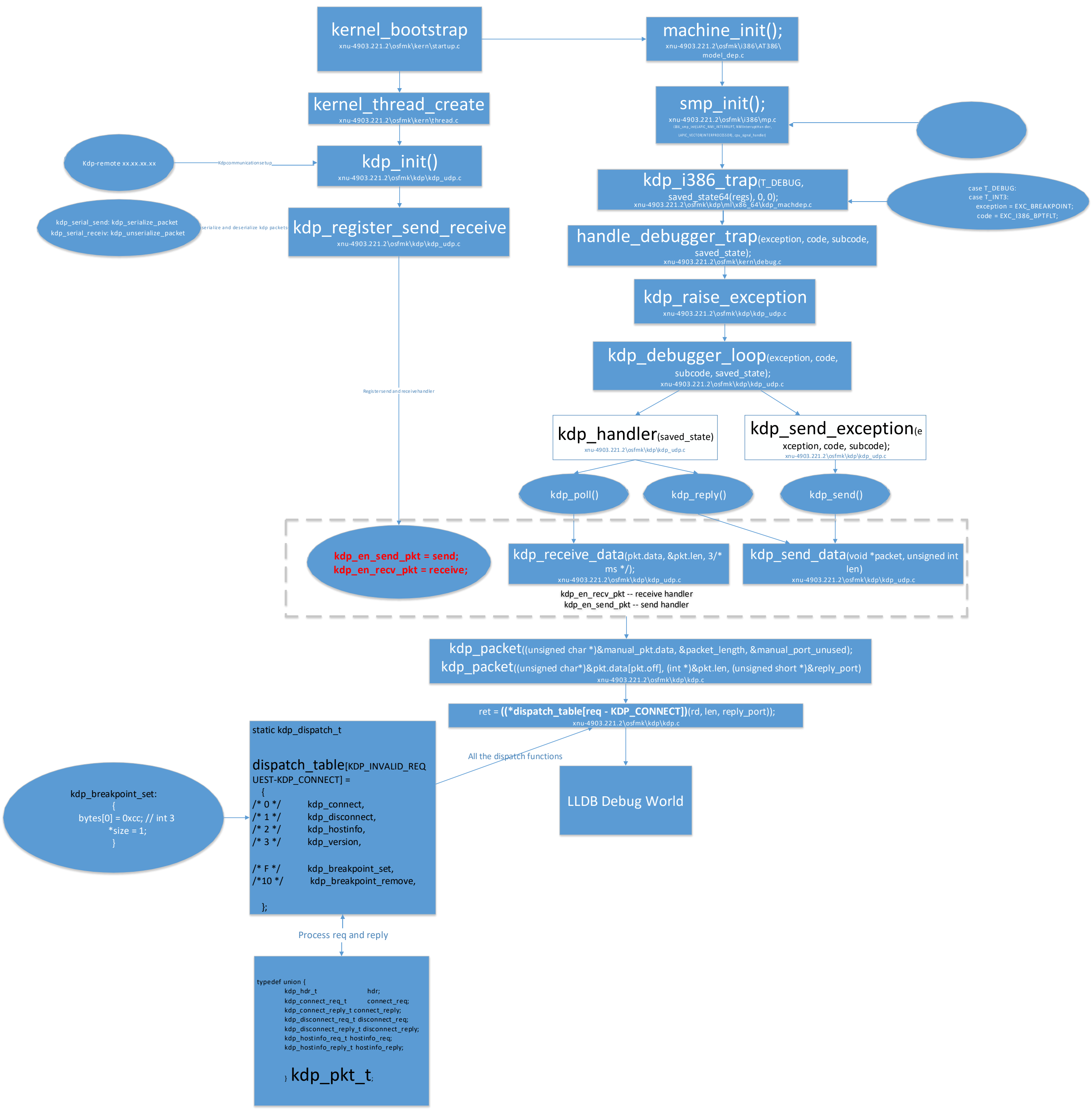



*Figure 1. KDP protocol init process during kernel bootstrap*

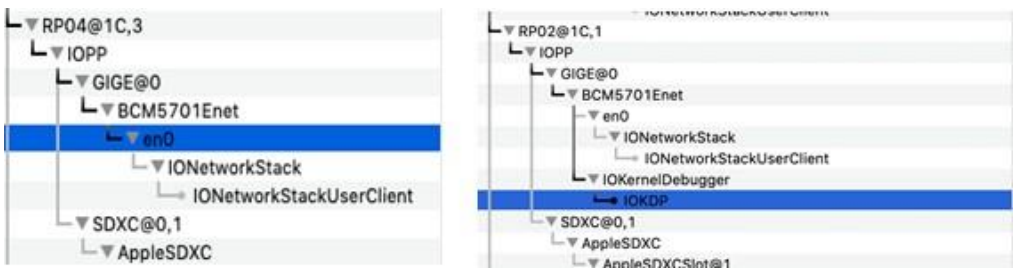



*Figure 2. Drivers that support remote debugging*

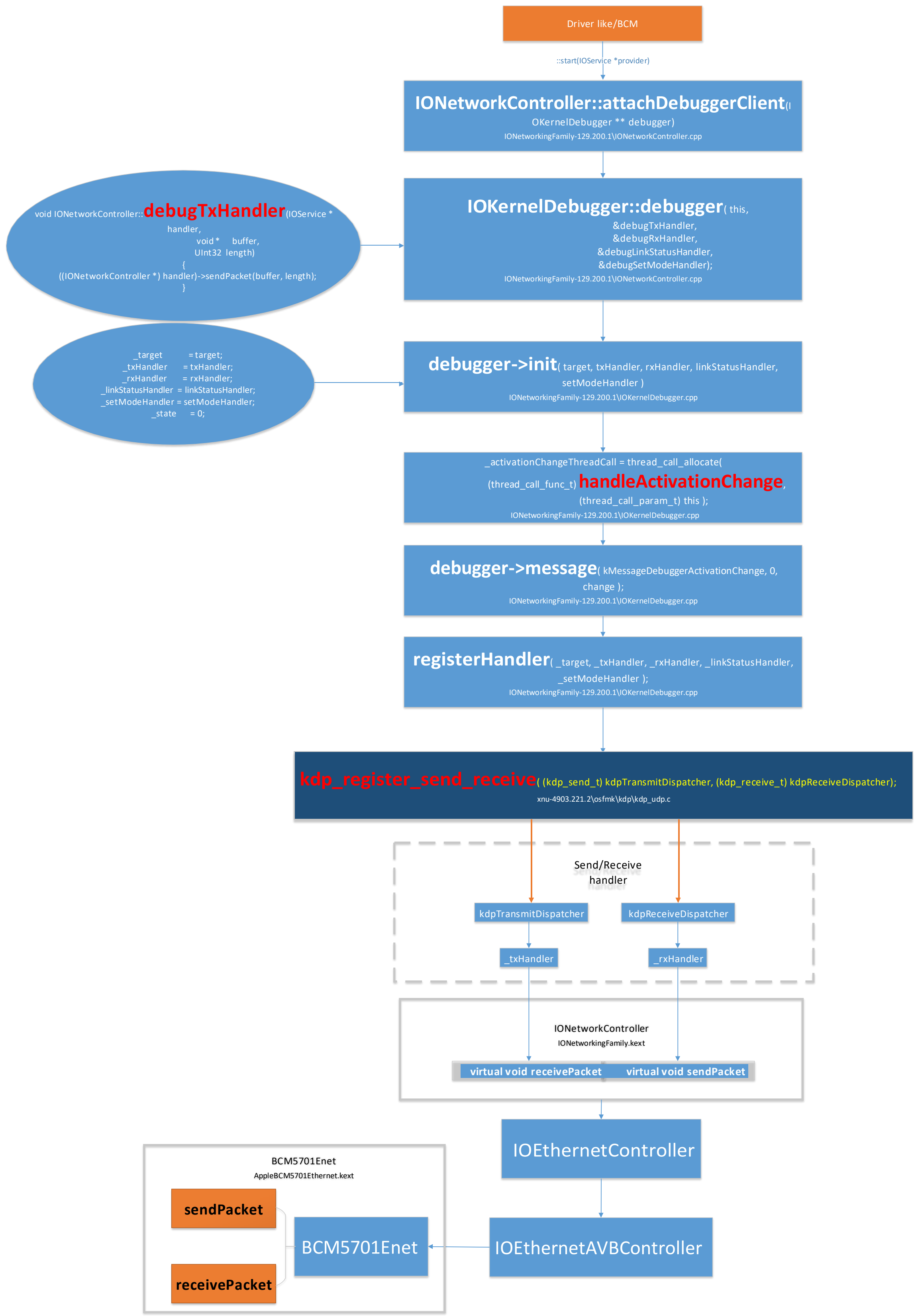



*Figure 3. The architecture of KDP debugger implementation in Drivers*

**Debugger Toolset available for MacOS**

Apple also provides some debug scripts that support kernel debugging, as shown in Figure 4.

```
|------------ xnu scripts -------------|
|   |- lldb Command/Scripting-|   |   <- provides scriptability for kernel data structures through summary/command invocation.
|   |       |--lldb core--|      |  |   <- interacts with remote kernel or corefile.
|   |-------------------------------|   |
|--------------------------------------|
```

*Figure 4. XNU debug scripts provided by Apple*

```
xnu/
 |-tools/
    |-lldbmacros/
       |-core/          # Core logic about kernel, lldb value abstraction, configs etc.
       |-plugins/       # Holds plugins for kernel commands.
       |-xnu.py         # xnu debug framework along with kgmhelp, xnudebug commands.
       |-xnudefines.py
       |-utils.py
       |-process.py     # files containing commands/summaries code for each subsystem
       |-...
```

*Figure 5. The XNU debug script file layout*

The Core directory provides many basic components used in the debugger process, such as API wrappers that encapsulate the basic LLDB Scripting Bridge APIs. The plugins directory contains a plugin that can create performance reports for zprint output. The xun.py file includes the LLDB initialization code, which is used to load plugins and additional debug commands. The process.py script mainly contains the debug commands implementation code.

**Kernel Debug Process**

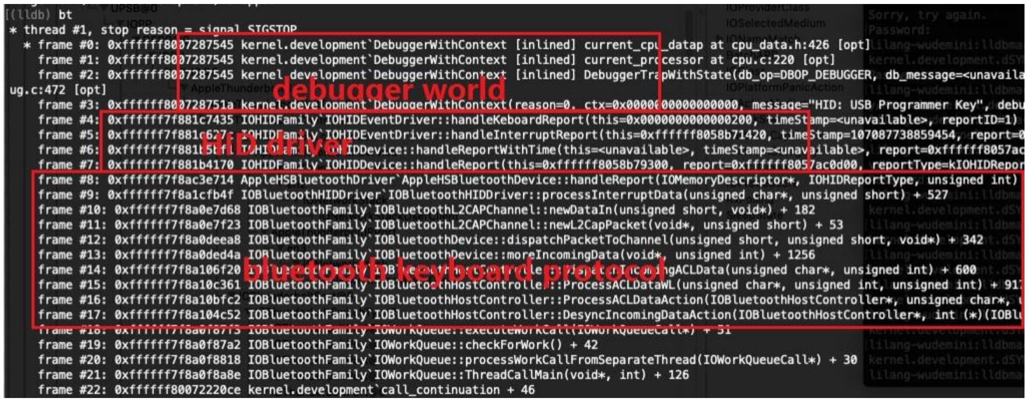



*Figure 6. Back trace after using NMI interruption*

Figure 6 shows the back trace after using NMI interruption. For remote kernel debug, a NMI (Command-Option-Control-Shift-Escape) signal can be manually generated to interrupt the target machine during execution, which gives an opportunity for the remote debugger to connect. However, the configuration to enable the debugger and how to debug a remote device will not be introduced here.

**LLDBFuzzer overview**

Although LLDB is not suitable for debugging low-level kernel components, it can debug almost all the kernel extensions and XNU codes after the required hardware is operational. Based on these features, we introduce a novel fuzzing architecture we call LLDBFuzzer.

**The LLDBFuzzer architecture**

Figure 7 shows the architecture of our LLDB fuzz solution. As mentioned previously, this solution is based on the remote kernel debugger system, so our fuzz solution contains two machines. One is the remote machine, which runs our main fuzzing logic; and the other is the target machine, which is loaded with a custom kernel and deploys our fuzz point. The target machine can be a MacOS VM or a real device.

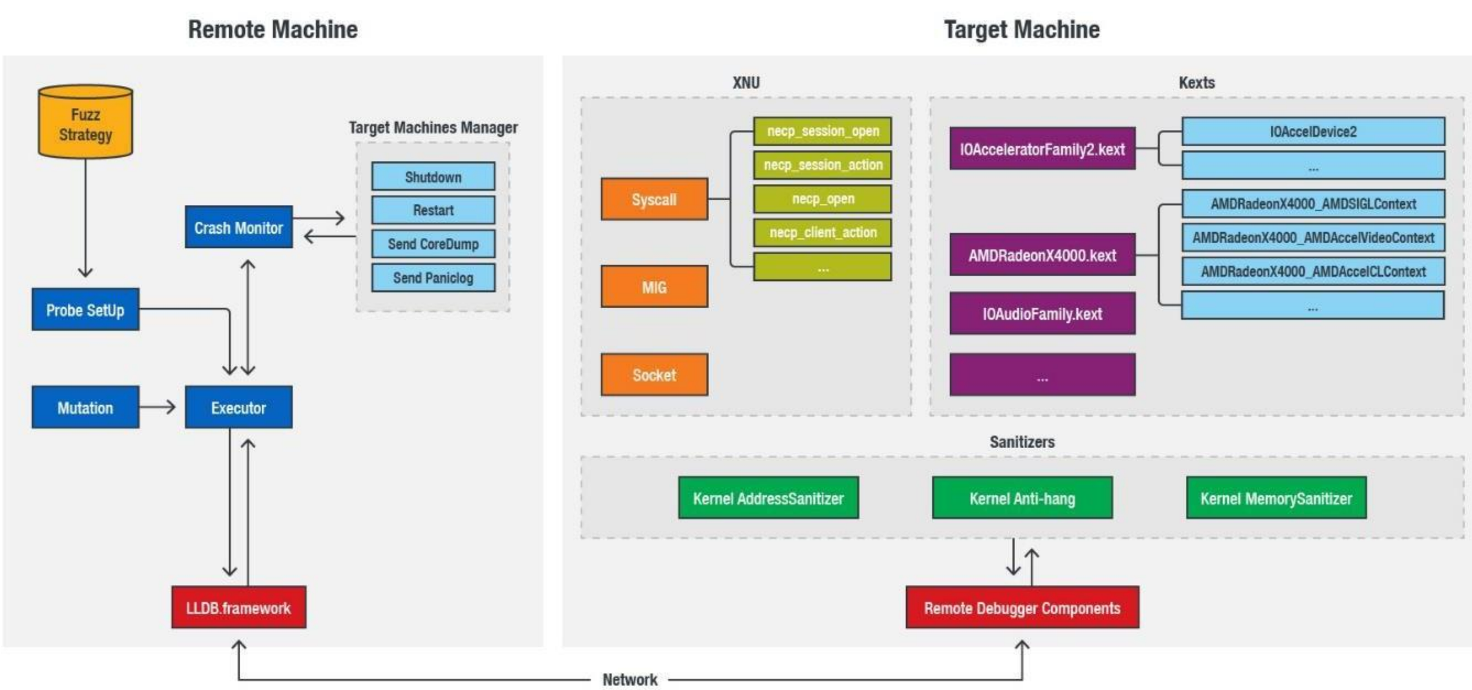



*Figure 7. The LLDBFuzzer architecture*

The following details each module:

- **Probe Setup -** It will query the fuzz strategy, which contains all the attack surfaces we revised from XNU and KEXTs, and parse them for an executor to deploy probes on the target machine.
- **Mutation -** Executor will break at probe point, then bit flip their input buffer. However, not all the inputs need mutation because the inputs are not always buffers; the executor will use the debug function (such as "showobject") to check them.
- **Crash Monitor -** This module will monitor the status of target machines via the fuzzing log and return the signal. It can also use the manager toolset to restart or send core dump and panic logs to fuzzing servers for further reproduction.
- **Executor -** This is a fuzz controller for all fuzzing steps.
- **Sanitizers -** The target machine loads our custom XNU, which is compiled with a kernel address sanitizer (KSAN) and a kernel memory sanitizer (KMSAN). These two sanitizers were introduced in our BlackHat Europe 2018 presentation.
- **Remote Debugger Components**: This module is an essential part of our whole fuzzing solution. It is implemented in the Ethernet driver; however, not all drivers implement the kernel debugger functions (an example would be the Intel Mausi Network Driver). Section 2.3 will introduce how to implement a remote kernel debugger in the open source driver.
- **XNU and KEXTs**: Unusually, due to the features of an LLDB debugger, LLDBFuzzer will not only pay attention to the normal attack surface, such as "is_io_connect_method" and "unix_syscall64", but also to the deeper attack surface, such as the "IOAccelCommandStreamInfo" process functions in the AMDRadeonX4000_AMDSIGLContext service.

## 1.3. The fuzz attack surface on Macintosh

**Hacking into AMD graphic drivers**

AMD Graphic Drivers are used to accelerate and optimize 2D, 3D, and video rendering. They contain many interfaces that the user space can access, so we chose them as our research target.

Below, we will show how to uncover deeper and hidden potential attack surfaces that can allow malicious actors to hack into AMD accelerator family in Radeon Drivers.

**Determine the active accelerator in the target machine**

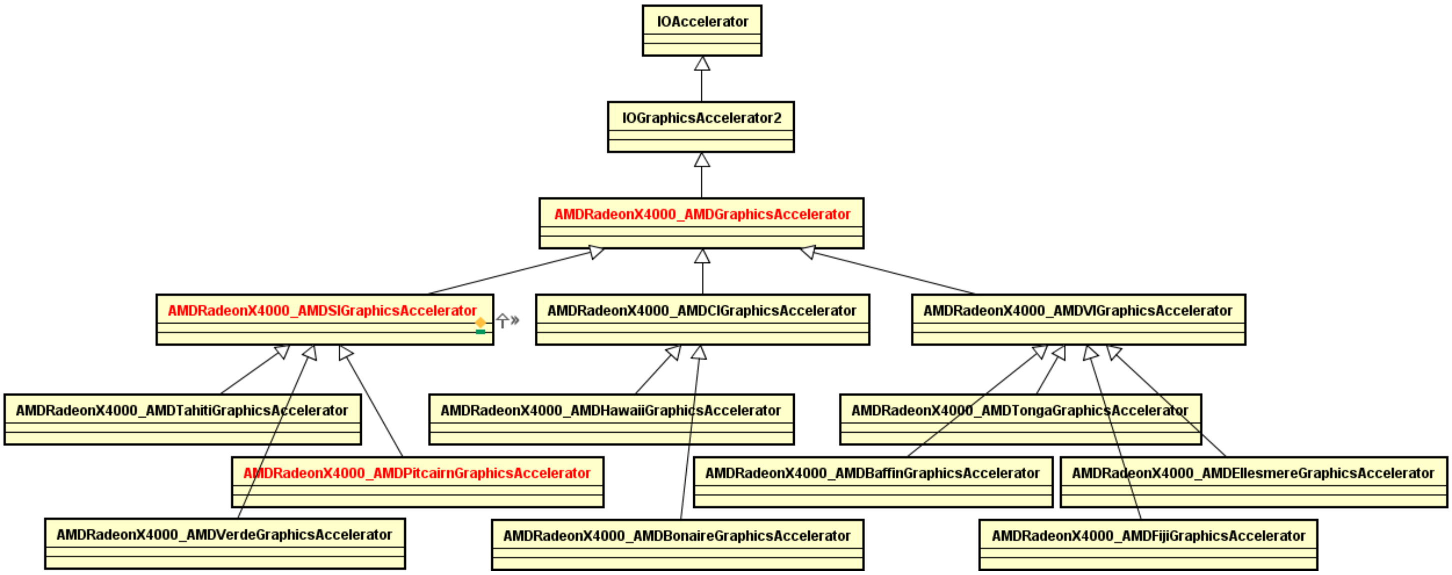



*Figure 8. Class diagram of IOAccelerator and its derived class in AMD Graphic Driver*

Figure 8 shows the whole accelerator family in an AMDRadeonX4000 driver, each of them adaptable for different GPU models. Our Mac Pro test machine features two AMD FirePro GPUs (shown in Figure 9); AMDRadeonX4000_AMDPitcairnGraphicsAccelerator is active.

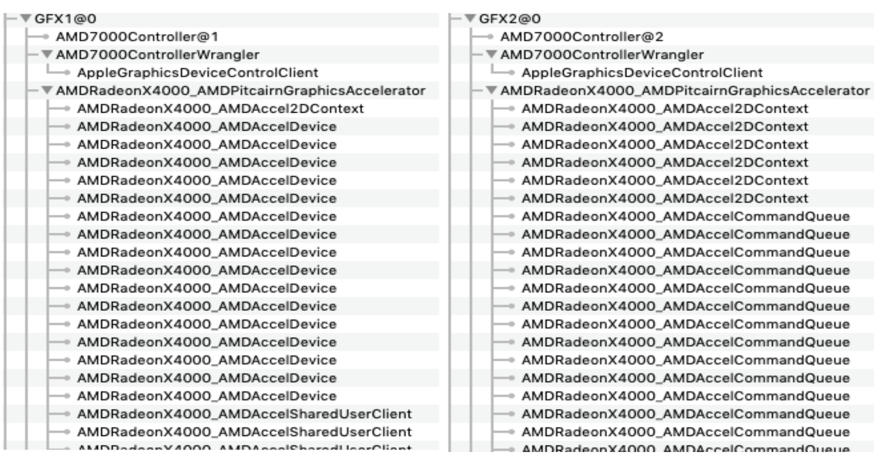



*Figure 9. The two AMD graphics accelerators in Mac Pro, featuring two AMD FirePro GPUs*

**Get the usual attack surface for AMDPitcairnGraphicsAccelerator**

```
switch ( opentype )
{
  case 0u:
    LODWORD(v13) = ((int (__fastcall *)(IOGraphicsAccelerator2 *))this->vtable->newSurface)(this);
    v14 = (IOAccelDisplayPipeUserClient2 *)v13;// AMDRadeonX4000_AMDGraphicsAccelerator::newSurface(void)
    v7 = -536870210;
    if ( v13 )
    {
      v12 = 0LL;
      v15 = IOAccelSurface2::init(v13, 0LL, v6);
      goto LABEL_25;
    }
    return (unsigned int)v7;
  default:
    LODWORD(v16) = ((int (__fastcall *)(IOGraphicsAccelerator2 *, _QWORD))this->vtable->__ZN22IOGraphicsAccelerator210newContextEj)(
                     this,
                     opentype);
    v17 = (IOUserClient *)v16;
    v7 = -536870206;
    if ( v16 )
    {
      if ( (unsigned __int8)IOAccelContext2::init(v16, 0LL, v6) )
        goto LABEL_32;
      (*(void (__fastcall **)(IOUserClient *, _QWORD))(*(_QWORD *)v17 + 40LL))(v17, 0LL);
      v7 = -536870210;
    }
    return (unsigned int)v7;
  case 2u:
    LODWORD(v18) = ((int (__fastcall *)(IOGraphicsAccelerator2 *))this->vtable->new2DContext)(this);
    v14 = (IOAccelDisplayPipeUserClient2 *)v18;// AMDRadeonX4000_AMDGraphicsAccelerator::new2DContext(void)
    v7 = -536870210;
    if ( v18 )
    {
      v12 = 0LL;
      v15 = IOAccelContext2::init(v18, 0LL, v6);
      goto LABEL_25;
    }
    return (unsigned int)v7;
```

*Figure 10. The pseudo code of the IOGraphicsAccelerator2::newUserClient function*

The *newUserClient* function is used to create a connection for an IOServce with a type that the caller specifies. Based on the pseudo code shown in Figure 10, IOAcceleratorFamily2.kext has many available associated services that can be accessed from user space using the *IOServiceOpen* function. Table 3 lists the actual derived services and access types. These derived services are also available if the device uses the Intel series GPU AppleIntelHD5000Graphics.kext or other kernel extensions.

| Open Type | Parent Service | Derived Service in AMDRadeonX4000.kext |
|---|---|---|
| 0 | IOAccelSurface2 | AMDRadeonX4000_AMDAccelSurface |
| 1 | IOAccelContext2→ IOAccelGLContext2 | AMDRadeonX4000_AMDSIGLContext |
| 2 | IOAccelContext2→ IOAccel2DContext2 | AMDRadeonX4000_AMDAccel2DContext |
| 3 | IOAccelContext2→ IOAccelVideoContext2 | AMDRadeonX4000_AMDAccelVideoContext→ AMDRadeonX4000_AMDSIVideoContext |
| 4 | IOAccelDisplayPipe2 | AMDRadeonX4000_AMDAccelDisplayPipe |
| 5 | IOAccelDevice2 | AMDRadeonX4000_AMDAccelDevice |
| 6 | IOAccelSharedUserClient2 | AMDRadeonX4000_AMDAccelSharedUserClient |
| 7 | IOAccelMemoryInfoUserClient | |
| 8 | IOAccelContext2→ IOAccelCLContext2 | AMDRadeonX4000_AMDAccelCLContext→ AMDRadeonX4000_AMDSICLContext |
| 9 | IOAccelCommandQueue | AMDRadeonX4000_AMDAccelCommandQueue |

*Table 3. Graphic Services and its Open Type from User Space (A→B means B extends A)*

Besides getting these AMD services, getting the external methods dispatch is also essential so that we can find the first level of attack surfaces. *IOUserClient::externalMethod* and *IOUserClient::getTargetAndMethodForIndex* are the common override functions to reverse to get the dispatch table. Some of the services may fully rewrite these two functions, which makes reverse engineering a little difficult and not friendly for automation, but it can still be effective after some effort. Table 3 shows the main IOServices and their extended relationships. Table 6 shows the external method and its index of *AMDRadeonX4000_AMDSIGLContext*. Since IOAccelGLContext2 extends IOAccelContext2, the other GL context operation functions are implemented in IOAccelContext2 class as shown in Table 4.

| index | flags | count 1 | count2 | Methods Name |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | IOAccelContext2::finish(void) |
| 1 | 4 | 0 | 0xffffffff | IOAccelContext2::set_client_info(IOAccelClientInfo *,ulong long) |
| 2 | 3 | 0x88 | 0xffffffff | IOAccelContext2::submit_data_buffers(IOAccelContextSubmitDataBuffersIn *,IOAccelContextSubmitDataBuffersOut *,ulong long,ulong long *) |
| 3 | 3 | 8 | 0xffffffff | IOAccelContext2::get_data_buffer(IOAccelContextGetDataBufferIn *,IOAccelContextGetDataBufferOut *,ulong long,ulong long *) |
| 4 | 0 | 0 | 0 | IOAccelContext2::reclaim_resources(void) |
| 5 | 0 | 1 | 0 | IOAccelContext2::finish_fence_event(uint) |
| 6 | 0 | 0 | 0 | |
| 7 | 0 | 1 | 0 | IOAccelContext2::set_background_rendering(uint) |

*Table 4. The external method of IOAccelContext2*

| Sele-ctor | Scalar InputCount | Structure InputSize | Scalar Output Count | Structure OutputSize | Methods Name |
|---|---|---|---|---|---|
| 256 | 0 | 0x30 | 0 | 0 | IOAccelGLContext2::s_set_surface(IOAccelGLContext2*,void *,IOExternalMethodArguments *) |
| 257 | 0 | 0x30 | 0 | 0x28 | IOAccelGLContext2::s_set_surface_get_config_status(IOAccelGLContext2*,void *,IOExternalMethodArguments *) |
| 258 | 4 | 0 | 0 | 0 | IOAccelGLContext2::s_set_swap_rect(IOAccelGLContext2*,void *,IOExternalMethodArguments *) |

| | | | | | |
|---|---|---|---|---|---|
| 259 | 2 | 0 | 0 | 0 | IOAccelGLContext2::s_set_swap_interval(IOAccelGLContext2*,void *,IOExternalMethodArguments *) |
| 260 | 1 | 0 | 0 | 0 | IOAccelGLContext2::s_set_surface_volatile_state(IOAccelGLContext2 *,void *,IOExternalMethodArguments *) |
| 261 | 0 | 0x20 | 0 | 0 | IOAccelGLContext2::s_read_buffer(IOAccelGLContext2*,void *,IOExternalMethodArguments *) |

*Table 5. The external method dispatch of IOAccelGLContext2*

| index | flags | count1 | count2 | Methods Name |
|---|---|---|---|---|
| 512 | 4 | 0 | 0xffffffff | AMDRadeonX4000_AMDSIGLContext::readPixelsFBO(sATIGLContextReadPixelsFBOData *,ulong long) |
| 513 | 4 | 0 | 0x18 | AMDRadeonX4000_AMDSIGLContext::SurfaceCopy(uint *,ulong long) |

*Table 6. the external method of AMDRadeonX4000_AMDSIGLContext*

**More Hidden Attack Surfaces**

Though the usual attack surfaces can be tested and fuzzed directly from user space, there are still multiple functions within the drivers that cannot be touched. Mainly, these functions contains three kinds of interfaces:

1) Interfaces that are protected by filter driver, which researcher Yu Wang introduced in DEFCON 26
2) Interfaces that are controlled by the shared memory
3) Interfaces that cannot be indirectly touched by user space processes, but can be accessed by Safari and special processes

We will illustrate the second and the third type of hidden and deep attack surfaces.

*A. Interfaces which are controlled by the shared memory*

AMDRadeonX4000_AMDSIGLContext provides a set of side band buffer process functions called by the "processSidebandToken" method and controlled through the IOAccelCommandStreamInfo object.

```
v4 = (char *)&this->commandStreamInfo;        // cmdinfo start address, can be control by share memory offset 16
LODWORD(this->member199) = 0;
this->commandStreamInfo = 0LL;
this->member194 = 0LL;
LODWORD(this->member195) = 0;
v5 = this->shareMem_start_vm_address_187 + 16;
this->member196 = v5;
LODWORD(this->member200) = 0;
BYTE4(this->member200) = a3;                  // =1
while ( 1 )
{
  v6 = this->shareMem_end_vm_address_188;
  if ( v5 + 8 > v6 )
  {
    v14 = _os_log_default_0;
    _os_log_internal(
      &dword_0,
      _os_log_default_0,
      17LL,
      IOAccelContext2::processSidebandBuffer(IOAccelCommandDescriptor *,bool)::_os_log_fmt,
      "virtual bool IOAccelContext2::processSidebandBuffer(IOAccelCommandDescriptor *, bool)");
    v15 = LOWORD(this->commandStreamInfo_offset32);
    v16 = WORD1(this->commandStreamInfo_offset32);
    _os_log_internal(
      &dword_0,
      v14,
      17LL,
      IOAccelContext2::setContextError(unsigned int)::_os_log_fmt,
      "void IOAccelContext2::setContextError(uint32_t)");
    goto LABEL_18;
  }
  LOWORD(this->commandStreamInfo_offset32) = *(_WORD *)v5;
  v7 = *(_WORD *)(v5 + 2);
  WORD1(this->commandStreamInfo_offset32) = v7;
  v8 = *(_DWORD *)(v5 + 4);
  HIDWORD(this->commandStreamInfo_offset32) = v8;// set the commandstreaminfo
  this->member198 = v5 + 8;
```

*Figure 11. The accelerator command stream info is controlled by shared memory*

Figure 11 shows that *v5* points to the shared memory start address and offset 16 bit, and *commandStreamInfo_offset32* points to the commandStreamInfo structure and offset 32 bit. Then, the following code assigns two words and one DWORD data of *v5* to *commandStreamInfo_offset32*, and passes them to the *AMDRadeonX4000_AMDSIGLContext::processSidebandToken* function. This function gets the first word of *commandStreamInfo_offset32* and subtracts 120 as the index of the *ati_token_process_methods* dispatch array, as shown in Figure 12. After that, the methods hide behind the *IOAccelContext2::submit_data_buffers* external method, which has a selector of "2" as shown in Table 4, and can be accessed.

```
void __fastcall AMDRadeonX4000_AMDSIGLContext::processSidebandToken(IORegistryEntry *this, uintptr_t cmdinfo)
{
  uintptr_t v2; // r12@1
  unsigned __int16 v3; // ax@1
  int v4; // ebx@3
  int v5; // eax@3
  signed __int64 v6; // rax@6
  unsigned __int8 v7; // cf@6
  _DWORD *v8; // rbx@7
  uintptr_t v9; // r15@8
  uintptr_t v10; // rax@8
  uintptr_t v11; // r15@10
  uintptr_t v12; // rax@10
  __int64 v13; // rax@12
  unsigned __int64 v14; // rax@13

  v2 = cmdinfo;
  v3 = *(_WORD *)(cmdinfo + 32);                  // can be controlled by share memeory which offset is 16

  ... ...

  v4 = HIBYTE(v3) - 128;
  v5 = *(_DWORD *)((char *)tokenArgSizeVaries + (((unsigned int)v4 >> 3) & 0x1FFC));
  if ( _bittest(&v5, v4)
    || (unsigned __int8)IOAccelContext2::validateTokenSize(
                          this,
                          cmdinfo,
                          (unsigned __int16)(((unsigned int)tokenArgSizes[(unsigned __int16)v4] + 11) >> 2)) )
  {
    if ( *(_DWORD *)(*(_QWORD *)(cmdinfo + 24) + 4LL) < *((_DWORD *)this + 1150) )
    {
      v6 = 2LL * (unsigned __int16)v4;
      v7 = __CFADD__(AMDRadeonX4000_AMDSIGLContext::ati_token_process_methods[v6 + 1], this);
      JUMPOUT(__CS__, AMDRadeonX4000_AMDSIGLContext::ati_token_process_methods[v6]);
    }
    IOAccelContext2::setContextError(this, 0xFFFFFFFC);
  }

           public __ZN29AMDRadeonX4000_AMDSIGLContext25ati_token_process_methodsE
_QWORD AMDRadeonX4000_AMDSIGLContext::ati_token_process_methods[]
__ZN29AMDRadeonX4000_AMDSIGLContext25ati_token_process_methodsE dq offset __ZN29AMDRadeonX4000_AMDSIGLContext23proces
                                        ; DATA XREF: AMDRadeonX4000_AMDSIGLContext::processSidebandToken(IOAccelComma
                                        ; AMDRadeonX4000_AMDSIGLContext::process_StateShadowInfo(IOAccelCommandStrea
           align 10h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext24process_InvalidateObjectER24IOAccelCommandStreamInfo ;
           align 20h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext18handle_BindObjectsER24IOAccelCommandStreamInfo ; AMDRa
           align 10h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext20handle_UnbindObjectsER24IOAccelCommandStreamInfo ; AMD
           align 20h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext23handle_UnusedDataBufferER24IOAccelCommandStreamInfo ;
           align 10h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext22process_UnhandledTokenER24IOAccelCommandStreamInfo ; A
           align 20h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext22process_UnhandledTokenER24IOAccelCommandStreamInfo ; A
           align 10h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext22process_UnhandledTokenER24IOAccelCommandStreamInfo ; A
           align 20h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext25process_PatchStreamTexBufER24IOAccelCommandStreamInfo
           align 10h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext22process_UnhandledTokenER24IOAccelCommandStreamInfo ; A
           align 20h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext18process_DrawBufferER24IOAccelCommandStreamInfo ; AMDRa
           align 10h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext22process_UnhandledTokenER24IOAccelCommandStreamInfo ; A
           align 20h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext22process_StretchTex2TexER24IOAccelCommandStreamInfo ; A
           align 10h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext22process_CopyColorScaleER24IOAccelCommandStreamInfo ; A
           align 20h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext17process_AAResolveER24IOAccelCommandStreamInfo ; AMDRad
           align 10h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext23process_StretchSurf2TexER24IOAccelCommandStreamInfo ;
           align 20h
           dq offset __ZN29AMDRadeonX4000_AMDSIGLContext25process_ClearDepthStencilER24IOAccelCommandStreamInfo
```

*Figure 12. The side band buffer process functions hide behind the external method, which has a selector of 2*

The shared memory can be operated from user space using the code shown in Figure 13. Figure 14 lists the main service class and their relationship to help analyze the IOAcceleratorFamily2.kext. The figure in the appendix shows the important field variables and their offsets in each service class. We will also clarify each service class's main function and what role they play in the IOAcceleratorFamily extension below.

```c
// connect IOAccelSharedUserClient2 opentype 6
shm_service = IOServiceGetMatchingService(kIOMasterPortDefault, IOServiceMatching("AMDRadeonX4000_AMDGraphicsAccelerator"));//IOAccelSharedUserClient2
printf("Step2: got service %x\n", shm_service);
ret = IOServiceOpen(shm_service, mach_task_self(), 6, &shm_conn);
if (ret != KERN_SUCCESS) {
    printf("Step2 Error: unable to get user client connection, error code %x\n", ret);
    //return 0;
}
printf("Step2: got connection %x\n", shm_conn);
IOConnectAddClient(glc_conn, shm_conn);


// map memory Connect
ioc_service = IOServiceGetMatchingService(kIOMasterPortDefault, IOServiceMatching("AMDRadeonX4000_AMDGraphicsAccelerator"));//AMDRadeonX4000_AMDSIGLContext
    AMDRadeonX4000_AMDGraphicsAccelerator
printf("Step3: got service %x\n", ioc_service);
ret = IOServiceOpen(ioc_service, mach_task_self(), 2, &ioc_conn);
if (ret != KERN_SUCCESS) {
    printf("Step3 Error: unable to get user client connection, error code %x\n", ret);
    //return 0;
}
printf("Step3: got connection %x\n", ioc_conn);


mach_vm_address_t atAddress = 0;
mach_vm_size_t ofsize = 0x1000;
ret = IOConnectMapMemory64(ioc_conn, 0, mach_task_self(), &atAddress, &ofsize, kIOMapAnywhere);
if (ret != KERN_SUCCESS) {
    printf("Step4 Error: IOConnectMapMemory64 error, error code: %x\n", ret);
}
printf("Step4: got share memory address %#llx\n", atAddress);
printf("Step4: got size %#llx\n", ofsize);
*((uint16_t*)(atAddress + 16)) = 0x8c00;
*((uint16_t*)(atAddress + 16) + 1) = 0x001a;
*((uint32_t*)(atAddress + 16) + 1) = 0x00000364;
```

*Figure 13. The demo showing how to operate the accelerator context share memory*

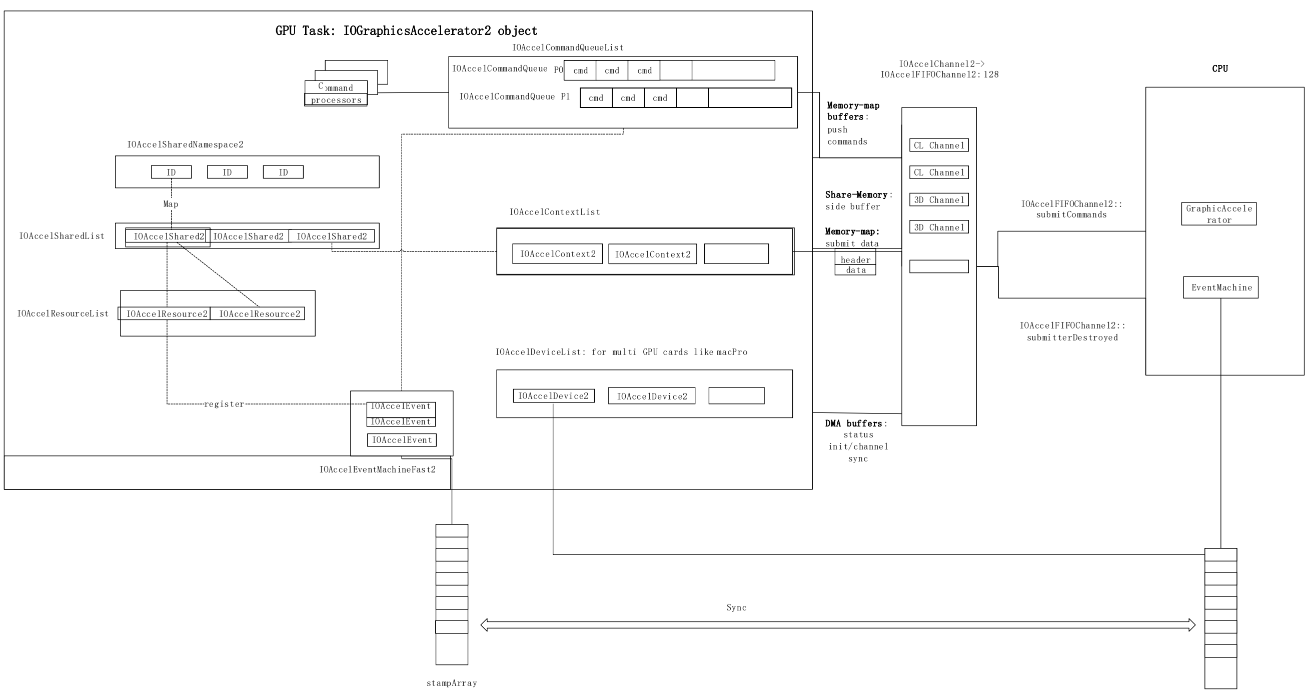



*Figure 14. The details of the AMD Accelerator driver*

B. *Interfaces which cannot be indirectly touched by user space processes, but can be accessed by the Safari (and others) special processes*

The IOFramebuffer service defines APIs used to publish a linear framebuffer device. AMD device writers extend this class and provide an AMDFramebuffer driver. It creates three types of connections: kIOFBServerConnectType, kIOFBSharedConnnectType, and kIOFBDiagnoseConnectType. However, the kIOFBServerConnectType connection cannot be accessed through the normal user-mode process.

But, that does not mean that there is no vulnerability there: one example of a bug would be CVE-2018-4462, which we reported to Apple (details will be introduced in vulnerability section below). The details of the external method dispatch can be referred to in *IOFramebufferUserClient::externalMethod* in the IOFramebufferUserClient.cpp file. However, the execution methods are those implemented in AMDFramebuffer.kext, as shown in Figure 15.

```
dq offset __ZN14AMDFramebuffer16getApertureRangeEi ; AMDFramebuffer::getApertureRange(int)
dq offset __ZN14AMDFramebuffer12getVRAMRangeEv ; AMDFramebuffer::getVRAMRange(void)
dq offset __ZN14AMDFramebuffer16enableControllerEv ; AMDFramebuffer::enableController(void)
dq offset __ZN14AMDFramebuffer15getPixelFormatsEv ; AMDFramebuffer::getPixelFormats(void)
dq offset __ZN14AMDFramebuffer19getDisplayModeCountEv ; AMDFramebuffer::getDisplayModeCount(void)
dq offset __ZN14AMDFramebuffer15getDisplayModesEPi ; AMDFramebuffer::getDisplayModes(int *)
dq offset __ZN14AMDFramebuffer28getInformationForDisplayModeEiP24IODisplayModeInformation ; AMDFramebuffer::getInf
dq offset __ZN14AMDFramebuffer29getPixelFormatsForDisplayModeEii ; AMDFramebuffer::getPixelFormatsForDisplayMode(i
dq offset __ZN14AMDFramebuffer19getPixelInformationEiiiP18IOPixelInformation ; AMDFramebuffer::getPixelInformatio
dq offset __ZN14AMDFramebuffer21getCurrentDisplayModeEPiS0_ ; AMDFramebuffer::getCurrentDisplayMode(int *,int *)
dq offset __ZN14AMDFramebuffer14setDisplayModeEii ; AMDFramebuffer::setDisplayMode(int,int)
dq offset __ZN13IOFramebuffer17setApertureEnableEij ; IOFramebuffer::setApertureEnable(int,uint)
dq offset __ZN14AMDFramebuffer21setStartupDisplayModeEii ; AMDFramebuffer::setStartupDisplayMode(int,int)
dq offset __ZN14AMDFramebuffer21getStartupDisplayModeEPiS0_ ; AMDFramebuffer::getStartupDisplayMode(int *,int *)
dq offset __ZN14AMDFramebuffer18setCLUTWithEntriesEP12IOColorEntryjjj ; AMDFramebuffer::setCLUTWithEntries(IOColor
dq offset __ZN14AMDFramebuffer13setGammaTableEjjjPv ; AMDFramebuffer::setGammaTable(uint,uint,uint,void *)
dq offset __ZN14AMDFramebuffer12setAttributeEjm ; AMDFramebuffer::setAttribute(uint,ulong)
dq offset __ZN14AMDFramebuffer12getAttributeEjPm ; AMDFramebuffer::getAttribute(uint,ulong *)
dq offset __ZN14AMDFramebuffer27getTimingInfoForDisplayModeEiP19IOTimingInformation ; AMDFramebuffer::getTimingInf
dq offset __ZN14AMDFramebuffer22validateDetailedTimingEPvy ; AMDFramebuffer::validateDetailedTiming(void *,ulong l
dq offset __ZN14AMDFramebuffer18setDetailedTimingsEP7OSArray ; AMDFramebuffer::setDetailedTimings(OSArray *)
dq offset __ZN13IOFramebuffer18getConnectionCountEv ; IOFramebuffer::getConnectionCount(void)
dq offset __ZN14AMDFramebuffer25setAttributeForConnectionEijm ; AMDFramebuffer::setAttributeForConnection(int,uint
dq offset __ZN14AMDFramebuffer25getAttributeForConnectionEijPm ; AMDFramebuffer::getAttributeForConnection(int,uir
```

*Figure 15. The execution functions implement in AMDFramebuffer.kext*

**Hacking into special syscalls**

*Unix_syscall64* is the dispatch function for syscall in XNU and the corresponding function in user space is syscall. This is one of most important attack interfaces towards kernel privilege escalation crossing platforms (including OSX and iOS).

```
* thread #1, stop reason = breakpoint 3.1
  * frame #0: 0xffffff800e038d52 kernel.development`sysctl(p=0xffffff800e6e6e50, uap=0xffffff8015bb7040,
retval=<unavailable>) at kern_newsysctl.c:1705 [opt]
    frame #1: 0xffffff800e15b845 kernel.development`unix_syscall64(state=0xffffff80164fed80) at systemcal
ls.c:389 [opt]
    frame #2: 0xffffff800da2a456 kernel.development`hndl_unix_scall64 + 22
(lldb)
```

*Figure 16. unix_syscall64 in call stack (sysctl for example)*

Above is the typical system call backtrace, where we used sysctl as an example. From the brief implementation of unix_syscall64 listed below, we can get import system call info from the input argument “state” that includes registers of execution context, system call number, arguments zone in kernel mode, and so forth.

```
__attribute__((noreturn)) void unix_syscall64(x86_saved_state_t *state)
{

        p = current_proc();
        regs = saved_state64(state);
        //Get system call number from saved registers
        uSyscallNumber = regs->rax & SYSCALL_NUMBER_MASK;
        //uSyscallNumber = regs->rdi;//indirect system call
        callp = &sysent[uSyscallNumber];

        //copy in user data to kernel address(vt or uthread->uu_arg)
        vt = (void *)uthread->uu_arg;
        copyin_count = (callp->sy_narg - args_in_regs) * sizeof(syscall_arg_t);
        //int     copyin(const user_addr_t uaddr, void *kaddr, size_t len);
        error = copyin(
                (user_addr_t)(regs->isf.rsp + sizeof(user_addr_t)),
                (char *)&uthread->uu_arg[args_in_regs]/*kernel address*/,
                copyin_count);

        //Call system call
        error = (*(callp->sy_call))((void *) p/*current process*/,
                vt/*kernel address for arguments*/,
                &(uthread->uu_rval[0]));

}
```

We analyzed system call trigger statistics, taken for about 10 minutes in a typical runtime environment (which would happen for example playing 3D online games, website visits via Safari, running VLC media etc.) on the latest Mac OSX 10.14.4. The first column is the total hit number, the second column is the system call number, and the last column is the system call prototype.

We have neglected less important system calls for passive fuzzing based on several principles. The basic idea is that the more data structure or buffers are accepted as user input, the more attack interfaces the system call will open. For example, for effective fuzzing, we ignore system calls with no input argument or all input arguments that are only integer compatible and so forth.

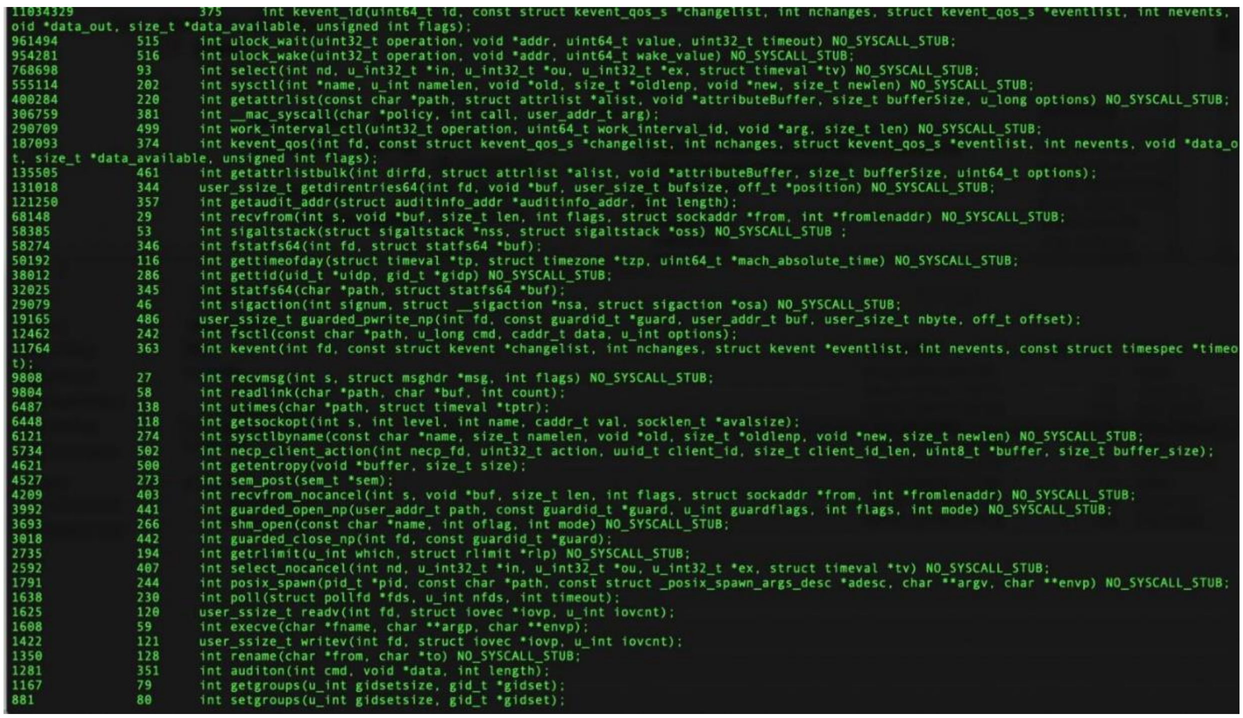



*Figure 17. Typical system call hit statistics*

To provide better references for fuzzing, we have classified the system call hit statistics into different categories according the system call hit number, as seen below.


*Figure 18. System call hit more than 100k*

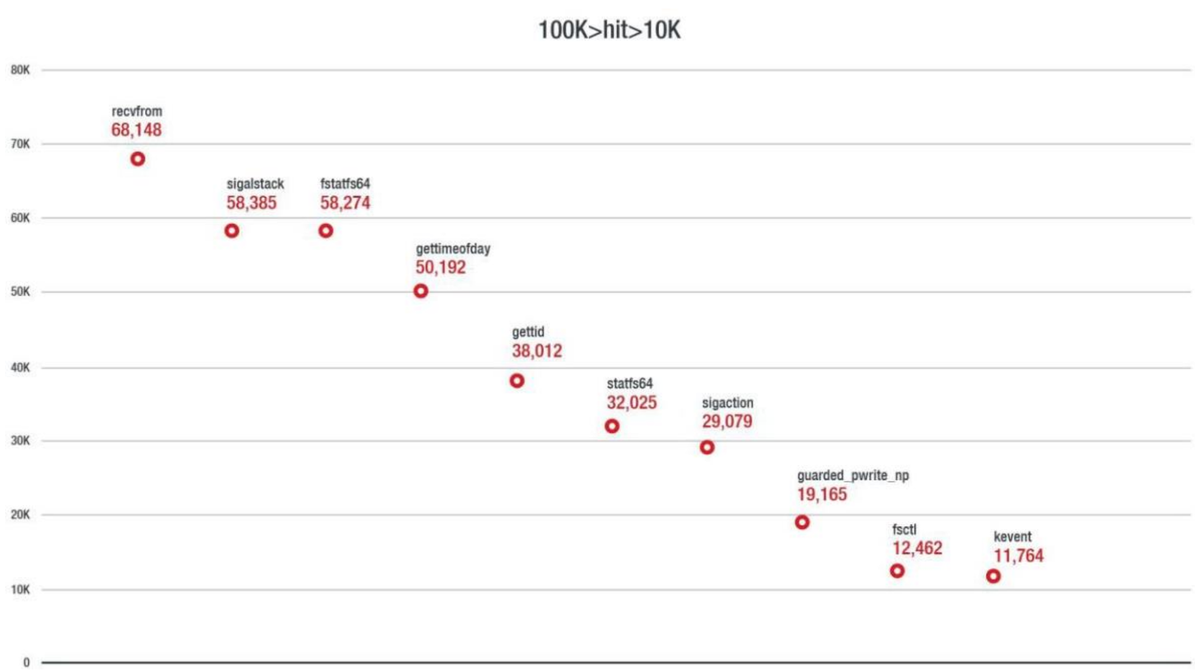



*Figure 19. System call hit between 100k and 10k*

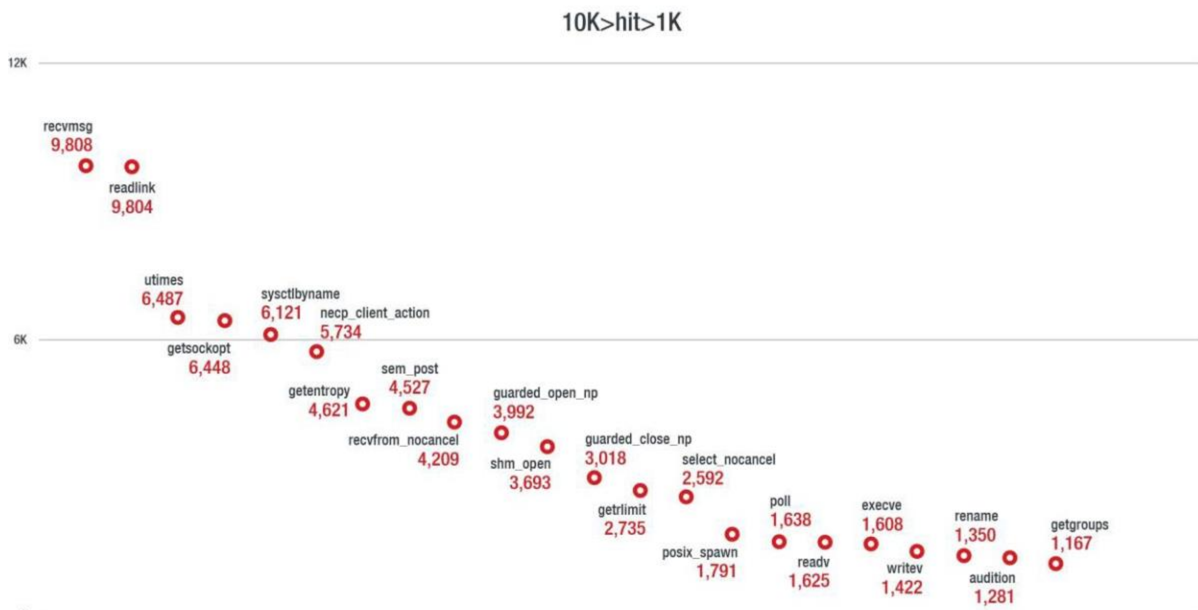



*Figure 20. System call hit between 10k and 1k*

## 1.4. The prototype of LLDBFuzzer

This section details how to implement LLDBFuzzer, including how to setup a fuzz probe and how to mutate the buffer data and the main fuzz logic.

**Probe setup**

Our fuzzing interfaces contain the depth functions, so we should first get the MacOS kernel slide in order to parse the offset of functions or variables. The probe can be one of two different kinds, function address and function names.

```python
@lldb_command("fuzzTargetSetup")
def fuzzTargetSetup(cmd_args=None):
    global target
    global bp_unix_sc_memcpy
    target = lldb.debugger.GetSelectedTarget()

    # get the MacOS kernel base
    kernel_base = 0
    kernel_load_addr = 0
    for m in target.module_iter():
        header_addr = m.GetObjectFileHeaderAddress()
        load_addr = header_addr.GetLoadAddress(target)
        file_addr = header_addr.GetFileAddress()
        print "load_addr = %s, file_addr= %s" % (hex(load_addr), hex(file_addr))
        if "kernel.development" in m.__str__():
            kernel_base = int(load_addr) - int(file_addr)
            kernel_load_addr = int(load_addr)
            break

    # set probe at unix_syscall64 -> memcpy address
    unix_syscall64_memcpy_offset = 0x75852f
    bp_unix_sc_memcpy = target.BreakpointCreateByAddress(kernel_load_addr + unix_syscall64_memcpy_offset)
    bp_unix_sc_memcpy.SetIgnoreCount(5)
    logger(str(bp_unix_sc_memcpy.GetHitCount()), 1)

    # set probe at is_io_connect_method
    bp_iscm = target.BreakpointCreateByName("is_io_connect_method")
    bp_iscm_loc = bp_iscm.GetLocationAtIndex(0)
    bp_iscm_loc.SetCondition('val == 3')

    # AMD
    bp_amd_type_2_selector_2 = target.BreakpointCreateByName("IOAccelContext2::submit_data_buffers")
```

*Figure 21. The code snippet for setting up the fuzz probe*

**Fuzz executor**

After setting up the fuzz probe, the main fuzz logic is:

1) Intercept the fuzz probe and capture the input data buffer
2) Read the input data buffer, mutate it and write them to kernel memory, as shown in Figure 22
3) Continue the interface, check the return value and monitor the fuzzing status
4) In a crash, send the core dump and panic log to the fuzz server and restart the target machine, as shown in Figure 23

```python
scalar_input_content = lldb_SBProcess.ReadMemory(ArgumentStringToInt(scalar_input_addr), scalar_input_len, lldb_err).encode("hex")
if lldb_err.Success():
    scalar_input_bytes = split_bytes(scalar_input_content)

    #scalar_input_data = scalar_input_content

    #logger("before fuzz scalar_input_content = %s" % scalar_input_content, 0)
    logger("before fuzz scalar_input_bytes = %s" % scalar_input_bytes, 0)
    #logger("before fuzz scalar_input_data = %s, len = %d" % (scalar_input_data, len(scalar_input_data)), 0)
    flip_n_byte(scalar_input_bytes, scalar_input_len, FLIP_N_RAND_LIMIT, FLIP_N_RAND_MIN, FLIP_N_RAND_MAX, FLIP_MIN_BYTES,
                FILP_MAX_BYTES)
    logger("after fuzz scalar_input_data = %s" % scalar_input_bytes, 0)

    result = lldb_SBProcess.WriteMemory(ArgumentStringToInt(scalar_input_addr), "".join(scalar_input_bytes).decode("hex"), lldb_err)
    if not lldb_err.Success():
        logger('SBProcess.WriteMemory() failed!', 1)
    else:
        logger("fuzzing object %s, selector = %s, scalar_input_bytes = %s" % (kobj_str, selector, scalar_input_bytes), 1)
```

*Figure 22. Reading data memory, mutating it and writing it back*

```
# if the target machine is paniced by lldb problem
# if yes, reboot the machine and exit
stop_reason_lldbself = True
if not stop_reason_trace:
    try:
        child.expect_exact(lldb_crash_prompt, timeout=5)
        child.sendline('kdp-reboot')
        time.sleep(2)
        logger("BCM5701Enet::cleanupTransmitRing bad instruction", 2)
        #child.expect_exact(lldb_prompt)
        exit(-1)
    except Exception, e:
        stop_reason_lldbself = False

# if the target machine is paniced by vulnerability
# if yes, send the core dump to server and detach
stop_reason_exc_bad_access = True
if not stop_reason_lldbself:
    try:
        child.expect_exact("stop reason = EXC_BAD_ACCESS", timeout=5)
        logger("Congraduations! Found EXC_BAD_ACCESS error!", 2)
        child.sendline('sendcore 10.64.20.83')
        child.expect_exact(lldb_prompt)
        #child.expect_exact(lldb_prompt)
        child.sendline('detach')
        exit(-1)
    except Exception, e:
        stop_reason_exc_bad_access = False

if not stop_reason_exc_bad_access:
    try:
        child.expect_exact("stop reason = signal SIGSTOP", timeout=5)
        logger("Congraduations! Found signal SIGSTOP error!", 2)
        child.sendline('sendcore 10.64.20.83')
        child.expect_exact(lldb_prompt)
        # child.expect_exact(lldb_prompt)
        child.sendline('detach')
        exit(-1)
    except Exception, e:
        stop_reason_exc_bad_access = False
```

*Figure 23. Monitoring the fuzz status and managing the target machine after crash*

**Mutation strategy**

We use the bit flip method to mutate the input data buffer. Then, some parameters are introduced in order to control the fuzz frequency for the fuzzing probe and fuzz ratio for data mutation, as shown in Figure 24. The parameters u_rand_limit, u_rand_min, and u_rand_max are used to control mutation ratio, while u_min_bytes and u_max_bytes control the minimum and the maximum mutation bytes.

```
def flip_byte(data, datalen):
    offset = random.randint(0, 10000) % datalen
    #print "data[%d] = %s" % (offset, data[offset])
    data[offset] = '{:0>2x}'.format(random.randint(0, 0xffffffffffffffff) % 0xff)


def flip_n_byte(data, data_len, u_rand_limit, u_rand_min, u_rand_max, u_min_bytes, u_max_bytes):
    if not (data and data_len):
        return -1

    """
    FLIP_N_RAND_LIMIT = 100
    FLIP_N_RAND_MIN   = 10
    FLIP_N_RAND_MAX   = 35
    FLIP_MIN_BYTES    = 1
    FILP_MAX_BYTES    = 50
    """

    try_fuzz_bytes  = data_len * rand_rate(u_rand_limit, u_rand_min, u_rand_max)
    u_max_legal     = get_min(data_len, u_max_bytes)
    u_min_legal     = get_min(data_len, u_min_bytes)
    real_fuzz_bytes = u_min_legal

    #if try_fuzz_bytes >= u_min_legal and try_fuzz_bytes >= u_max_legal:
    if u_min_legal <= try_fuzz_bytes <= u_max_legal:
        real_fuzz_bytes = try_fuzz_bytes

    print "real_fuzz_bytes = %d" % real_fuzz_bytes
    for i in range(int(real_fuzz_bytes)):
        flip_byte(data, data_len)


def rand_rate(u_rand_limit, u_rand_min, u_rand_max):
    if u_rand_max <= u_rand_min:
        return 0
    else:
        u_temp = random.randint(0, 10000) % (u_rand_max - u_rand_min)
        return (u_rand_min + u_temp)/float(u_rand_limit)


def get_min(num1, num2):
    if num1 <= num2:
        return num1
    return num2


def get_rand_int():
    return random.randint(0, 225) % 10
```

*Figure 24. Code snippet of the bit flip mutation strategy*

**Crash monitor**

The crash monitor module is separated independently from the target machine and is used to monitor target machine kernel panic caused by fuzzing, collect necessary crash core dump for reproduction, and reboot target machine for roll repeatedly. Below are the crash issues that the crash monitor generates automatically.

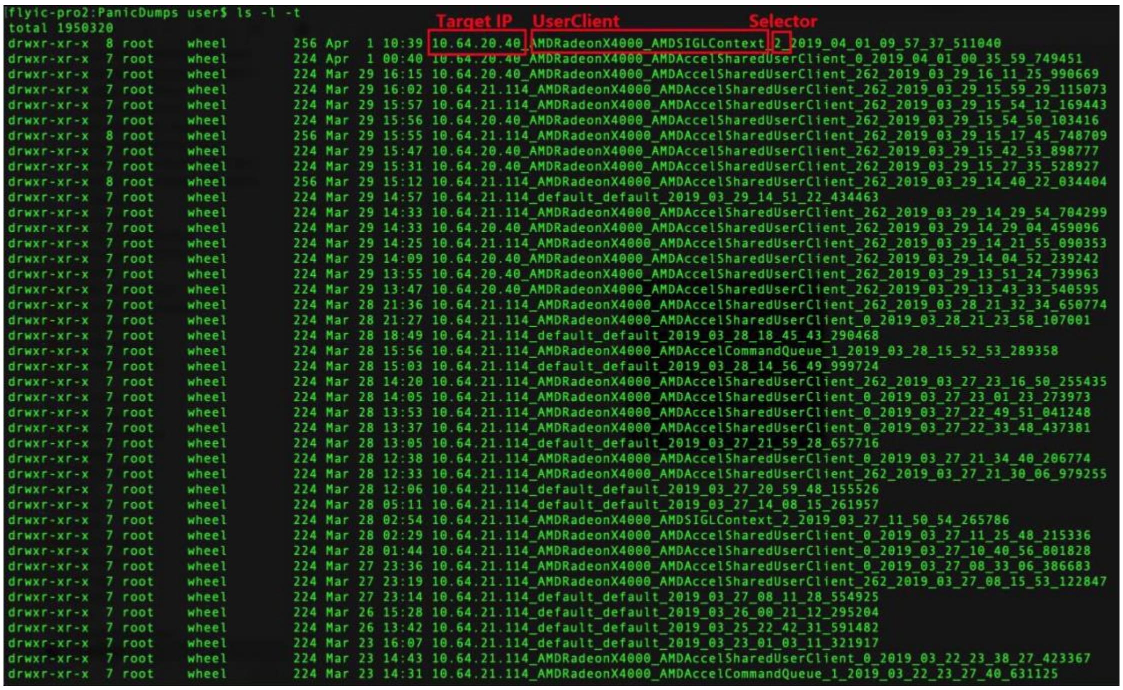



*Figure 25. Snapshot of crash issues*

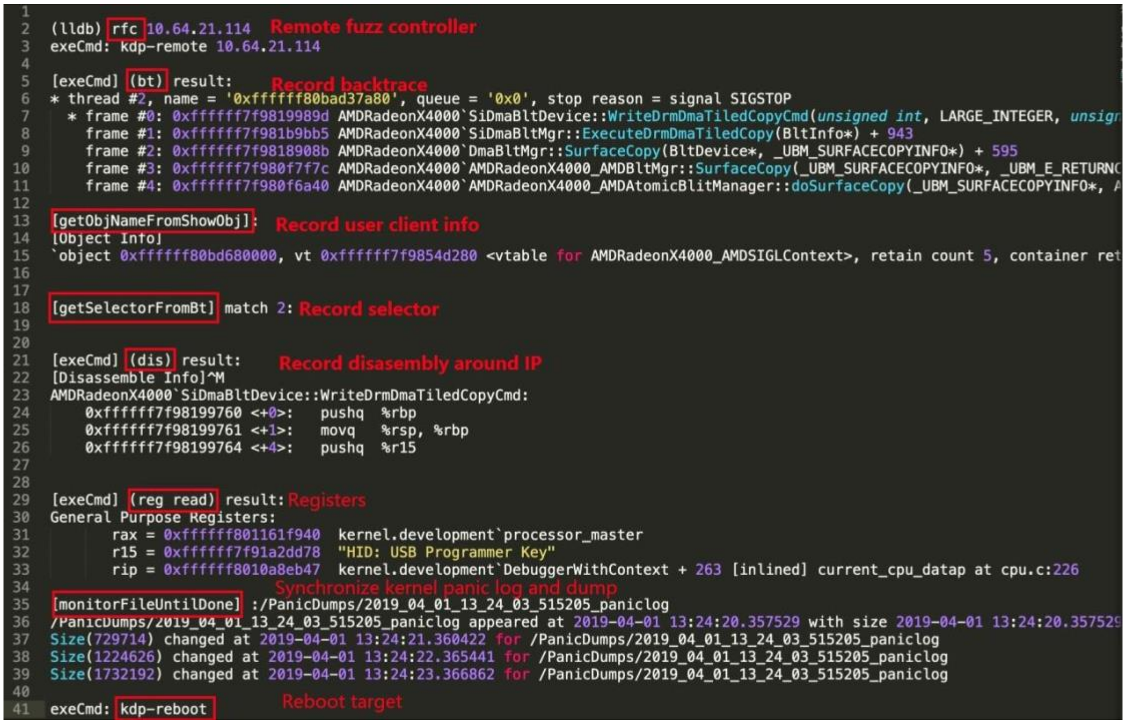



*Figure 26. LLDB monitor logic in brief*

As shown in the figure above, we have introduced the new LLDB command remote fuzz controller (RFC) in Python to monitor and remotely control the target machine. This command will query the target machine to crash in "kdp-remote" in a whole loop. Whenever an attachment to a target kernel is done, the backtrack stack, user client info, registers, and disassembly around IP (indicated in red boxes in figure above) will be collected using an internal LLDB command. Finally, it will reboot the target machine to roll repeatedly.

## 1.5. Fuzzing best practices

**Trigger more fuzzing sources**

On the first day of our test, we got an OOB vulnerability (which allows for data exfiltration) in the AMDRadeonX4000.kext, as show in the Figure 27. This was not a surprise since this is the usual attacker surface. A deeper probe revealed many other crashes. All the vulnerabilities' details will be introduced in the section below.

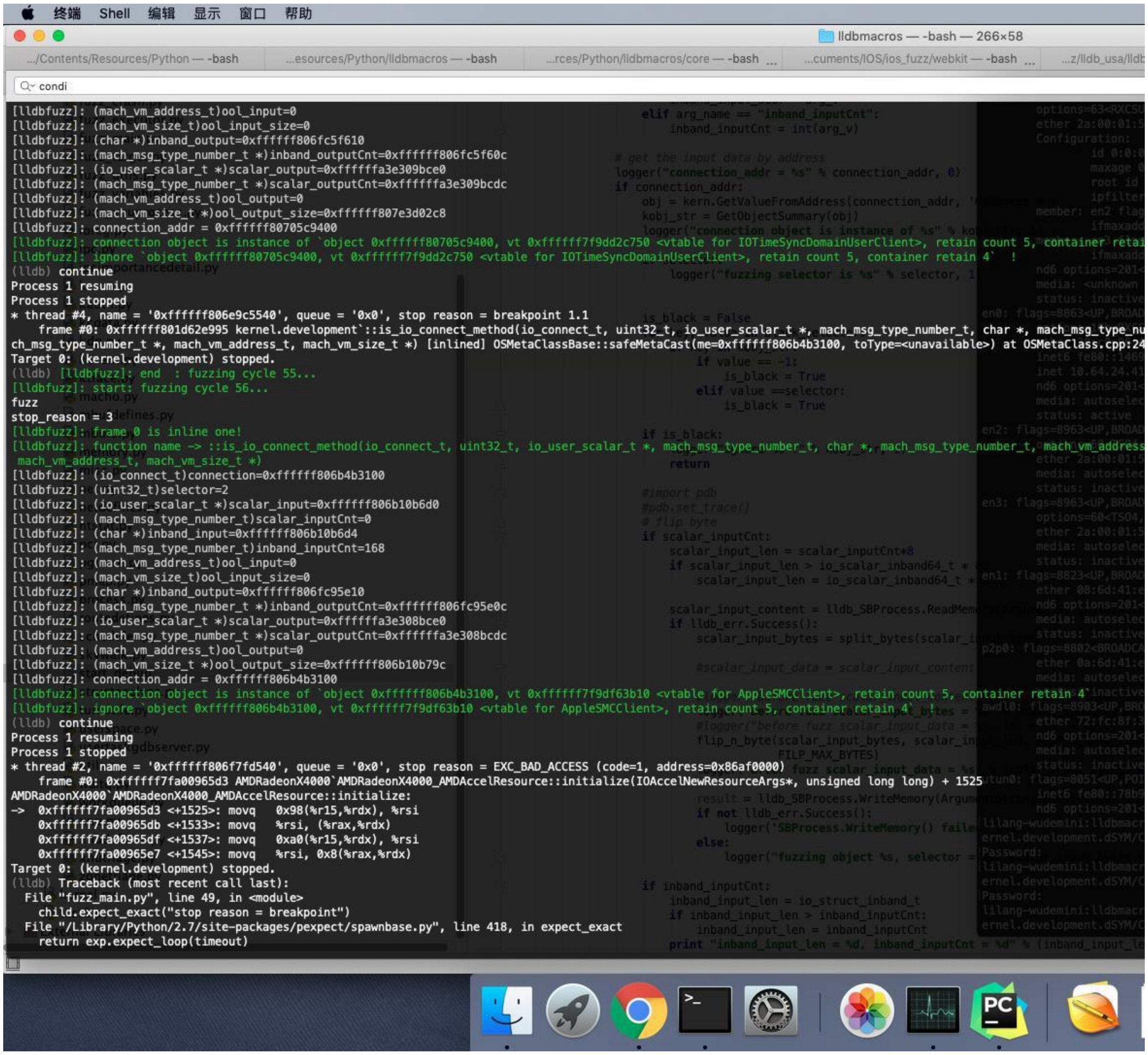



*Figure 27. The OOB vulnerability we got using LLDBFuzzer*

LLDBFuzzer also belongs to passive fuzz. In order to touch deeper attack interfaces, the following methods can be very effective:

- Run 3D games in the user space;
- Run bench marking programs in the user space, like Xbench and GFXbench;
- Run an active fuzzing tool in the user space.

These methods can make the rendering function call more frequently than usual, which helps us improve the fuzzing efficiency.

**Timely reboot in kernel for anti-hang**

```c
void doReboot()
{

    fnPEHaltRestart  afnPEHaltRestart = NULL;
    fnhalt_all_cpus afnhalt_all_cpus = NULL;
    afnPEHaltRestart = (fnPEHaltRestart) solve_kernel_symbol(&g_kernel_info,"PEHaltRestart");
    afnPEHaltRestart(kPERestartCPU);
    afnhalt_all_cpus = (fnhalt_all_cpus)solve_kernel_symbol(&g_kernel_info, "halt_all_cpus");
    afnhalt_all_cpus(TRUE);
}
void watchdogTimelyRebootThread(__unused void *arg, __unused wait_result_t wr)
{
    unsigned int nCountSeconds = (unsigned int)arg;
    struct timespec ts = { nCountSeconds, 0 };
    int error = 0;
    lck_mtx_lock(watch_dogt_timely_reboot_mutex);
    while (1) {

        fnIOSleep aIOSleep = (fnIOSleep)solve_kernel_symbol(&g_kernel_info, API_SYMBOL_IO_SLEEP);
        aIOSleep(nCountSeconds*1000);
        printf("[DEBUG] doReboot: end...\n");
        doReboot();
        //doCheck();
    }
}

kern_return_t startWatdogForTimelyReboot(unsigned int nCountSeconds)
{
    printf("[DEBUG] startWatdogForTimelyReboot (%d): begin...\n",nCountSeconds);
    kern_return_t kr =  KERN_SUCCESS;
    watch_dogt_timely_reboot_grp = lck_grp_alloc_init("startWatdogForTimelyReboot", LCK_GRP_ATTR_NULL);
    watch_dogt_timely_reboot_mutex = lck_mtx_alloc_init(watch_dogt_timely_reboot_grp, LCK_ATTR_NULL);
    kr = kernel_thread_start(watchdogTimelyRebootThread, (void *)nCountSeconds, &tWatchdogThreadHandle);
    printf("[DEBUG] startWatdogForTimelyReboot (%d): end...\n",nCountSeconds);
    return kr;
}
```

*Figure 28. Kernel thread for timely reboots*

The biggest problem for kernel fuzzing would be to have the kernel actively hang but not crash. This condition would consume time and create a false busy run for kernel fuzzing, and it could be caused by multiple conditions such as a kernel waiting for a mistake event or a watchdog mechanism.

We decided to introduce a kernel thread (kernel_thread_start API) to a timely reboot machine ("PEHaltReboot" and "halt_all_cpus" API, reversed from panic_hanlder) because the kernel thread would almost always be scheduled to execute in most "hang" conditions.

# 2. Implementing a debugger for Hackintosh

## 2.1. Why must it support kernel debugging?

As we all know, many kernel extensions can only be active beyond the real hardware, so to discover the vulnerabilities within them, the real machines are essential. Because the hardware of VMs are emulated, the kexts do not work. However, it's different for syscall fuzz because of the monolithic XNU. We can simply deploy many fuzz instances using MacOS virtual machines to improve the efficiency. For hackintosh, it's also necessary to install an open source network driver if the existing driver is not suitable for your network card.

However, many open source network drivers do not support remote kernel debugger, such as AppleIntelE1000e, RealtekRTL8111, and IntelMausiEthernet. Therefore, making them support a remote kernel debugger is a necessary precondition.

## 2.2. Kernel debugging implementation internals

Above, Figure 3 has shown the architecture of the KDP debugger implementation with an Ethernet extension. Three steps can be taken to support kernel debugging, and we can illustrate the implementation of kernel debugging by reversing the AppleBCM5701Ethernet extension:

1) Initialize a kernel debugger object and attach it
2) Implement the sendPacket() and receivePacket() virtual methods in IONetworkController
3) Implement the enable() and disable() virtual methods in IONetworkController

**Initialize the kernel debugger client**

The *attachDebuggerClient()* function in IONetworkController can allocate an IOKernelDebugger object and attach it as a client. This client is the bridge between the remote debugger and debugging world in XNU. Figure 29 shows how to attach a debugger client — it just declares a IOKernelDebugger object and calls attachDebuggerClient to attach it.

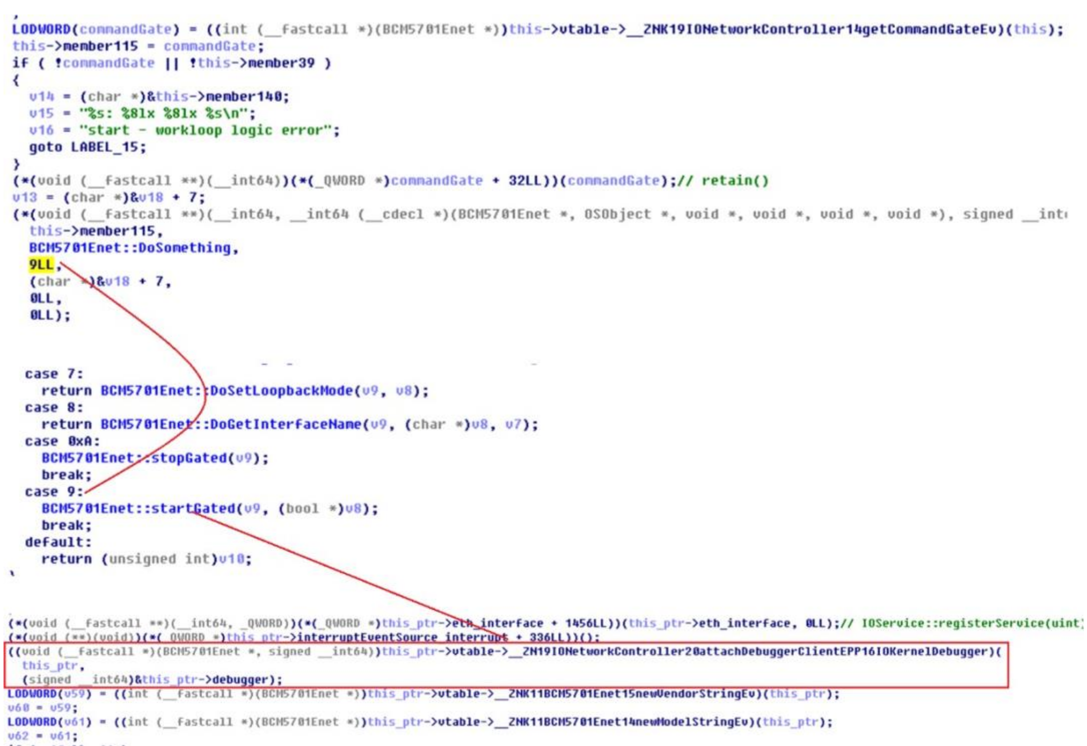



*Figure 29. Attach debugger client method in AppleBCM5701Ethernet*

### Override the packet send and receive handler functions

The sendPacket and receivePacket are the virtual methods used to declare an IONetworkController.h file. They are responsible for sending an outbound packet or polling for an incoming packet when the kernel debugger is active. An Ethernet driver that supports kernel debugging, as shown in Figure 30, must implement these two functions.

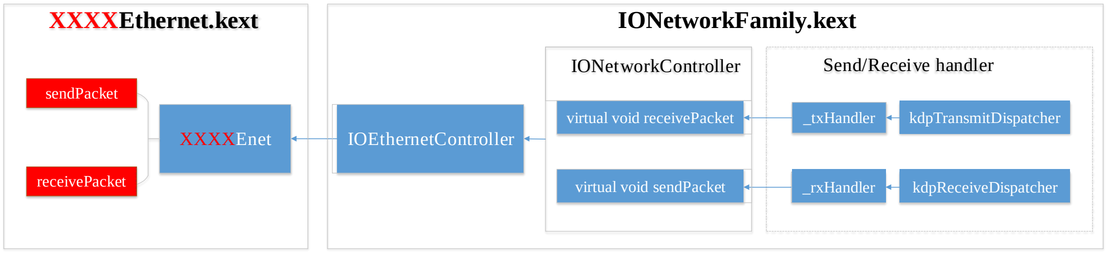



*Figure 30. The architecture for implementing kernel debugging*

### The packet send handler implementation

Figure 31 shows the one send packet cycle in AppleBCM5701Ethernet, and the following steps can be followed:

1) Allocate a packet with a data buffer
2) Move the send pkt info to the newly allocated buffer and set its length
3) Call the transmitPacket to send the packet
4) Call the transmitKick function to update the related status registers
5) Check if there is a timeout

```
LODWORD(v11) = mbuf_data(v5->allocate_packet);
memmove(v11, pkt, v4);
mbuf_setlen(v5->allocate_packet, v4);
mbuf_pkthdr_setlen(v5->allocate_packet, v4);
BCM5701Enet::transmitPacket((#13 *)v5, (IOService *)v5->allocate_packet, 3, 0LL);
BCM5701Enet::transmitKick(v5, 0);
do
{
  BCM5701Enet::serviceTxInterrupt(v5, 1, 0);
  clock_get_system_nanotime(&v15, &v16);
  result = v16;
  if ( v16 >= v14 )
  {
    v12 = v15;
  }
  else
  {
    result = (unsigned int)(result + 1000000000);
    v16 = result;
    v12 = v15-- - 1;
  }
  v8 = HIDWORD(v5->member193);
  v9 = LODWORD(v5->member194);
  if ( (_DWORD)v8 == (_DWORD)v9 )
    break;
  result = 1000 * (v12 - v13) + ((unsigned int)result - v14) / 0xF4240;
}
while ( (unsigned int)result < 0x1388 );
if ( (_DWORD)v8 != (_DWORD)v9 )
{
  v10 = "sendPacket - timeout - packet failed to send";
  goto LABEL_20;
}
```

*Figure 31. The one send packet cycle in AppleBCM5701Ethernet*

If we only reference the reverse code of the *transmitPacket* function in AppleBCM5701Ethernet, it will be difficult to get how it transmits the packet. Luckily, there are many open source Ethernet drivers in GitHub as mentioned before, so we can research those codes such as "RTL8111::outputStart" in the RealtekRTL8111.cpp file or the "IntelMausi::outputStart" function in the IntelMausiEthernet.cpp file.

To transmit the packet, follow these steps:

1) Prepare the packet header and command bits according to the network protocol such as IPV4 or IPV6, as shown in Figure 32
2) Get the physical segments of packet and compute the VLAN tag, as shown in Figure 33
3) Set the VLAN tag for the descriptors in physical segments, as shown in Figure 34
4) Update the polling bits in the register

```
if (mbuf_get_tso_requested(m, &tsoFlags, &mssValue)) {
    DebugLog("Ethernet [RealtekRTL8111]: mbuf_get_tso_requested() failed. Dropping packet.\n");
    freePacket(m);
    continue;
}
if (tsoFlags & (MBUF_TSO_IPV4 | MBUF_TSO_IPV6)) {                    step 1
    if (tsoFlags & MBUF_TSO_IPV4) {
        getTso4Command(&cmd, &opts2, mssValue, tsoFlags);
    } else {
        /* The pseudoheader checksum has to be adjusted first. */
        adjustIPv6Header(m);
        getTso6Command(&cmd, &opts2, mssValue, tsoFlags);
    }
} else {
    /* We use mssValue as a dummy here because it isn't needed anymore. */
    mbuf_get_csum_requested(m, &checksums, &mssValue);
    getChecksumCommand(&cmd, &opts2, checksums);
}
```

*Figure 32. Prepare the packet header according to the network protocol*

```
/* Finally get the physical segments. */     step 2
numSegs = txMbufCursor->getPhysicalSegmentsWithCoalesce(m, &txSegments[0], kMaxSegs);

/* Alloc required number of descriptors. As the descriptor which has been freed last must be
 * considered to be still in use we never fill the ring completely but leave at least one
 * unused.
 */
if (!numSegs) {
    DebugLog("Ethernet [RealtekRTL8111]: getPhysicalSegmentsWithCoalesce() failed. Dropping packet.\n");
    freePacket(m);
    continue;
}
OSAddAtomic(-numSegs, &txNumFreeDesc);
index = txNextDescIndex;
txNextDescIndex = (txNextDescIndex + numSegs) & kTxDescMask;
firstDesc = &txDescArray[index];
lastSeg = numSegs - 1;

/* Next fill in the VLAN tag. */              step 3
opts2 |= (getVlanTagDemand(m, &vlanTag)) ? (OSSwapInt16(vlanTag) | TxVlanTag) : 0;
```

*Figure 33. Get the physical segments and VLAN tag*

```
/* And finally fill in the descriptors. */              step 4
for (i = 0; i < numSegs; i++) {
    desc = &txDescArray[index];
    opts1 = (((UInt32)txSegments[i].length) | cmd);
    opts1 |= (i == 0) ? FirstFrag : DescOwn;

    if (i == lastSeg) {
        opts1 |= LastFrag;
        txMbufArray[index] = m;
    } else {
        txMbufArray[index] = NULL;
    }
    if (index == kTxLastDesc)
        opts1 |= RingEnd;

    desc->addr = OSSwapHostToLittleInt64(txSegments[i].location);
    desc->opts2 = OSSwapHostToLittleInt32(opts2);
    desc->opts1 = OSSwapHostToLittleInt32(opts1);

    //DebugLog("opts1=0x%x, opts2=0x%x, addr=0x%llx, len=0x%llx\n", opts1, opts2, txSegments[i].location, txSegments[i].length);
    ++index &= kTxDescMask;
}
firstDesc->opts1 |= DescOwn;
```

*Figure 34. Set the VLAN tag for the descriptor in each segment*

**Implement the packet receive handler**

Figure 35 shows the implementation of the “receive handler” in AppleBCM5701Ethernet. This handler only calls the *receivePackets* function to complete its task. To analyze the *receivePackets* functions, we found that it's not just called by *receivePacket*; many other functions simply call this function to return. Another fact is that RxInterrupt is used for Ethernet to receive frames. Therefore, if other open source extensions implement it, we can simply refer to it. Luckily, it is implemented in RealtekRTL8111 and IntelMausi etc. drivers.

```
if ( LOBYTE(this->member123) && BYTE4(this->member123) )
{
  this->member125 = (__int64)ptk;
  LODWORD(this->member126) = 0;
  clock_get_system_nanotime(&v7, &v8);
  while ( 1 )
  {
    BCM5701Enet::receivePackets(this, 1u, 0LL, 1);
    clock_get_system_nanotime(&v9, &v10);
    result = LODWORD(this->member126);
    if ( (_DWORD)result )
      break;
    if ( (v10 - v8) / 0xF4240u >= timeout_v )
    {
      result = 0LL;
      break;
    }
  }
  *v6 = result;
}
```

*Figure 35. The implementation of receive handlers in AppleBCM5701Ethernet*

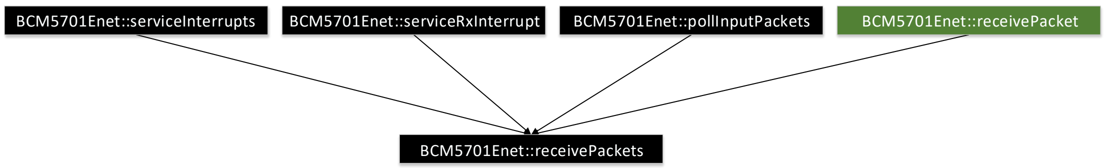



*Figure 36. The call diagram of BCM5701Enet::receivePackets function*

The packet receiving can be seen as the reverse process of packet sending by following these steps:

1) Check the receive register (E1000_RXD_STAT_DD), receive the packet and move it to a new packet with a data buffer, as shown in Figure 37
2) Get the packet's physical segment, its location, and VLAN tag, as shown in Figure 38
3) For the RealtekRTL8111 we are working with, complete the extra length information of newPkt and enqueue the inputPacket queue, as shown in Figure 39. However, the debugger receive handler only receives one packet after calling the receivePacket function and returning it to XNU to parse the debugging command. So, copy the received packet to the reference parameter in the receivePacket function instead of enqueuing it. The copy code can be simply called the memcpy, such as "memcpy(pkt, newPkt, pktSize)"
4) Update the descriptors for the segment if necessary
5) Add the timeout check for receivePacket function to avoid hanging

step 1

```
descStatus2 = OSSwapLittleToHostInt32(desc->opts2);
pktSize = (descStatus1 & 0x1fff) - kIOEthernetCRCSize;
bufPkt = rxMbufArray[rxNextDescIndex];
vlanTag = (descStatus2 & RxVlanTag) ? OSSwapInt16(descStatus2 & 0xffff) : 0;
//DebugLog("rxInterrupt(): descStatus1=0x%x, descStatus2=0x%x, pktSize=%u\n", descStatus1, descStatus2, pktSize);

newPkt = replaceOrCopyPacket(&bufPkt, pktSize, &replaced);

if (!newPkt) {
    /* Allocation of a new packet failed so that we must leave the original packet in place. */
    DebugLog("Ethernet [RealtekRTL8111]: replaceOrCopyPacket() failed.\n");
    etherStats->dot3RxExtraEntry.resourceErrors++;
    opts1 |= kRxBufferPktSize;
    goto nextDesc;
}
```

*Figure 37. Receive the packet and copy it to the new packet buffer*

step 2

```
if (replaced) {
    if (rxMbufCursor->getPhysicalSegments(bufPkt, &rxSegment, 1) != 1) {
        DebugLog("Ethernet [RealtekRTL8111]: getPhysicalSegments() failed.\n");
        etherStats->dot3RxExtraEntry.resourceErrors++;
        freePacket(bufPkt);
        opts1 |= kRxBufferPktSize;
        goto nextDesc;
    }
    opts1 |= ((UInt32)rxSegment.length & 0x0000ffff);
    addr = rxSegment.location;
    rxMbufArray[rxNextDescIndex] = bufPkt;
} else {
    opts1 |= kRxBufferPktSize;
}
getChecksumResult(newPkt, descStatus1, descStatus2);

/* Also get the VLAN tag if there is any. */
if (vlanTag)
    setVlanTag(newPkt, vlanTag);
```

*Figure 38. Get the physical segment and its location*

step 3

```
mbuf_pkthdr_setlen(newPkt, pktSize);
mbuf_setlen(newPkt, pktSize);
interface->enqueueInputPacket(newPkt, pollQueue);
goodPkts++;
```

should copy it to pkt instead of enqueue

*Figure 39. Set the newPkt buffer length and enqueue input packet*

After overriding the send and receive handler, the Ethernet extensions can support remote kernel debugging. However, to control the active debugger, the enable and disable virtual methods should also be overriden. You can refer to the *IONetworkInterface* enable and disable functions in RealtekRTL8111 for more details.

# 3. Zero Day vulnerabilities found by LLDBFuzzer

This section analyzes vulnerabilities with root causes that we know of.

## 3.1. OOB read vulnerability found in AMDRadeonX4000_AMDAccelResource Initialize Process (CVE-2019-8519)

```
* thread #1, stop reason = signal SIGSTOP

  * frame #0: 0xffffff7fa00965d3
AMDRadeonX4000`AMDRadeonX4000_AMDAccelResource::initialize(IOAccelNewResourceArgs*, unsigned long long) + 1525

    frame #1: 0xffffff7f9fea346b IOAcceleratorFamily2`IOAccelSharedUserClient2::new_resource(IOAccelNewResourceArgs*, IOAccelNewResourceReturnData*, unsigned long long, unsigned int*) + 1893

    frame #2: 0xffffff7f9fea4a41 IOAcceleratorFamily2`IOAccelSharedUserClient2::s_new_resource(IOAccelSharedUserClient2*, void*, IOExternalMethodArguments*) + 151

    frame #3: 0xffffff801d625ab8 kernel.development`IOUserClient::externalMethod(this=<unavailable>, selector=<unavailable>, args=0xffffff83dd4b3b58, dispatch=0xffffff7f9fee8260, target=0xffffff80854fd780, reference=0x0000000000000000) at IOUserClient.cpp:5358 [opt]

    frame #4: 0xffffff7f9fea4d98 IOAcceleratorFamily2`IOAccelSharedUserClient2::externalMethod(unsigned int, IOExternalMethodArguments*, IOExternalMethodDispatch*, OSObject*, void*) + 120

    frame #5: 0xffffff801d62eb7f kernel.development`::is_io_connect_method(connection=0xffffff80854fd780, selector=0, scalar_input=<unavailable>, scalar_inputCnt=<unavailable>, inband_input=<unavailable>, inband_inputCnt=2424, ool_input=0, ool_input_size=0, inband_output="", inband_outputCnt=0xffffff806ba03e0c, scalar_output=0xffffff83dd4b3ce0, scalar_outputCnt=0xffffff83dd4b3cdc, ool_output=0, ool_output_size=0xffffff8085919d5c) at IOUserClient.cpp:3994 [opt]

    frame #6: 0xffffff801cfbbce4 kernel.development`_Xio_connect_method(InHeadP=<unavailable>, OutHeadP=0xffffff806ba03de0) at device_server.c:8379 [opt]

    frame #7: 0xffffff801ce8d27d kernel.development`ipc_kobject_server(request=0xffffff8085919000, option=<unavailable>) at ipc_kobject.c:359 [opt]

    frame #8: 0xffffff801ce59465 kernel.development`ipc_kmsg_send(kmsg=0xffffff8085919000, option=3, send_timeout=0) at ipc_kmsg.c:1832 [opt]

    frame #9: 0xffffff801ce78a75 kernel.development`mach_msg_overwrite_trap(args=<unavailable>) at mach_msg.c:549 [opt]

    frame #10: 0xffffff801cff6323 kernel.development`mach_call_munger64(state=0xffffff806ca9c480) at bsd_i386.c:573 [opt]
```

frame #11: 0xffffff801ce23486 kernel.development`hndl_mach_scall64 + 22

*Figure 40. Crash backtrace ZDI-19-569*

**Root cause analysis**

This vulnerability could allow an attacker access to restricted memory.

As shown in the table below, the register of rax is the address of the buffer that is created from the IOMalloc function. The r15 register is pointing to the structureInput buffer, which is controlled by usermode. The ecx register stores the length of IOMalloc buffer, and the rdx register is used as an index to copy the structureInput buffer content to IOMalloc buffer. However, here, ecx is taken directly from the usermode, which is structureInput offset 62 dword. If we set ecx at a big value, it will read overflow from the structureInput buffer.

```
__text:000000000000E58E loc_E58E:                 ; CODE XREF:
AMDRadeonX4000_AMDAccelResource::initialize(IOAccelNewResourceArgs *,ulong long)+58Dj

__text:000000000000E58E          mov    ecx, [r15+0F8h]

__text:000000000000E595          test rcx, rcx

__text:000000000000E598          jz     short loc_E603

__text:000000000000E59A          shl    rcx, 3

__text:000000000000E59E          lea    rdi, [rcx+rcx*2]

__text:000000000000E5A2          call  _IOMalloc

__text:000000000000E5A7          mov    [r12+178h], rax --- rax== buffer address which is created by IOMalloc

__text:000000000000E5AF          test rax, rax

__text:000000000000E5B2          jz     short loc_E62A

__text:000000000000E5B4          or     byte ptr [r12+186h], 8

__text:000000000000E5BD          mov    ecx, [r15+0F8h] -------- r15==structureInput, ecx=( (uint32_t*) structureInput+62)

__text:000000000000E5C4          mov    [r12+180h], ecx

__text:000000000000E5CC          test rcx, rcx

__text:000000000000E5CF          jz     short loc_E639

__text:000000000000E5D1          xor    edx, edx

__text:000000000000E5D3
```

```
__text:000000000000E5D3 loc_E5D3:                          ; CODE XREF:
AMDRadeonX4000_AMDAccelResource::initialize(IOAccelNewResourceArgs *,ulong long)+621j

__text:000000000000E5D3              mov     rsi, [r15+rdx+98h] ----- mov structureInput+rdx+0x98 to rsi

__text:000000000000E5DB              mov     [rax+rdx], rsi ---- mov rsi to rax+rdx, rax== buffer address which is created by
IOMalloc

__text:000000000000E5DF              mov     rsi, [r15+rdx+0A0h]

__text:000000000000E5E7              mov     [rax+rdx+8], rsi

__text:000000000000E5EC              mov     esi, [r15+rdx+0A8h]

__text:000000000000E5F4              mov     [rax+rdx+10h], esi

__text:000000000000E5F8              add     rdx, 18h

__text:000000000000E5FC              dec     rcx

__text:000000000000E5FF              jnz     short loc_E5D3
```

*Table 7. The asm code snippet of AMDRadeonX4000_AMDAccelResource::initialize*

## 3.2. OOB read vulnerability found in AMDRadeonX4000_AMDAccelResource Initialize Process (CVE-2019-8692)

```
(lldb) bt

* thread #1, stop reason = signal SIGSTOP

    frame #0: 0xffffff7f9dcd9459
AMDRadeonX4000`AMDRadeonX4000_AMDAccelResource::initialize(IOAccelNewResourceArgs*, unsigned long long) + 947

    frame #1: 0xffffff7f9dc345ee IOAcceleratorFamily2`IOAccelSharedUserClient2::new_resource(IOAccelNewResourceArgs*,
IOAccelNewResourceReturnData*, unsigned long long, unsigned int*) + 1886

    frame #2: 0xffffff7f9dc35bb5 IOAcceleratorFamily2`IOAccelSharedUserClient2::s_new_resource(IOAccelSharedUserClient2*,
void*, IOExternalMethodArguments*) + 151

    frame #3: 0xffffff801b424978 kernel.development`IOUserClient::externalMethod(this=<unavailable>, selector=<unavailable>,
args=0xffffffa76a5bb9b8, dispatch=0xffffff7f9dc79260, target=<unavailable>, reference=<unavailable>) at
IOUserClient.cpp:5689 [opt]

    frame #4: 0xffffff7f9dc35f0b IOAcceleratorFamily2`IOAccelSharedUserClient2::externalMethod(unsigned int,
IOExternalMethodArguments*, IOExternalMethodDispatch*, OSObject*, void*) + 119

  * frame #5: 0xffffff801b42da02 kernel.development`::is_io_connect_method(connection=<unavailable>, selector=0,
scalar_input=<unavailable>, scalar_inputCnt=<unavailable>, inband_input=<unavailable>, inband_inputCnt=2424, ool_input=0,
```

ool_input_size=0, inband_output="", inband_outputCnt=0xffffff80bf24e60c, scalar_output=0xffffffa76a5bbce0, scalar_outputCnt=0xffffffa76a5bbcdc, ool_output=0, ool_output_size=0xffffff80beec9d5c) at IOUserClient.cpp:4304 [opt]

frame #6: 0xffffff801adbc386 kernel.development`_Xio_connect_method(InHeadP=<unavailable>, OutHeadP=0xffffff80bf24e5e0) at device_server.c:8379 [opt]

frame #7: 0xffffff801ac948fd kernel.development`ipc_kobject_server(request=0xffffff80beec9000, option=3) at ipc_kobject.c:361 [opt]

frame #8: 0xffffff801ac6088e kernel.development`ipc_kmsg_send(kmsg=0xffffff80beec9000, option=3, send_timeout=0) at ipc_kmsg.c:1868 [opt]

frame #9: 0xffffff801ac800e3 kernel.development`mach_msg_overwrite_trap(args=<unavailable>) at mach_msg.c:553 [opt]

frame #10: 0xffffff801adf702b kernel.development`mach_call_munger64(state=0xffffff80bd7429a0) at bsd_i386.c:580 [opt]

frame #11: 0xffffff801ac2a476 kernel.development`hndl_mach_scall64 + 22

(lldb) register read

General Purpose Registers:

**rax = 0x0000000000003740**

rbx = 0x00000000000003c8

rcx = 0x0000000000000000

rdx = 0x00000000000003c8

rdi = 0xffffff80cdadd400

rsi = 0xffffff80beec9974

rbp = 0xffffffa76a5bb850

rsp = 0xffffffa76a5bb820

r8 = 0xffffff80cdadd400

r9 = 0xffffff801b6c7210 kernel.development`zone_array + 8336

r10 = 0xffffff801b6c5180 kernel.development`zone_array

r11 = 0x0000000000000000

r12 = 0xffffff80c37dd700

r13 = 0xffffff80beec95ac

r14 = 0x0000000000000001

r15 = 0xffffff80beec93c4

rip = 0xffffff7f9dcd9459 AMDRadeonX4000`AMDRadeonX4000_AMDAccelResource::initialize(IOAccelNewResourceArgs*, unsigned long long) + 947

rflags = 0x0000000000010202

cs = 0x0000000000000008

fs = 0x0000000000000000

gs = 0x0000000000000000

*Figure 41. Crash backtrace CVE-2019-8692*

## Root cause analysis

As shown in the backtrace above, the system will call the *AMDRadeonX4000_AMDAccelResource::initialize* function to initialize an AMD resource object and take structureInput and structureInputSize as parameters (structureInput is the inband input which can be controlled by the userspace directly). As shown in Figure 42, this function will first use the *IOAccelResource2::initialize* function to initialize some resource properties, like BYTE4(this->member21), BYTE5(this->member21), and BYTE6(this->member21), using the same parameters as *AMDRadeonX4000_AMDAccelResource::initialize*.

However, in the following code, *AMDRadeonX4000_AMDAccelResource::initialize* directly uses BYTE6(this->member21) << 6 as the offset to read the buffer of v36. Thus, we can control it and use it to read out of boundary memory.


*Figure 42. Root cause analysis for this OOB vulnerability*

## 3.3. Double free vulnerability found when AMDRadeonX4000_AMDSIGLContext processes a sideband token (CVE-2019-8635)

* thread #1, stop reason = signal SIGSTOP

frame #0: 0xffffff7f8d7adc37 IOAcceleratorFamily2`IOAccelResource2::clientRelease(IOAccelShared2*) + 13

frame #1: 0xffffff7f8d880dad AMDRadeonX4000`AMDRadeonX4000_AMDSIGLContext::process_StretchTex2Tex(IOAccelCommandStreamInfo&) + 2893

frame #2: 0xffffff7f8d79b5d5 IOAcceleratorFamily2`IOAccelContext2::processSidebandBuffer(IOAccelCommandDescriptor*, bool) + 273

frame #3: 0xffffff7f8d8885e4 AMDRadeonX4000`AMDRadeonX4000_AMDSIGLContext::processSidebandBuffer(IOAccelCommandDescriptor*, bool) + 182

frame #4: 0xffffff7f8d79bae7 IOAcceleratorFamily2`IOAccelContext2::processDataBuffers(unsigned int) + 85

frame #5: 0xffffff7f8d7a2380 IOAcceleratorFamily2`IOAccelGLContext2::processDataBuffers(unsigned int) + 804

frame #6: 0xffffff7f8d798c30 IOAcceleratorFamily2`IOAccelContext2::submit_data_buffers(IOAccelContextSubmitDataBuffersIn*, IOAccelContextSubmitDataBuffersOut*, unsigned long long, unsigned long long*) + 1208

frame #7: 0xffffff800b027a3c kernel.development`::shim_io_connect_method_structureI_structureO(method=<unavailable>, object=<unavailable>, input=<unavailable>, inputCount=<unavailable>, output=<unavailable>, outputCount=0xffffff8742023968) at IOUserClient.cpp:0 [opt]

frame #8: 0xffffff800b025ca0 kernel.development`IOUserClient::externalMethod(this=<unavailable>, selector=<unavailable>, args=0xffffff87420239b8, dispatch=0x0000000000000000, target=0x0000000000000000, reference=<unavailable>) at IOUserClient.cpp:5459 [opt]

* frame #9: 0xffffff800b02ebff kernel.development`::is_io_connect_method(connection=0xffffff80b094e000, selector=2, scalar_input=<unavailable>, scalar_inputCnt=<unavailable>, inband_input=<unavailable>, inband_inputCnt=136, ool_input=0, ool_input_size=0, inband_output="", inband_outputCnt=0xffffff80b0d81e0c, scalar_output=0xffffff8742023ce0, scalar_outputCnt=0xffffff8742023cdc, ool_output=0, ool_output_size=0xffffff80ab5c7574) at IOUserClient.cpp:3994 [opt]

frame #10: 0xffffff800a9bbd64 kernel.development`_Xio_connect_method(InHeadP=<unavailable>, OutHeadP=0xffffff8742023ce0) at device_server.c:8379 [opt]

frame #11: 0xffffff800a88d27d kernel.development`ipc_kobject_server(request=0xffffff80ab5c7400, option=<unavailable>) at ipc_kobject.c:359 [opt]

frame #12: 0xffffff800a859465 kernel.development`ipc_kmsg_send(kmsg=0xffffff80ab5c7400, option=3, send_timeout=0) at ipc_kmsg.c:1832 [opt] frame #13: 0xffffff800a878a75 kernel.development`mach_msg_overwrite_trap(args=<unavailable>) at mach_msg.c:549 [opt]

frame #14: 0xffffff800a9f63a3 kernel.development`mach_call_munger64(state=0xffffff80af471bc0) at bsd_i386.c:573 [opt]

frame #15: 0xffffff800a823486 kernel.development`hndl_mach_scall64 + 22

*Figure 43. Crash backtrace CVE-2019-8635*

**Root cause analysis**

This is a double free vulnerability that an attacker can use to gain escalated privileges. We published an in-depth discussion of it in June.

In Figure 44 below, we can see that if v15 equals 0x8c00, the *accelResource_offset8* and *accelResource_offset12* are both taken from IOAccelShared2 with a shared memory offset 24 and 28 value as the index.

This function will release *accelResource_*offset12 from IOAccelShared2 first, and if *accelResource_offset8->member2* is not equal to 10, this function will also release the *accelResource_offset8* from IOAccelShared2. However, if we set the shared memory offsets 24 and 28 to the same value, it will release the same accelResource twice.


*Figure 44. The pseudo code snippet of AMDRadeonX4000_AMDSIGLContext process_StretchTex2Tex function*

From Figure 405 below, we can also see that the shared memory address is pointing to command stream info offset 24, but the command stream info buffer is set in the *IOAccelContext2::processSidebandBuffer* function, as shown in the same figure. We can also see that v5 points to the shareMem offset 16, and this->member196 points to the commandStreamInfo offset 24.

```
v3 = a2;
v4 = (char *)&this->commandStreamInfo;          // cmdinfo start address, can be control by share memory offset 16
LODWORD(this->member199) = 0;
this->commandStreamInfo = 0LL;
this->member194 = 0LL;
LODWORD(this->member195) = 0;
v5 = this->shareMem_start_vm_address_187 + 16;
this->member196 = v5;                           // this->commandStreamInfo + 24
LODWORD(this->member200) = 0;
BYTE4(this->member200) = a3;                    // =1
while ( 1 )
{
  v6 = this->shareMem_end_vm_address_188;
  if ( v5 + 8 > v6 )
  {
    v14 = _os_log_default_0;
    _os_log_internal(
      &dword_0,
      _os_log_default_0,
      17LL,
      IOAccelContext2::processSidebandBuffer(IOAccelCommandDescriptor *,bool)::_os_log_fmt,
      "virtual bool IOAccelContext2::processSidebandBuffer(IOAccelCommandDescriptor *, bool)");
    v15 = LOWORD(this->commandStreamInfo_offset32);
    v16 = WORD1(this->commandStreamInfo_offset32);
    _os_log_internal(
      &dword_0,
      v14,
      17LL,
      IOAccelContext2::setContextError(unsigned int)::_os_log_fmt,
      "void IOAccelContext2::setContextError(uint32_t)");
    goto LABEL_18;
  }
  LOWORD(this->commandStreamInfo_offset32) = *(_WORD *)v5;
  v7 = *(_WORD *)(v5 + 2);
  WORD1(this->commandStreamInfo_offset32) = v7;
  v8 = *(_DWORD *)(v5 + 4);
  HIDWORD(this->commandStreamInfo_offset32) = v8;// set the commandstreaminfo
  this->member198 = v5 + 8;
```

*Figure 405 the pseudo code snippet of IOAccelContext2::processSidebandBuffer*

Figure 46 shows the pseudo code snippet of *IOAccelContext2::clientMemoryForType* function, which is the well-known API "*IOConnectMapMemory64*" that can map a userspace buffer to kernel space. When using the *IOConnectMapMemory64* function, we set the connect object, memory type etc., and other args. Here, the connect object is the instance of IOAccelContext2, and memory type is 0. When we set memory type to 0, the *clientMemoryForType* function will create a buffer memory descriptor and return the start address to userspace, what's more, it will also set the buffer memory address to the "shareMem_start_vm_address_187" var which is named by the user. This var is exactly the value which is used in the *IOAccelContext2::processSidebandBuffer* function. Therefore, we can control the share buffer and set the two resource indexes to the same value, which can trigger the double free bug.

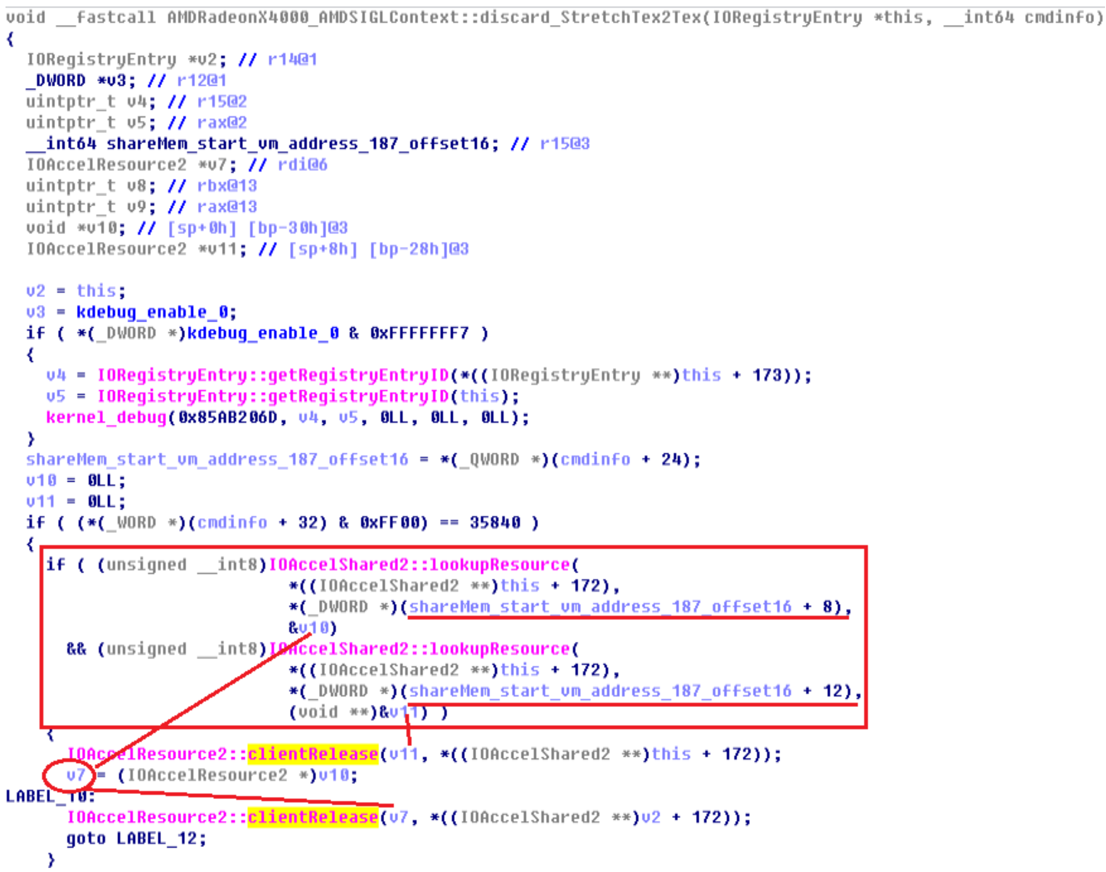
```
if ( !type )
{
  v9 = this->member206;
  if ( v9 < HIDWORD(this->member206)
    && *(_DWORD *)(this->shareMem_start_vm_address_187 + 8) >= (unsigned int)(3
                                                                          * *(_DWORD *)this->shareMem_start_vm_address_187) >> 2 )
  {
    LODWORD(v10) = (*(int (__fastcall **)(__int64, void *, _QWORD, _QWORD))(*(_QWORD *)this->graphicAcc_v
                                                                           + 2336LL))(
                     this->graphicAcc_v,
                     &loc_10023,
                     2 * v9,
                     *(_QWORD *)page_size_0);// IOGraphicsAccelerator2::createBufferMemoryDescriptorWithOptions(uint,ulong,ulong)
    v11 = v10;
    if ( v10 )
    {
      v12 = this->shareMem_type0;
      if ( v12 )
        (*(void (**)(void))(*(_QWORD *)v12 + 40LL))();
      v5->shareMem_type0 = v11;
      LODWORD(v5->member206) = 2 * v9;
    }
    else
    {
      HIDWORD(this->member206) = v9 & 0x7FFFFFFF;
      v11 = this->shareMem_type0;
      if ( !v11 )
      {
        _os_log_internal(
          &dword_0,
          _os_log_default_0,
          17LL,
          IOAccelContext2::clientMemoryForType(unsigned int,unsigned int *,IOMemoryDescriptor **)::_os_log_fmt,
          "virtual IOReturn IOAccelContext2::clientMemoryForType(UInt32, IOOptionBits *, IOMemoryDescriptor **)");
        v7 = -536870210;
        goto LABEL_58;
      }
    }
    LODWORD(v49) = (*(int (__fastcall **)(__int64))(*(_QWORD *)v11 + 736LL))(v11);// IOBufferMemoryDescriptor::getBytesNoCopy(void)
    v5->shareMem_start_vm_address_187 = v49;
    v5->shareMem_end_vm_address_188 = v49 + (v5->member206 & 0xFFFFFFFC);
  }
  (*(void (**)(void))(*(_QWORD *)v5->shareMem_type0 + 32LL))();// OSObject::retain()
  *v55 = 0;
  *memory_ptr = (IOMemoryDescriptor *)v5->shareMem_type0;
  ((void (__fastcall *)(IOAccelContext2 *, __int64, _QWORD, _QWORD))v5->vtable->__ZN15IOAccelContext224initSidebandBufferHeaderEP27IOAccelSidebandBufferHea
    v5,
    v5->shareMem_start_vm_address_187,
    LODWORD(v5->member206),
    HIDWORD(v5->member206));
  goto LABEL_57;
```

*Figure 46 the pseudo code snippet of IOAccelContext2::clientMemoryForType function*

## 3.4. Double free vulnerability found when AMDRadeonX4000_AMDSIGLContext class processes a sideband token (CVE-2019-8635)

From Figure 7, we can see that if (cmdinfo+32) equals to 0x8c00, the IOAccelResource v10 and v11 both "get" from IOAccelShared2 with *(shareMem_start_address_187_offset16+8) and *(shareMem_start_address_187_offset16+12) value as index. This function will then release two accelerator resources using the *IOAccelResource2::clientRelease()* function. However, these two indexes can be directly controlled from user space by map memory with IOAccelContext2 userclient. If userspace maps the same index for lookupResource function, clientRelease will release the same resource client twice, so the double free vulnerability will occur.

The method for controlling the shared memory has been detailed in the above section covering CVE-2019-8635.



```
void __fastcall AMDRadeonX4000_AMDSIGLContext::discard_StretchTex2Tex(IORegistryEntry *this, __int64 cmdinfo)
{
  IORegistryEntry *v2; // r14@1
  _DWORD *v3; // r12@1
  uintptr_t v4; // r15@2
  uintptr_t v5; // rax@2
  __int64 shareMem_start_vm_address_187_offset16; // r15@3
  IOAccelResource2 *v7; // rdi@6
  uintptr_t v8; // rbx@13
  uintptr_t v9; // rax@13
  void *v10; // [sp+0h] [bp-30h]@3
  IOAccelResource2 *v11; // [sp+8h] [bp-28h]@3

  v2 = this;
  v3 = kdebug_enable_0;
  if ( *(_DWORD *)kdebug_enable_0 & 0xFFFFFFF7 )
  {
    v4 = IORegistryEntry::getRegistryEntryID(*((IORegistryEntry **)this + 173));
    v5 = IORegistryEntry::getRegistryEntryID(this);
    kernel_debug(0x85AB206D, v4, v5, 0LL, 0LL, 0LL);
  }
  shareMem_start_vm_address_187_offset16 = *(_QWORD *)(cmdinfo + 24);
  v10 = 0LL;
  v11 = 0LL;
  if ( (*(_WORD *)(cmdinfo + 32) & 0xFF00) == 35840 )
  {
    if ( (unsigned __int8)IOAccelShared2::lookupResource(
                            *((IOAccelShared2 **)this + 172),
                            *(_DWORD *)(shareMem_start_vm_address_187_offset16 + 8),
                            &v10)
      && (unsigned __int8)IOAccelShared2::lookupResource(
                            *((IOAccelShared2 **)this + 172),
                            *(_DWORD *)(shareMem_start_vm_address_187_offset16 + 12),
                            (void **)&v11) )
    {
      IOAccelResource2::clientRelease(v11, *((IOAccelShared2 **)this + 172));
      v7 = (IOAccelResource2 *)v10;
LABEL_10:
      IOAccelResource2::clientRelease(v7, *((IOAccelShared2 **)v2 + 172));
      goto LABEL_12;
    }
```

*Figure 47. The pseudo code snippet of AMDRadeonX4000_AMDSIGLContext: discard_StretchTex2Tex function*

## 3.5. OOB vulnerability found in the AMDRadeonX4000_AMDAccelSharedUserClient RsrcAndXorByteFlag function (CVE-2019-8691)

(lldb) bt

* thread #1, stop reason = signal SIGSTOP

  * frame #0: 0xffffff7f849d49a0 AMDRadeonX4000`AMDRadeonX4000_AMDAccelResource::AndXorByteFlag(unsigned short, unsigned char, unsigned char) + 164

frame #1: 0xffffff7f849dad9d AMDRadeonX4000`AMDRadeonX4000_AMDAccelSharedUserClient::RsrcAndXorByteFlag(AMDRsrcAndXorByteFlagPacket const*, unsigned long long*) + 275

frame #2: 0xffffff8001c27a3c kernel.development`::shim_io_connect_method_structureI_structureO(method=<unavailable>, object=<unavailable>, input=<unavailable>, inputCount=<unavailable>, output=<unavailable>, outputCount=0xffffffa77393bab8) at IOUserClient.cpp:0:9 [opt]

frame #3: 0xffffff8001c25ca0 kernel.development`IOUserClient::externalMethod(this=<unavailable>, selector=<unavailable>, args=0xffffffa77393bb58, dispatch=0x0000000000000000, target=0x0000000000000000, reference=<unavailable>) at IOUserClient.cpp:5459:9 [opt]

frame #4: 0xffffff7f8493af0b IOAcceleratorFamily2`IOAccelSharedUserClient2::externalMethod(unsigned int, IOExternalMethodArguments*, IOExternalMethodDispatch*, OSObject*, void*) + 119

frame #5: 0xffffff8001c2ebff kernel.development`::is_io_connect_method(connection=0xffffff80bff43fd0, selector=262, scalar_input=<unavailable>, scalar_inputCnt=<unavailable>, inband_input=<unavailable>, inband_inputCnt=12, ool_input=0, ool_input_size=0, inband_output="", inband_outputCnt=0xffffff80bfc3260c, scalar_output=0xffffffa77393bce0, scalar_outputCnt=0xffffffa77393bcdc, ool_output=0, ool_output_size=0xffffff809d1e0b0c) at IOUserClient.cpp:3994:19 [opt]

frame #6: 0xffffff80015bbd64 kernel.development`_Xio_connect_method(InHeadP=<unavailable>, OutHeadP=0xffffff80bfc325e0) at device_server.c:8379:18 [opt]

frame #7: 0xffffff800148d27d kernel.development`ipc_kobject_server(request=0xffffff809d1e0a40, option=<unavailable>) at ipc_kobject.c:359:3 [opt]

frame #8: 0xffffff8001459465 kernel.development`ipc_kmsg_send(kmsg=0xffffff809d1e0a40, option=3, send_timeout=0) at ipc_kmsg.c:1832:10 [opt]

frame #9: 0xffffff8001478a75 kernel.development`mach_msg_overwrite_trap(args=<unavailable>) at mach_msg.c:549:8 [opt]

frame #10: 0xffffff80015f63a3 kernel.development`mach_call_munger64(state=0xffffff80be434b20) at bsd_i386.c:573:24 [opt]

frame #11: 0xffffff8001423486 kernel.development`hndl_mach_scall64 + 22

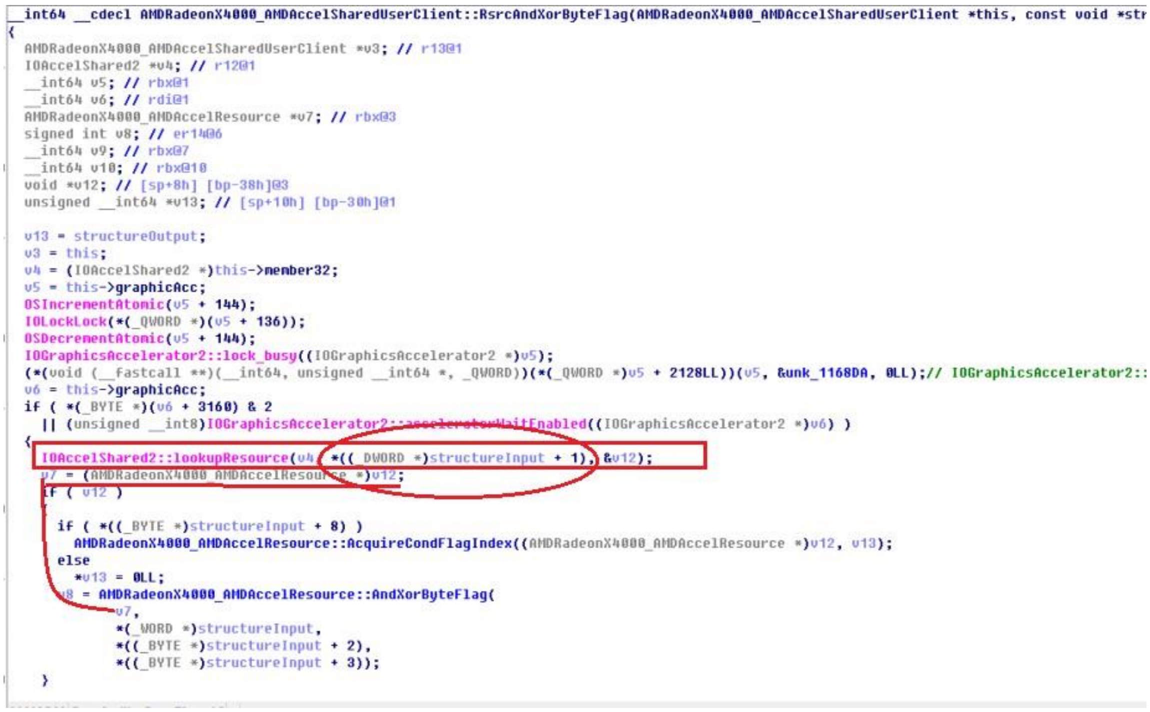
(lldb) register read
General Purpose Registers:
rax = 0x00b600d000b50128
rbx = 0x0000000000d20119
rcx = 0x0000000000000000
rdx = 0x0000000000000000
rdi = 0xffffff80b333a710
rsi = 0x0000000000000000
rbp = 0xffffffa77393b9f0
rsp = 0xffffffa77393b9c0
r8 = 0xffffffa77393bab8
r9 = 0x0000000000000000
r10 = 0xffffff80bfc32610
r11 = 0xffffff7f849dac8a AMDRadeonX4000`AMDRadeonX4000_AMDAccelSharedUserClient::RsrcAndXorByteFlag(AMDRsrcAndXorByteFlagPacket const*, unsigned long long*)
r12 = 0x0000000000000000
r13 = 0xffffff80b333a710
r14 = 0xffffff809d1e0ae0

```
    r15 = 0x0000000000000000
    rip = 0xffffff7f849d49a0 AMDRadeonX4000`AMDRadeonX4000_AMDAccelResource::AndXorByteFlag(unsigned short, unsigned char, unsigned char) + 164
 rflags = 0x0000000000010202
     cs = 0x0000000000000008
     fs = 0x0000000000000000
     gs = 0x0000000000000000
(lldb) dis
   0xffffff7f849d4990 <+148>: cmpl  %r12d, %ebx
   0xffffff7f849d4993 <+151>: jbe    0xffffff7f849d49ad        ; <+177>
   0xffffff7f849d4995 <+153>: movq 0x1c8(%r13), %rax
   0xffffff7f849d499c <+160>: movzwl %r12w, %edx
-> 0xffffff7f849d49a0 <+164>: andb (%rax,%rdx), %r15b
   0xffffff7f849d49a4 <+168>: xorb %cl, %r15b
   0xffffff7f849d49a7 <+171>: movb %r15b, (%rax,%rdx)
   0xffffff7f849d49ab <+175>: xorl %eax, %eax
   0xffffff7f849d49ad <+177>: addq $0x8, %rsp
   0xffffff7f849d49b1 <+181>: popq %rbx
   0xffffff7f849d49b2 <+182>: popq %r12
   0xffffff7f849d49b4 <+184>: popq %r13
   0xffffff7f849d49b6 <+186>: popq %r14
   0xffffff7f849d49b8 <+188>: popq %r15
   0xffffff7f849d49ba <+190>: popq %rbp
   0xffffff7f849d49bb <+191>: retq
```

*Figure 48. Crash backtrace CVE-2019-8691*

### Root cause analysis

In Figure 49, we can see that *RsrcAndXorByteFlag* function will first look up an AMDRadeonX4000_AMDAccelResource object from the IOAccelShared2 with "structureInput + 1" as the index. However, the structureInput is the buffer input from user space, and the system does not check for it. So, we can index any accelerator resource as our operation object, and use it as the parameter for the *AMDRadeonX4000_AMDAccelResource::AndXorByteFlag* function. The other three parameters can also be directly controlled from user space.



```
__int64 __cdecl AMDRadeonX4000_AMDAccelSharedUserClient::RsrcAndXorByteFlag(AMDRadeonX4000_AMDAccelSharedUserClient *this, const void *str
{
  AMDRadeonX4000_AMDAccelSharedUserClient *v3; // r13@1
  IOAccelShared2 *v4; // r12@1
  __int64 v5; // rbx@1
  __int64 v6; // rdi@1
  AMDRadeonX4000_AMDAccelResource *v7; // rbx@3
  signed int v8; // er14@6
  __int64 v9; // rbx@7
  __int64 v10; // rbx@10
  void *v12; // [sp+8h] [bp-38h]@3
  unsigned __int64 *v13; // [sp+10h] [bp-30h]@1

  v13 = structureOutput;
  v3 = this;
  v4 = (IOAccelShared2 *)this->member32;
  v5 = this->graphicAcc;
  OSIncrementAtomic(v5 + 144);
  IOLockLock(*(_QWORD *)(v5 + 136));
  OSDecrementAtomic(v5 + 144);
  IOGraphicsAccelerator2::lock_busy((IOGraphicsAccelerator2 *)v5);
  (*(void (__fastcall **)(__int64, unsigned __int64 *, _QWORD))(*(_QWORD *)v5 + 2128LL))(v5, &unk_1168DA, 0LL);// IOGraphicsAccelerator2::
  v6 = this->graphicAcc;
  if ( *(_BYTE *)(v6 + 3160) & 2
    || (unsigned __int8)IOGraphicsAccelerator2::acceleratorWaitEnabled((IOGraphicsAccelerator2 *)v6) )
  {
    IOAccelShared2::lookupResource(v4, *((_DWORD *)structureInput + 1), &v12);
    v7 = (AMDRadeonX4000_AMDAccelResource *)v12;
    if ( v12 )
    {
      if ( *((_BYTE *)structureInput + 8) )
        AMDRadeonX4000_AMDAccelResource::AcquireCondFlagIndex((AMDRadeonX4000_AMDAccelResource *)v12, v13);
      else
        *v13 = 0LL;
      v8 = AMDRadeonX4000_AMDAccelResource::AndXorByteFlag(
             v7,
             *(_WORD *)structureInput,
             *((_BYTE *)structureInput + 2),
             *((_BYTE *)structureInput + 3));
  }
```

*Figure 49. Code snippet of AMDRadeonX4000_AMDAccelSharedUserClient::RsrcAndXorByteFlag function*

As seen in Table 8, the AndXorByteFlag function uses two values, one is the value which "rdi+0x1d0" points to — our research found that it is a buffer size. The other one is the value of "r13+1C8h", which is actually equal to "rdi+0x1c8", which is a buffer start address.

From the table below, we can see that this function includes the following vulnerabilities:

- If we input an invalid index to lookup the Resource, the *IOAccelShared2::lookupResource(IOAccelShared2 *this, unsigned int a2, void **a3)* function will return '1' for a3. It is strange, but it actually happened, so crash point 1 will occur due to the access to protected memory.
- If we input a valid index and lookup a resource but the resource is not a good one, then its buffer start address becomes an invalid address. It is like the value of RAX register as seen in the above Figure 48 (the register read instruction, highlighted in red).
- If we input a valid index and also lookup a good resource, however, a bad rdx value in crash point 2 can be controlled from user space. It also an OOB vulnerability.

| | |
|---|---|
| __text:00000000000148FC | push rbp |
| __text:00000000000148FD | mov rbp, rsp |
| __text:0000000000014900 | push r15 |
| __text:0000000000014902 | push r14 |
| __text:0000000000014904 | push r13 |
| __text:0000000000014906 | push r12 |
| __text:0000000000014908 | push rbx |

```
__text:0000000000014909          push  rax
__text:000000000001490A          mov r15d, edx
__text:000000000001490D          mov r12d, esi
__text:0000000000014910          mov r13, rdi
__text:0000000000014913          mov     ebx, [rdi+1D0h]        // ebx is value of the resource object offset 0x1D0 crash point 1
__text:0000000000014919          cmp     ebx, esi             //compare [rdi+0x1d0] with the second parameter
__text:000000000001491B          ja      short loc_1498B        //if great than second para, then jump to loc_1498B
```

*--- omitted code ---*

```
__text:000000000001498B loc_1498B:                 ; CODE XREF: AMDRadeonX4000_AMDAccelResource::AndXorByteFlag(ushort,uchar,uchar)+1Fj
__text:000000000001498B          mov eax, 0E00002BDh
__text:0000000000014990          cmp ebx, r12d
__text:0000000000014993          jbe short loc_149AD
__text:0000000000014995          mov rax, [r13+1C8h] //here, rax is the value which rdi+0x1c8 point to. It actually is a buffer start address
__text:000000000001499C          movzx edx, r12w
__text:00000000000149A0          and r15b, [rax+rdx] // rax can be controlled by index different resource object. And rdx can be controlled by
                                 userspace structure input crash point2
__text:00000000000149A4          xor r15b, cl
__text:00000000000149A7          mov [rax+rdx], r15b
__text:00000000000149AB          xor eax, eax
__text:00000000000149AD
__text:00000000000149AD loc_149AD:                 ; CODE XREF: AMDRadeonX4000_AMDAccelResource::AndXorByteFlag(ushort,uchar,uchar)+97j
__text:00000000000149AD          add rsp, 8
__text:00000000000149B1          pop  rbx
```

*--- omitted code ---*

```
__text:00000000000149BB_ZN31AMDRadeonX4000_AMDAccelResource14AndXorByteFlagEthh endp
```

*Table 8. The assembly code snippet of AMDRadeonX4000_AMDAccelResource::AndXorByteFlag function*

## 3.6. EoP (elevation of privilege) bug found in IOAccelSharedUserClient2 start process (CVE-2019-8616)

```
(lldb) bt

* thread #1, stop reason = signal SIGSTOP

  * frame #0: 0xffffff8012ba4050 kernel.development`memcpy + 11

    frame #1: 0xffffff7f98f0358b AppleIntelHD5000Graphics`IntelAccelerator::newGTT(unsigned int**, bool, IGAccelTask&) + 173

    frame #2: 0xffffff7f98eebce8 AppleIntelHD5000Graphics`IntelPPGTT::init(IntelAccelerator&, bool, IGAccelTask&) + 24

    frame #3: 0xffffff7f98ef47dc AppleIntelHD5000Graphics`IGAccelTask::prepare(IntelAccelerator&) + 38

    frame #4: 0xffffff7f98f0348b AppleIntelHD5000Graphics`IntelAccelerator::createUserGPUTask() + 219

    frame #5: 0xffffff7f98980382 IOAcceleratorFamily2`IOAccelShared2::init(IOGraphicsAccelerator2*, task*) + 48

    frame #6: 0xffffff7f9899513b IOAcceleratorFamily2`IOGraphicsAccelerator2::createShared(task*) + 51

    frame #7: 0xffffff7f98983921 IOAcceleratorFamily2`IOAccelSharedUserClient2::sharedStart() + 43

    frame #8: 0xffffff7f98ee4e22 AppleIntelHD5000Graphics`IGAccelSharedUserClient::sharedStart() + 22

    frame #9: 0xffffff7f9898191a IOAcceleratorFamily2`IOAccelSharedUserClient2::start(IOService*) + 156

    frame #10: 0xffffff7f98994a1a IOAcceleratorFamily2`IOGraphicsAccelerator2::newUserClient(task*, void*, unsigned int,
IOUserClient**) + 1088

    frame #11: 0xffffff80133c9bc1 kernel.development`IOService::newUserClient(this=0xffffff8037dc4800,
owningTask=0xffffff803be31760, securityID=0xffffff803be31760, type=6, properties=0x0000000000000000,
handler=0xffffff9214a2bd10) at IOService.cpp:5856 [opt]

    frame #12: 0xffffff801342ce60 kernel.development`::is_io_service_open_extended(_service=0xffffff8037dc4800,
owningTask=0xffffff803be31760, connect_type=6, ndr=<unavailable>, properties=<unavailable>, propertiesCnt=<unavailable>,
result=0xffffff804e2b9bb8, connection=0xffffff9214a2bd60) at IOUserClient.cpp:3491 [opt]

    frame #13: 0xffffff8012dba714 kernel.development`_Xio_service_open_extended(InHeadP=0xffffff8046905504,
OutHeadP=0xffffff804e2b9b7c) at device_server.c:8003 [opt]

    frame #14: 0xffffff8012c8c27d kernel.development`ipc_kobject_server(request=0xffffff80469054a0, option=<unavailable>)
at ipc_kobject.c:359 [opt]

    frame #15: 0xffffff8012c58465 kernel.development`ipc_kmsg_send(kmsg=0xffffff80469054a0, option=3, send_timeout=0) at
ipc_kmsg.c:1832 [opt]

    frame #16: 0xffffff8012c77a75 kernel.development`mach_msg_overwrite_trap(args=<unavailable>) at mach_msg.c:549 [opt]

    frame #17: 0xffffff8012df52c3 kernel.development`mach_call_munger64(state=0xffffff803c0fea00) at bsd_i386.c:573 [opt]

    frame #18: 0xffffff8012c22486 kernel.development`hndl_mach_scall64 + 22
```

(lldb)

*Figure 50. Crash backtrace CVE-2019-8616*

**Root cause analysis**

This vulnerability can also be used to gain escalated privileges.

From Table 9 below, we can see that the memcpy destination address is the return value of the *IOAccelSysMemory::lockForCPUAccess* function. However, Table 10 shows that there are many places where the *IOAccelSysMemory::lockForCPUAccess* function will return an invalid address. Therefore, the memcpy is not secure here.

```
__text:0000000000027537        call    ZN16IOAccelSysMemory16lockForCPUAccessEP4taskj ;
IOAccelSysMemory::lockForCPUAccess(task *,uint)

__text:000000000002753C        mov     [r13+0], rax

__text:0000000000027540        test    r12b, r12b -----------------here, it will test r12b, and jmp to loc_2756C

__text:0000000000027543        jz      short loc_2756C

__text:0000000000027545        mov     rcx, [rbx+1118h]

__text:000000000002754C        test rcx, rcx

__text:000000000002754F        jz      short loc_275B9

__text:0000000000027551        mov     rdx, [rbx+1110h]

__text:0000000000027558        xor     esi, esi

__text:000000000002755A

__text:000000000002755A loc_2755A:                     ; CODE XREF: IntelAccelerator::newGTT(uint **,bool,IGAccelTask &)+8Aj

__text:000000000002755A        mov     edi, [rdx+rsi]

__text:000000000002755D        mov     ebx, esi

__text:000000000002755F        mov     [rax+rbx], edi

__text:0000000000027562        lea     esi, [rsi+4]

__text:0000000000027565        cmp     rcx, rsi

__text:0000000000027568        ja      short loc_2755A

__text:000000000002756A        jmp     short loc_275B9

__text:000000000002756C ; ---------------------------------------------------------------------------
```

```
__text:000000000002756C
__text:000000000002756C loc_2756C:                ; CODE XREF: IntelAccelerator::newGTT(uint **,bool,IGAccelTask &)+65j
__text:000000000002756C           mov    rcx, [rbx+160h] ---------------------memcpy len
__text:0000000000027573           mov    rsi, [rcx+268h] ; void * -----------------memcpy source address
__text:000000000002757A           mov    edx, [rbx+1138h]
__text:0000000000027580           shr    edx, 0Ah       ; size_t
__text:0000000000027583           mov    rdi, rax       ; void * --------memcpy destination address here, just move rax to rdi,
however, rax is the return value of ZN16IOAccelSysMemory16lockForCPUAccessEP4taskj function
__text:0000000000027586           call  _memcpy
__text:000000000002758B           mov    esi, [rbx+1140h] ; unsigned_int64
__text:0000000000027591           mov    edx, [rbx+1148h] ; unsigned_int64
__text:0000000000027597           mov    rdi, rbx       ; this
```

*Table 9. The asm code snippet of IntelAccelerator::newGTT*

```
__text:000000000004740B loc_4740B:                ; CODE XREF: IOAccelSysMemory::lockForCPUAccess(task *,uint)+102j
__text:000000000004740B                           ; IOAccelSysMemory::lockForCPUAccess(task *,uint)+1D1j ...
__text:000000000004740B           mov    rax, rbx
__text:000000000004740E           add    rsp, 8
__text:0000000000047412           pop    rbx
__text:0000000000047413           pop    r14
__text:0000000000047415           pop    r15
__text:0000000000047417           pop    rbp
__text:0000000000047418           retn
__text:0000000000047419 ; ---------------------------------------------------------------------------
__text:0000000000047419
__text:0000000000047419 loc_47419:                ; CODE XREF: IOAccelSysMemory::lockForCPUAccess(task *,uint)+181j
__text:0000000000047419           lea    rdi, dword_0
__text:0000000000047420           mov    rsi, cs:_os_log_default_0
```

```
__text:0000000000047427             lea     rcx,_ZZN16IOAccelSysMemory16lockForCPUAccessEP4taskjE11_os_log_fmt_1 ;
"%s: failed to create map.\n"

__text:000000000004742E             lea     r8, aMach_vm_addr_0 ; "mach_vm_address_t IOAccelSysMemory::loc"...

__text:0000000000047435             xor     ebx, ebx

__text:0000000000047437             mov     edx, 11h

__text:000000000004743C             xor     eax, eax            ---eax =0      --1)

__text:000000000004743E             call_os_log_internal

__text:0000000000047443             jmp     short loc_4740B     ---return eax

__text:0000000000047445 ; ---------------------------------------------------------------------------

__text:0000000000047445

__text:0000000000047445 loc_47445:                      ; CODE XREF: IOAccelSysMemory::lockForCPUAccess(task *,uint)+13Aj

__text:0000000000047445             lea     rdi, dword_0

__text:000000000004744C             mov     rsi, cs:_os_log_default_0

__text:0000000000047453             lea     rcx,_ZZN16IOAccelSysMemory16lockForCPUAccessEP4taskjE11_os_log_fmt ; "%s:
createMappingInTask failed to creat"...

__text:000000000004745A             lea     r8, aMach_vm_addr_0 ; "mach_vm_address_t IOAccelSysMemory::loc"...

__text:0000000000047461             xor     ebx, ebx

__text:0000000000047463             mov     edx, 11h

__text:0000000000047468             xor     eax, eax            ---eax =0     --2)

__text:000000000004746A             call_os_log_internal

__text:000000000004746F             jmp     short loc_4740B     ---return eax

__text:000000000004746F_ZN16IOAccelSysMemory16lockForCPUAccessEP4taskj endp
```

*Table 10. The asm code snippet of IOAccelSysMemory::lockForCPUAccess*

## 4. The benefits of LLDBFuzzer

These are only six of the many vulnerabilities we found through LLDBFuzzer; other crashes are still being analyzed and reported to Apple. As mentioned above, LLDB has a distinct advantage over other bug hunting methods because it can debug almost all the kernel extensions and XNU codes after the required hardware is operational, and it has roots in the built-in debug mechanism of operation systems themselves. Also, it uncovers and probes into the deeper attack surface as well as the normal attack surface.

## 5. Appendix

Refer to chart.

**TREND MICRO[TM] RESEARCH**

Trend Micro, a global leader in cybersecurity, helps to make the world safe for exchanging digital information.

Trend Micro Research is powered by experts who are passionate about discovering new threats, sharing key insights, and supporting efforts to stop cybercriminals. Our global team helps identify millions of threats daily, leads the industry in vulnerability disclosures, and publishes innovative research on new threats techniques. We continually work to anticipate new threats and deliver thought-provoking research.

**www.trendmicro.com**