LLDB FreeBSD Kernel Module Improvement

1st Sheng-Yi Hung
Computer Science and Information Engineering
National Taiwan Normal University
Taipei, Taiwan
aokblast@FreeBSD.org

***Abstract*—This paper introduces the low level debugger (LLDB) kernel module debug facility for the FreeBSD kernel. The current functional status of LLDB within the FreeBSD kernel is attributed to contributions from [1] and the collaborative efforts of the community. Key functionalities include core dump parsing and memory context building for the coredump, specifically integrated into the process plugin within LLDB for the FreeBSD kernel. This enhancement equips LLDB for effective post-mortem debugging on the FreeBSD kernel. While the implementation of the process plugin has been successfully completed, the paper emphasizes the imperative need to implement the DynamicLoader plugin for the kernel loader. This plugin plays a critical role in loading the symbol file of the kernel module, ensuring comprehensive parsing of symbols for loadable kernel modules. Additionally, given the potential existence of the kernel module as either a relocatable file (for x86) or a shared object (for ARM), the implementation should confirms the usability of both types of ELF format kernel modules.**

***Index Terms*—LLDB, Kernel Module, FreeBSD.**

I. Introduction

The landscape of kernel-level debugging in the FreeBSD operating system has witnessed notable improvement, primarily through the introduction of the LLDB kernel module debug facility. The current functional capabilities of LLDB within the FreeBSD kernel owe their existence to the contributions detailed in [1] and the collaborative efforts of the community. Noteworthy functionalities encompass register and core dump parsing, coupled with memory context building tailored for coredumps. These capabilities have been great implemented in the process plugin within LLDB, offering an enhanced framework for post-mortem debugging on the FreeBSD kernel.

While the implementation of the process plugin represents a significant stride forward, this paper underscores the imperative necessity to extend LLDB's capabilities further by implementing the DynamicLoader plugin tailored for the kernel loader. The DynamicLoader plugin assumes a pivotal role in loading the symbol file of the kernel module, thereby ensuring a comprehensive parsing of symbols for loadable kernel modules. Given the diverse nature of kernel modules, which may manifest either as relocatable files (for x86) or shared objects (for ARM), the implementation strives to confirm the adaptability of LLDB to both types of ELF format kernel modules. The shared object has been well implemented as it is the common format for libraries in userspace. In the relocatable file format, there are some work needs to be done in the ObjectFile. This critical extension is poised to augment LLDB's functionality, providing a more robust and versatile debugging environment for the FreeBSD kernel.

The outcomes of this paper's work have been merged into the LLVM codebase and can be found in the following pull request: https://github.com/llvm/llvm-project/pull/67106.

The following sections contains the background of this paper, the implementation of this paper, and some works has not been done when the paper is written.

II. Background

A. Toolchain and Debugger

The toolchain serves as the foundational element for operating system (OS) development, acting as the essential framework that developers employ to build both the OS userspace and its kernel. An integral facet of this toolchain is the debugger, which enhances developers' capabilities, providing a refined experience for the analysis of program behavior.

In the FreeBSD operating system, there are currently two officially supported debuggers: GDB for the kernel (KGDB) [2] and LLDB [3]. Originally the debugger of choice for both FreeBSD userland and kernel space, KGDB delivers comprehensive functionality, allowing developers to seamlessly debug kernel coredumps within user space. The introduction of LLDB to FreeBSD coincided with a pivotal transition in the binding toolchain, shifting from GNU to LLVM. Consequently, this transition required the re-implementation of all infrastructure previously established in the GNU toolchain, now residing within the LLVM codebase. This integration has proven successful, extending across both FreeBSD kernel space and userspace.

However, despite these contributions, a challenge persists within the kernel space: LLDB has no ability on parsing symbols for loadable kernel modules [4]. This paper addresses this issue by proposing and implementing the dynamicloader plugin in LLDB, aiming to enhance its capability to accurately parse symbols of loadable kernel modules.

B. Kernel Module Loading in LLDB: A Comparison with Userspace Shared Libraries

The analogy between kernel modules and shared libraries in userspace is underscored by the shared characteristic of dynamic loading. In userspace, a shared library is loaded using system calls like mmap, facilitated by a dynamic loader. The independence from the kernel allows for the existence of multiple implementations, such as ld.ldd [5] and gold.

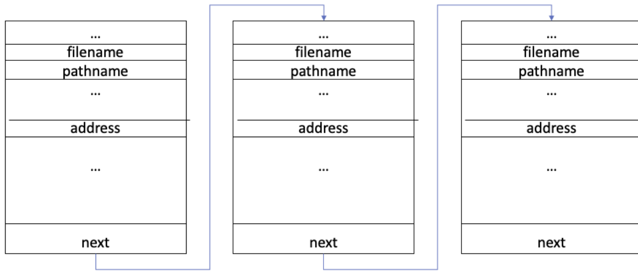

Fig. 1. Structure of linker_files

In contrast, within the kernel space, dynamic loading is managed by the kernel itself, specifically through the kld [4] subsystem in FreeBSD. The kld subsystem is tasked with critical responsibilities, including the mapping of the kernel modules' memory image into the kernel's address space. Additionally, it oversees the registration and initialization of kernel modules within the kernel framework.

Consequently, the process of loading a kernel module in LLDB for the FreeBSD kernel closely aligns with the implementation of a dynamic loader plugin tailored for the FreeBSD kernel. This parallel underscores the conceptual symmetry between userspace and kernel space in the context of dynamic loading.

Moreover, recognizing the inherent similarities between userspace and kernel space, certain functionalities established in userspace, such as ELF parsing, can be seamlessly repurposed for application in the kernel space with minimal modification. This cross-applicability not only streamlines development efforts but also emphasizes the potential for shared techniques in enhancing the functionality of LLDB within the context of FreeBSD kernel module analysis.

C. The current infrastructure in LLDB

In prior efforts, the community developed a plugin named "ProcessFreeBSDKernel" designed to parse kernel coredumps and extract memory information from them. This plugin employs either libfbsdvmcore [6] for cross-platform debugging or libkvm [7] for debugging the kernel within the FreeBSD environment. Through the utilization of this plugin, the community gains the capability to conduct post-mortem debugging, facilitating a basic analysis of kernel-related bug. However, the lack of dynamic loader plugin for the kernel still makes a stunt on symbol parsing, making LLDB cannot replace KGDB as a primary tool for kernel debugging.

III. DynamicLoaderFreeBSDKernel

This section delves into the intricate details of our module's implementation, containing the design, implementation, modifications to the ELF file parsing in the original LLDB, and the integration into the FreeBSD source code.

A. Design

In the present study, we introduce a novel LLDB plugin named "DynamicLoaderFreeBSDKernel." The primary objective of this plugin is to facilitate the integration of LLDB with the FreeBSD kernel module. The plugin undertakes a series of essential steps to enable LLDB support for the FreeBSD kernel. These steps are delineated as follows:

1) Initialization by ProcessFreeBSDKernel: The plugin is loaded by the ProcessFreeBSDKernel when the designated target is identified as a kernel image or a coredump.
2) Find and Verification of Kernel Memory Address: The plugin acquires and verifies the memory address of the kernel within the coredump, recognizing the potential variability in coredump structures that may not strictly adhere to the characteristics of a kernel coredump.
3) Parsing of Kernel-Loaded Module Addresses: A parsing operation is performed on the linked list within the kernel, which encapsulates the loading addresses and name of all loaded kernel modules.
4) Relocatable File Handling: For each file encountered during parsing, the plugin distinguishes whether it is a relocatable file. In the case of relocatable files, the loading address is established, and the process is concluded.
5) Adjustment of In-Memory Section Addresses: For files that are not relocatable, the plugin iteratively adjusts the addresses of all in-memory sections before concluding the processing and returning the relevant results.

By implementing these steps, the DynamicLoaderFreeBSDKernel plugin serves as a pivotal component in extending LLDB's capabilities to seamlessly interact with and support the FreeBSD kernel module.

B. Plugin Implementation

```
// DynamicLoaderFreeBSDKernel.cpp

+while (current_kld != 0) {
+    addr_t kld_filename_addr =
+        m_process->ReadPointerFromMemory(
    current_kld + kld_off_filename, error);
+    addr_t kld_pathname_addr =
+        m_process->ReadPointerFromMemory(
    current_kld + kld_off_pathname, error);
+
+    m_process->ReadCStringFromMemory(
    kld_filename_addr, kld_filename,
+                                    sizeof(
    kld_filename), error);
+    m_process->ReadCStringFromMemory(
    kld_pathname_addr, kld_pathname,
+                                    sizeof(
    kld_pathname), error);
+    kld_load_addr =
+        m_process->ReadPointerFromMemory(
    current_kld + kld_off_address, error);
+
+    kmods_list.emplace_back();
+    KModImageInfo &kmod_info = kmods_list.back();
+    kmod_info.SetName(kld_filename);
+    kmod_info.SetLoadAddress(kld_load_addr);
+    kmod_info.SetPath(kld_pathname);
+
```

```
+     current_kld =
+         m_process->ReadPointerFromMemory(
    current_kld + kld_off_next, error);
+     if (kmod_info.GetName() == "kernel")
+       kmods_list.pop_back();
+     if (error.Fail())
+       return false;
+   }
```

Listing 1. Main loop for the module parsing

Within this subsection, the focus is on locating and validating the loaded address of the kernel and kernel module. The current methodology involves reading the loaded address from the kernel's ELF file, a viable approach given the absence of kASLR in the FreeBSD kernel. The kernel loading address is considered as the virtual address of the first segment in the ELF file. Through alignment with KGDB and subsequent confirmation, this address is corroborated. Memory information within the loaded address is extracted from a coredump using the previously discussed "ProcessFreeBSDKernel" plugin, verifying the OS type and ELF header format. With the validated loading address, symbol information within the kernel can be parsed.

Subsequent to confirming the kernel's loaded address, the next imperative is to ascertain the loaded kernel module names and their respective addresses. This information is graped from the kernel by parsing the "linker_files" variable, depicted in Figure 1. This variable constitutes a linked list exposed by the kernel, documenting structures that record the names and addresses of currently loaded kernel modules. The parsing process, facilitated by the "ProcessFreeBSDKernel" plugin, involves iterating through this linked list. In fig 1 we show the loop for the algorithm. First, we get the address of member from the offset by "ReadPointerFromMemory". Then we get the content of member by "ReadCStringFromMemory". We continue by read the address of the next structure until the end. For each kernel module, the associated object file is located in the system path using the provided name in "linker_files." Simultaneously, the symbol file is attached to the kernel module object file. Notably, kernel modules may exist in two forms: relocatable files for x86 and shared libraries for aarch64. For relocatable files, which is without segments, we just set the loaded address. Conversely, for shared objects, the addresses of each segment are aligned with those of the kernel. Then we add the symbolized module in the module list in LLDB. Consequently, the kernel module is comprehensively configured, with precisely identified symbols and structures amenable to LLDB.

C. Modification of ELF Parsing Plugin in original LLDB

```
// ObjectFileELF.cpp

-    if (ObjectType == ObjectFile::Type::
     eTypeObjectFile && Segments.empty() && (H.
     sh_flags & SHF_ALLOC)) {

+    // When this is a debug file for relocatable
     file, the address is all zero
+    // and thus needs to use accumulative method
+    if ((ObjectType == ObjectFile::Type::
     eTypeObjectFile ||
+         (ObjectType == ObjectFile::Type::
    eTypeDebugInfo && H.sh_addr == 0)) &&
+        Segments.empty() && (H.sh_flags & SHF_ALLOC)
    ) {
```

Listing 2. Modification of symbol file for relocatable file

During the implementation of this plugin, a challenge emerged wherein the kernel module's address was inaccurately recorded in LLDB. Subsequent experimentation revealed that the issue lay in the symbol file attached to the kernel module. Specifically, the section address of the relocatable file was consistently recorded as zero. In the original LLDB implementation, a prefix sum of offsets in the section was employed, neglecting consideration for the symbol file in relocatable files. In our modification, depicted in Listing 2, we account for the symbol file with a zero section address as relocatable file, necessitating an accumulative method.

```
// ObjectFileELF.cpp

+  if (GetType() == ObjectFile::eTypeObjectFile) {
+   for (SectionHeaderCollIter I = std::next(
    m_section_headers.begin());
+        I != m_section_headers.end(); ++I) {
+     const ELFSectionHeaderInfo &header = *I;
+     if (header.sh_flags & SHF_ALLOC)
+       return Address(GetSectionList()->
    FindSectionByID(SectionIndex(I)), 0);
+   }
+   return LLDB_INVALID_ADDRESS;
+ }
```

Listing 3. Add base addr for relocatable file

Another critical modification involves the base address, employed as the preferred loaded address for ELF files in LLDB. While this information is readily available for shared objects and executables, relocatable kernel modules lack a specified base address because normal relocatable file cannot be loaded. However, given the adjustments made to the load address in the kernel module, we ascertain a safe base address for relocatable kernel modules. As illustrated in Listing 3, we iteratively search for sections in the file containing the PT_ALLOC flag, designating the first section with this flag as the loaded address.

```
// ObjectFileELF.cpp

  case llvm::ELF::ET_EXEC:
-    // 2 - Executable file
-    // TODO: is there any way to detect that an
    executable is a kernel
-    // related executable by inspecting the program
     headers, section headers,
-    // symbols, or any other flag bits???
-    return eStrataUser;
+    {
+      SectionList *section_list = GetSectionList();
+      if (section_list) {
+        static ConstString loader_section_name(".
    interp");
+        SectionSP loader_section =
+            section_list->FindSectionByName(
    loader_section_name);
+        if (loader_section) {
+          char buffer[256];
+          size_t read_size =
```

```
+            ReadSectionData(loader_section.get(),
      0, buffer, sizeof(buffer));
+
+          // We compare the content of .interp
    section
+          // It will contains \0 when counting
    read_size, so the size needs to
+          // decrease by one
+          llvm::StringRef loader_name(buffer,
    read_size - 1);
+          llvm::StringRef
    freebsd_kernel_loader_name("/red/herring");
+          if (loader_name.equals(
    freebsd_kernel_loader_name))
+            return eStrataKernel;
+        }
+      }
+      return eStrataUser;
+    }
```

Listing 4. ELF kernel type modification

The final modification, outlined in Listing 4, pertains to kernel format ELF files. In the ELF design, the kernel is conceived as a conventional executable employing bare metal as its runtime. Unlike formats such as Mach-O, ELF lacks a flag indicating that an executable is a kernel. In the original LLDB, this remained in the TODO list. However, our observation reveals that the FreeBSD kernel employs a spurious file named ”/red/herring” in the interpreter section. This anomaly allows us to conclusively identify ELF files associated with FreeBSD kernels.

D. Integration to FreeBSD src

Until the time this paper is written, LLVM still doesn't put our patch into release. Thus the following modification I made may not be the final result appear in the FreeBSD src. The endeavors detailed in the preceding section have done within the LLVM source. Given the distinct build systems employed by FreeBSD (Makefile project) and LLVM (CMake project), additional efforts were requisite to seamlessly integrate the modifications into the FreeBSD project. The entirety of the modifications is meticulously documented in reference [8]. One specific alteration that warrants emphasis is found in ”lib/-clang/include/Plugins/Plugins.def.” Although the code remains compilable without this modification, its absence precludes its runtime discovery of this plugin within LLDB. Notably, there is no explicit indication to amend this file, leading to a substantial time investment in diagnosing and resolving this issue.

IV. Result

To illustrate the efficacy of our modifications, we present demonstrations in Listings 6 and 7.

In Listing 5, the original module list only parsed the kernel itself. Subsequent to our modifications, as evident in Listing 6, LLDB successfully recognizes the kernel module, its associated symbol file, and the corresponding load address.

```
(lldb) image list
[  0] 013C9080-98CC-F2F1-237C-AFAA727809F7-8144DCF4
    0xffffffff80200000 /boot/kernel/kernel
      /usr/lib/debug/boot/kernel/kernel.debug
```

Listing 5. Original module list

```
(lldb) image list
[  0] 013C9080-98CC-F2F1-237C-AFAA727809F7-8144DCF4
    0xffffffff80200000 /boot/kernel/kernel
      /usr/lib/debug/boot/kernel/kernel.debug
[  1] F67FE954-8379-A3D1-848C-946C8C239092-6EAD3444
    0xffffffff81f50000 /boot/kernel/zfs.ko
      /usr/lib/debug/boot/kernel/zfs.ko.debug
[  2] C580C510-DF33-FB7D-81AC-7989646C889C-26A50D5F
    0xffffffff82644000 /boot/kernel/cryptodev.ko
      /usr/lib/debug/boot/kernel/cryptodev.ko.debug
[  3] 34D8ADD5-836D-7AB7-F62E-0C35987F19D0-9659DE07
    0xffffffff82b18000 /boot/kernel/intpm.ko
      /usr/lib/debug/boot/kernel/intpm.ko.debug
[  4] B90304EB-A9FE-5495-9B77-E92790CF6EAF-FC35AA21
    0xffffffff82b1c000 /boot/kernel/smbus.ko
      /usr/lib/debug/boot/kernel/smbus.ko.debug
...
```

Listing 6. Module list after modification

Furthermore, as portrayed in Listing 7, we present a demonstration involving the extraction of symbol information for the ”cryptodev_mod,” corresponding to the symbol from the ”cryptodev.ko” kernel module. The results showcase the comprehensive recognition of the symbol and its underlying structure by LLDB.

```
(lldb) p cryptodev_mod
(moduledata_t) $0 = {
  name = 0xffffffff8264718f "cryptodev"
  evhand = 0xffffffff82644000 (cryptodev.ko`
      cryptodev_modevent at cryptodev.c:1281)
  priv = 0x0000000000000000
```

Listing 7. Demo of the symbol information

These demonstrations underscore the successful enhancement of LLDB's capabilities through our proposed modifications, enabling the precise identification and parsing of kernel modules, symbol files, and associated load addresses.

V. Conclusion

The introduction of the LLDB kernel module debug facility has successfully enabled LLDB to recognize the kernel loader and load kernel modules recorded in coredumps through the ProcessFreeBSDKernel plugin. This accomplishment represents a critical advancement in LLDB's capabilities, particularly in the domain of post-mortem debugging within the FreeBSD kernel.

Moreover, our contributions have transcended the conceptual realm and have been concretely integrated into the LLVM source code. The successful merger of these modifications awaits release, as well as integration into the FreeBSD source code.

VI. Future work

Several avenues for future research and development are listed, involving further modifications to both LLVM source code and the FreeBSD kernel:

1) Kernel executable hash section loading: Presently, the verification of kernel matching with a given coredump relies solely on the ELF header. To enhance this verification, an additional step can be introduced by comparing the .note.gnu.build-id section between the object file and the kernel image. This section contains the hash

value of the file. However, our observations indicate that the kernel may not load this section, resulting in its absence in the coredump. Addressing this limitation is imperative for an effective matching mechanism. The concrete solution for this is to modify the bootloader to load this section.

2) Hash value of kernel module: Extending the hash value checking concept to kernel modules presents another prospective avenue. This entails comparing the version of a kernel module in the filesystem with the version recorded in the coredump. By implementing such a mechanism, a more comprehensive validation of kernel modules can be achieved.
3) Live debug support: The current framework focuses on post-mortem debugging, limiting its applicability to scenarios following a system crash. To broaden its utility, support for live debugging can be incorporated. This entails setting breakpoints when kldload and kldunload are invoked. Upon triggering these breakpoints, the "linker_files" can be parsed anew, facilitating the loading of newly added kernel modules during live debugging sessions. This evolution would significantly enhance the versatility and real-time applicability of the plugin.

REFERENCES

[1] "ProcessFreeBSDKernel", LLDB FreeBSD Kernel Debugger. (Online) Available: https://www.moritz.systems/blog/lldb-freebsd-live-kernel-debugging-support
[2] "KGDB Manual". (Online) Available: https://man.freebsd.org/cgi/man.cgi?query=kgdb&sektion=1
[3] "LLDB Manual". (Online) Available: https://man.freebsd.org/cgi/man.cgi?query=lldb&sektion=1
[4] "KLD Manual". (Online) Available: https://man.freebsd.org/cgi/man.cgi?query=kld&sektion=4
[5] "ld.ldd". (Online) Available: https://man.freebsd.org/cgi/man.cgi?ld(1).
[6] "libfbsdvmcore". (Online) Available: https://github.com/Moritz-Systems/libfbsdvmcore
[7] "libkvm". (Online) Available: https://man.freebsd.org/cgi/man.cgi?query=kvm&sektion=3&manpath=FreeBSD+5.3-RELEASE
[8] Merge Process from LLVM to FreeBSD. (Online) Available: https://github.com/aokblast/freebsd-src/commit/b4e8232961a038d8a4d29df442e4826de7e3da34#diff-30f905803372e92aba2018edcd3a56a61c9e25653e7e1215cae7f84951391f6f